CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76506 CARLOS EDUARDO ANTAO
24CC2502 - 1 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
850391046
533774641
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
VASSOURA
TOMATE
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76506 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76663 DANILO DEPETRIS SOARES
24CC2502 - 2 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
659237176
840052314
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
CREME
FARINHA
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76663 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76513 EDUARDO CAVALHEIRO DIAS
24CC2502 - 3 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
227131217
615078666
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
CERVEJA
DETERGENTE
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76513 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76520 ERIC CORDEIRO
24CC2502 - 4 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
561777190
364577961
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
AMEIXA
FRANGO
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76520 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)

VIVXq80a V: 1.02
76537 FELIPE EIJI KANO
24CC2502 - 5 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
853401074
551933921
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
WHISKIE
FIGO
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76537 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76544 FELIPE JESMIEL LEITE
24CC2502 - 6 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
561563439
686443837
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
TOMATE
UVA
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76544 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76551 GABRIELA BERNARDO PASQUALOTTO
24CC2502 - 7 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.
- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.
- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).
- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.
- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:
- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.
- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).
- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.
- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.
- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.
- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
713429636
232142169
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
TOMATE
ALFACE
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76551 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76737 GRAZIELLY A. L. VALICOSKI
24CC2502 - 8 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
859287054
232692289
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
AMEIXA
CERVEJA
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76737 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76568 GUILHERME VITOR DA SILVA
24CC2502 - 9 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
822466761
485668024
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
UVA
MAMAO
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76568 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76706 KAWAN DAIKI HASEGAWA
24CC2502 - 10 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
515670975
493396542
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
AMEIXA
MAMAO
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76706 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76575 LAURA PACHECO DELATTRE
24CC2502 - 11 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
824021105
511407391
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
WHISKIE
FRANGO
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76575 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)

VIVXq80a V: 1.02
76694 LAURO HENRIK RUTZ
24CC2502 - 12 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
849761897
353058236
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
BALAS
CERVEJA
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76694 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação
07/10/2024 - 07:40:43.8
Matemática aplicada
Prof Dr P Kantek (pkantek@gmail.com)

VIVXq80a V: 1.02
76582 LUIZ MIGUEL OLINEK
24CC2502 - 13 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
599122053
334301132
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
SFIHA
CERVEJA
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76582 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76599 NICOLE DA SILVA DE AZEVEDO
24CC2502 - 14 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
393979392
622168976
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
ABOBORA
AMEIXA
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76599 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76601 PAULO JOSE FREIRE CORREA
24CC2502 - 15 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
894180828
401457380
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
AMEIXA
CERVEJA
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76601 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)

VIVXq80a V: 1.02
76618 PEDRO HENRIQUE PIEKARSKI DE OL
24CC2502 - 16 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
443438353
805095701
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
CERVEJA
BROCOLI
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76618 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76625 RAUL KNAPKI CUNHA
24CC2502 - 17 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo? Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
649980080
593404533
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
CEBOLA
BROCOLI
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76625 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76632 VICTOR BRONDANI DOS SANTOS
24CC2502 - 18 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
603202897
895859059
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
FEIJAO
FIGO
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76632 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)

VIVXq80a V: 1.02
76649 VICTOR HUGO DOS SANTOS DE CAMA
24CC2502 - 19 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
706456661
623312100
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
ABOBORA
BOLACHA
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76649 - /

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 07/10/2024 - 07:40:43.8
Matemática aplicada Prof Dr P Kantek (pkantek@gmail.com)
VIVXq80a V: 1.02
76656 VITOR EDUARDO DE GOES FONTES
24CC2502 - 20 _____ / _____ / _____

Tabelas Hash

Como medir o desempenho de um algoritmo?

Se você quiser medir o desempenho de um programa de computador, a coisa pode ser um pouco fácil. Basta analisar quanta memória é necessária para sua execução, e sobretudo quanto tempo ele demora. Embora a humanidade ainda não saiba muito bem o que é o tempo, aqui a saída é fácil: basta ligar um cronômetro, rodar o programa e quando ele (o programa) acabar, basta efetuar a leitura no cronômetro. Parece fácil, mas existem alguns problemas:

- O ambiente (hardware, sistema operacional, arquitetura, linguagem de programação utilizada) tem profunda influência no desempenho de um programa. Por exemplo, se você comparar o mesmo programa feito em C++ e feito em Python, não causará surpresa se ele o primeiro for 10 vezes mais rápido que o segundo.

- Para medir como sugerido, você tem que escrever, testar, depurar e finalmente implementar o algoritmo através de um programa de computador. Nós gostaríamos de ter uma avaliação ANTES (ou melhor ainda, sem ter que) de implementar. Esta avaliação prévia sobre dois algoritmos diferentes pode lhe orientar sobre qual deles implementar.

- O mesmo programa pode (deve) ter um desempenho para processar 1.000 ítens e outro desempenho muito diferente para processar 1.000.000 de ítens. É até esperado que isto aconteça. Então a medida não pode ser absoluta, devendo ser relativizada ao tamanho da massa de dados que o algoritmo manuseia.

Teoria da complexidade

Complexidade neste contexto não é o sentido usual da palavra (*algo complicado ou difícil*) e sim é uma medida do desempenho de um algoritmo. A teoria da complexidade vai definir algumas poucas funções matemáticas em função do tamanho da entrada (sempre definido pela variável n) e vai nos ajudar a associar praticamente qualquer algoritmo a uma dessas funções. Essa associação será facilmente obtida pela aplicação de algumas regras:

- Demora unitária: Esta regra diz que não importa o que a instrução faça, ela sempre demora 1 unidade (Isto é uma simplificação e tanto: uma divisão real tende a demorar muito mais do que uma soma inteira).

- Instruções fora de laços (enquanto, para e repita), podem ser desprezadas. Isto significa que apenas instruções internas a laços devem ser contadas.

- Constantes multiplicativas podem ser ignoradas. Assim se o algoritmo A demora um tempo x e o algoritmo B demora $3x$, para efeitos desta análise, ambos demoram a mesma coisa.

Após esta aplicação, tenta-se alocar o algoritmo em estudo a uma das principais classes existentes, que são identificadas por um formalismo chamado $Big-O$. A letra "O" em Big-O é uma abreviação da palavra alemã "*Ordnung*", que significa "ordem"ou "ordem de grandeza". Em outras palavras, a notação Big-O nos dá uma ideia da ordem de crescimento de uma função, ou seja, quão rápido ela cresce em relação a outra função quando os valores de entrada se tornam muito grandes.

Principais classes

$O(1)$ Esta classe de algoritmo tem desempenho constante, independente do tamanho da entrada (que lembrando, é representada pela letra n).

$O(n)$ Aqui a demora é linearmente proporcional ao tamanho da entrada. Por exemplo, se um algoritmo desta classe demorar 1 hora para processar 1.000.000 de registros, ele demorará 2 horas para processar 2 milhões.

$O(n^2)$ Agora as coisas começam a ficar mais demoradas. Quando n cresce, a demora cresce proporcional a n^2. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 4.

$O(n^3)$ Agora as coisas começam a ficar muito mais demoradas. Quando n cresce, a demora cresce proporcional a n^3. No exemplo acima, ao dobrar a massa de entrada, o tempo é multiplicado por 9.

$O(k^n)$ Para esta classe, só é possível resolver instâncias muito pequenas: os chamados $toy-domains$. No exemplo acima, supondo $k = 2$, a demora seria de 2^{1000}, algo impossível de pensar.

Existem outras classes, mas antes de estudá-las, precisamos conhecer e trabalhar com o conceito de **logaritmo**. Não confundir com algoritmo, são coisas completamente diferentes.

O logaritmo é como uma espécie de "inverso" da potenciação. Em vez de perguntar "Quanto é 2 elevado a 3?", o logaritmo pergunta "Qual o expoente que eu preciso colocar em 2 para obter 8?". Em outras palavras atente que se $a^c = b$ então $\log_a b = c$. Um exemplo: qual o logaritmo de base 2 de 1000 ? Reescrevendo a pergunta, quantas vezes 2 deve ser multiplicado por 2 até chegar em 1000 ? A resposta é que um valor próximo a 10. (Na verdade 9.9657). Este numero pode ser encontrado examinando a tabela a seguir:

$2^0 = 1$	$\Rightarrow$	$log_2 1 = 0$
$2^1 = 2$	$\Rightarrow$	$log_2 2 = 1$
$2^2 = 4$	$\Rightarrow$	$log_2 4 = 2$
$2^3 = 8$	$\Rightarrow$	$log_2 8 = 3$
$2^4 = 16$	$\Rightarrow$	$log_2 16 = 4$
$2^5 = 32$	$\Rightarrow$	$log_2 32 = 5$
$2^6 = 64$	$\Rightarrow$	$log_2 64 = 6$
$2^7 = 128$	$\Rightarrow$	$log_2 128 = 7$
...		...
$2^{10} = 1024$	$\Rightarrow$	$log_2 1024 = 10$

Os logaritmos já foram muito importantes na ciência. Desde sua invenção no século XVI (Napier, Inglaterra), até o uso das réguas de cálculo em plena década de 1970 ao permitirem simplificar muito os cálculos. Veja esta imagem

Há 2 classes muito comuns na teoria da complexidade que são $O(log_2 n)$ e $O(n \times log_2 n)$. Para entender este conceito, vamos imaginar um array de n elementos. Suponha que um computador consegue realizar 10 operações básicas por segundo. Quanto demorariam os algoritmos em cada classe ?

T	Classe	Classe	Classe	Classe	Classe
N	$O(\log n)$	$O(n)$	$O(n.logn)$	$O(n^2)$	$O(n!)$
10	0.3seg	1 seg	3.3 seg	10 seg	4.2 dias
100	0.6 seg	10 seg	66.4 seg	16.6 min	*
1000	1 seg	100 seg	996 seg	27.7 horas	**

*=2.9×10^{144} anos
**= 1.27×10^{2559} anos

Alguns algoritmos

Acompanhe

Algoritmo	desempenho
Busca linear	$O(n)$
Busca binária	$O(\log_2 n)$
Busca hash	$O(1)$
Ordenações (bolha, seleção, inserção...)	$O(n^2)$
Problema da mochila	$O(n^2)$
Multiplicação de matrizes	$O(n^3)$
Quick sort	$O(n.\log_2 n)$
Menor caminho (Floyd-Warshall)	$O(n^3)$
Caixeiro viajante	$O(n!)$

Deve-se notar que pequenas (ou nem tanto) mudanças em algum algoritmo podem deslocá-lo de uma classe para a outra. Aqui a maravilhosa beleza do estudo de algoritmos.

Tabelas hash

Se alguém percebeu na tabela acima, existe uma busca que garante $O(1)$. Traduzindo em palavras, não importa o tamanho da área de busca, gasta-se uma quantidade unitária de tempo para localizar um item lá.

Vamos a um exemplo literal. Suponha um mercado no qual o caixa tem um caderninho com os produtos e seus preços. A cada compra, ele precisa consultar o caderninho. Se este não estiver ordenado vai demorar n a cada item (aqui, n é a quantidade de produtos à venda). Se a lista estiver ordenada, a busca pode retroceder a $O(\log_2 n)$. Vamos a alguns tempos. Se o mercado tem 10.000 produtos e o caixa pode ler 1 linhas por segundo, o primeiro caso demora 2 horas e meia e o outro 14 segundos. Mas ainda assim, é muito. Queremos algo mais rápido.

Uma tabela hash é uma estrutura de dados que usa uma função hash ($h(chave) = indice$). Traduzindo em palavras uma função hash recebe uma chave – no exemplo, o nome do produto – e devolve a posição desse item (=chave+conteúdo) em uma tabela convencional.

Existem alguns requisitos para uma função assim:

- Deve ser consistente. Sempre que se entregar um valor à função hash, deve ser retornado o mesmo índice. Esta característica também tem o nome de determinístico:

- Baseada em funções matemáticas (uma das principais é o módulo) tão complexas quanto se queira.

- Mapear diferentes chaves para índices diferentes, mesmo que a mudança da chave seja sutil (muito pequena).

- Rápido de calcular: O cálculo do hash deve ser eficiente, mesmo para grandes entradas.

- Resistente a colisões: É improvável (pouco provável entradas produzam o mesmo valor hash.

- Avalanche: Pequenas mudanças na entr grandes mudanças no valor hash.

- O formato do resultado numérico deve ser padronizado (em tamanho e abrangência, por exemplo.

O requisito de pouca colisão nem sempre pode ser assegurado, e este é um dos possíveis problemas de funções hash. Mais sobre isso, adiante.

Outros usos da função hash

Essa idéia é tão boa que ela acabou sendo re-utilizada para outras aplicações. Uma bem importante é conhecida como **assinatura digital**. Trata-se de garantir a integridade de um objeto binário. Pense num arquivo WORD que você criou. Aplicando uma função hash ao arquivo tem-se um resultado numérico. Se você mudar um único caractere no arquivo word, reaplicando a mesma função hash certamente o resultado numérico será diferente. Se você mandar o arquivo para um amigo e o receber de volta alguns dias depois, a função hash pode garantir a você que o arquivo não foi modificado.

Um outro uso espetacular da função hash é na segurança de chaves. Aqui se usa uma característica

importante do conceito hash que é a unilateralidade. Assim é fácil calcular o índice a partir da chave, mas deve ser quase impossível fazer o caminho inverso. Em um arquivo de chaves é comum (o MySQL faz isso) guardar não a senha, mas sim o hash(senha). Cada vez que um usuário tenta se logar, é aplicada a função hash à senha fornecida e é comparado o resultado numérico com o resultado numérico guardado no arquivo. Se a comparação der igual, supõe-se que é o usuário dono da senha. A vantagem ? O bandido pode roubar o arquivo das "senhas" que ele não vai ter nada em mãos, apenas um bando de números inúteis.

Uma derivação desta aplicação são as árvores de Merkle, mecanismo usado no ambiente Bitcoin, para garantir que partes do blockchain recebidos pelos participantes estão inalterados. O mecanismo de Merkle, garante que partes do blockchain possam ser certificados ainda que o todo não esteja disponível (veja vivxn55).

Aproveitando a citação ao ambiente bitcoin, saiba-se que o mecanismo de mineração de novos bitcoins exige a busca de blocos binários quaisquer que tenham função hash começando com um certo número de zeros.

Problemas da função hash Um primeiro problema, é que o resultado numérico da função hash não preserva a ordem original da tabela de itens. Esta característica implica em que se for necessária uma listagem ordenada de itens, de duas uma: ou uma tabela adicional ordenada é mantida (o que exige a duplicação das tabelas: uma de hash e outra de índices), ou a tabela hash tem que ser ordenada antes de cada uso.

Um segundo problema, este mais grave é a colisão.

Colisão Para estudar a colisão, vamos definir algumas medidas. Voltando ao exemplo do mercado com 10.000 itens, a chave do universo é o nome do produto. Supondo nomes de 30 caracteres e o uso do alfabeto latino para o registro dos nomes, tem-se um **espaço de chaves** de $26^{30} = 2.813198901^{42}$, um número enorme, por qualquer régua.

Já o espaço de índices é 10.000, mas por razões que veremos a seguir é conveniente fazê-lo bem maior, digamos 100.000. Se qualquer maneira é evidente que $2.81^{42} >> 100.000$. Então é impossível garantir que dois produtos distintos não gerem o mesmo índice. Quando isto ocorre, tem-se uma colisão (duas chaves distintas, quando entregues à função hash, geram o mesmo índice). O que se busca aqui, é diminuir a probabilidade de que isto ocorra.

Mas, ocorrendo, este fenômeno tem que ser tratado. Um tratamento simples é localizar o próximo espaço livre no espaço de índices (daí porque passamos de 10.000 para 100.000 no exemplo acima).

Outro tratamento mais sofisticado é iniciar uma lista encadeada (lembram o exercício vivxq43 e vivx343 que já fizemos ?) conectando todas as chaves que geraram o mesmo índice.

Outra possibilidade é usar uma segunda função hash quando a primeira der colisão.

Enfim, o universo é o limite: eis a beleza da programação.

Alguns exemplo Vamos exercitar tudo o que se viu.

Exemplo naïve A palavra francesa naïve significa ingênuo, inocente ou bobinho, e é lida com sílaba tônica no i. Suponha o caderno de produtos do mercado com a função hash sendo a posição do alfabeto da primeira letra do produto. Assim, o produto maçã seria mapeado para a posição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567891          2
          01234567890123456
```

13 da tabela de produtos. O produto abacate seria mapeado para a posição 1, e quando chegasse o produto abacaxi, teríamos uma colisão. Aqui, claramente teríamos um espaço de chaves muito grande e um espaço de índices de 26: uma receita para o desastre.

Exemplo do CPF Suponha que vamos identificar os alunos do CEP (5.000 alunos) pelo seu CPF. O CPF pode ser entendido como um inteiro menor que que 1 bilhão (desconsiderados os dígitos verificadores que como sabemos têm outra função e aqui podem ser desprezados). Já o espaço de índices é de 5.000, mas vai ser ampliado para 100.000. Uma função razoável de hash aqui poderia ser `CPF % 99991` sendo que 99991 é o primo mais próximo a 100000. Então, a função hash neste problema seria:

```
function hash(cpf)
   return cpf % 99991
```

Como se sabe da função módulo, o espaço de índices aqui é 0..99990. Nada se diz sobre o que fazer em caso de colisão, mas considerando que o espaço virtual de índices (100000) é muito maior do que o espaço real de índices (10000) a estratégia de usar o próximo espaço livre pode ser bem interessante.

☞ Para você fazer

Ache o hash dos alunos de CPF, usando a função hash acima:

```
439779204
478523222
```

Responda aqui:

1	2

Exemplo do mercado agora mais sofisticado Vamos melhorar o exemplo do mercado. Ao invés de usar apenas a primeira letra (o que é ruim, já que implica em colisão entre `abacate`, `arroz` e `ameixa`), vamos usar as 10 primeira letras como dígitos em um sistema de numeração de base 26. Usando o que já aprendemos (sistemas de numeração - exercício vivxk04), vamos achar o número em questão e depois obter o resto de algum primo próximo a este número. Fica:

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
01234567891          2
          01234567890123456
```

Note que precisamos mapear o espaço em branco (correspondendo à posição 0 do sistema de numeração. A base agora é 27, já que são 27 "letras" usadas. O algoritmo para achar o número é

```
vetor = [' ABCDEFGHIJKLMNOPQRSTUVWXYZ']
função converte_q_2_10(num, bas)
   v = 0
   j = 0
   para i de tamanho.num-1 até i>=0 faça i=i-1
      para k de 0 até 27 faça
         se num[i]==vetor[k]
            saia
         fim{se}
      fim{para}
       v = v + (k * bas**j))
   fim{para}
   retorne v
fim{função}
```

Esta função em JS é

```
var vet=[" ", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N",
    "O", "P", "Q", "R", "S", "T", "U", "V",
    "W", "X", "Y", "Z"]
function q10(a){
   var v = 0;
   var j = 0;
   var i;
   for(i = a.length-1; i >= 0; i--){
      for(k = 0; k < 27; k++){
         if (a[i]== vet [k]){
            break
         }
      }
      v = v+(k * 27**j)
      j++
   }
   return v
}
```

A mesma função em C++ é

```
#include<iostream>
#include<cmath>
using namespace std;
int q10(string chave){
   string vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   long long int v=0, j=0, i, k;
   for (i=0;i<chave.length();i++){
       for (k=0;k<27;k++){
           if (chave[i]==vet[k]){
               break;
           }
       }
       cout<<k*pow(27,j)<<endl;;
       v=v+(k* (long long int)(pow(27,j)));
       j++;
   }
   return v;
}
int main(){
   cout<<q10("ABACAXI");
}
```

Antes que você se entusiasme demais com o código C++, há que se considerar os limites numéricos das variáveis inteiras em C++ que não é muito grande.

E em Python é

```
def q10(a):
    vet=" ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    v=0
    j=0
    for i in range(len(a)-1,-1,-1):
        for k in range(27):
            if a[i]==vet[k]:
                break
        v=v+k*27**j
        j=j+1
    return v
print(q10("ABACAXI"))
416710179
```

Obtido este numero, é hora de obter o resto da divisão dele por 99991 (o primo mais próximo de 100000, que é o nosso espaço de índices).

☞ Para você fazer

Ache o hash dos produtos de mercado, usando a função hash acima:

```
FEIJAO
AMEIXA
```

Responda aqui:

3	4

Para você testar seus programas

Se você aplicar este algoritmo nos seguintes dados

CPF= 111111111	índice=21110
CPF= 222222222	índice=42220
CPF= 123456789	índice=67895
produto=RUM	índice=13702
produto=ACUCAR	índice=60272
produto=AMENDOA	índice=44290

502-76656 - /