U Positivo - UTFPR - PUCPr – 11/02/2019 - 12:35:38.0
Prof Dr P Kantek (pkantek@gmail.com)
Estratégias de Programaçaão 1 VIVO666b, V: 1.05 1
- 1 _____ / _____ / _____

Esta folha está baseada no excelente livro BHARGAVA, Aditya. **Entendendo Algoritmos** - um guia ilustrado para programadores e outros curiosos. São Paulo: Novatec, 2017.

Como se mede um algoritmo ?

Um algoritmo consome 2 recursos: tempo de processador e memória. A análise tradicionalmente recai sobre o tempo. Como a demora de um programa depende da linguagem, computador, plataforma, SO, etc, busca-se aqui uma descrição matemática que independa de aspectos de implementação. Até porque, usualmente quer se discutir desempenho ANTES de implementar o algoritmo, isto é, ele ainda não existe.

Esta teoria atende pelo nome (inadequado) de Teoria da Complexidade. Aqui, complexidade é sinônimo de desempenho. Busca-se uma família de funções que descrevam o tempo em função do tamanho da entrada. Para estudar a demora antes do algoritmo ser criado, as simplificações são 2:

- Só se contam as instruções dentre de loops
- Cada instrução demora 1 unidade de tempo, independente do que fizer.

Grosso modo, eis os casos mais comuns para uma entrada de tamanho n

complexidade	explicação
$O(k)$	o algoritmo demora um tempo constante, independente do tamanho da entrada
$O(n)$	o algoritmo demora proporcionalmente ao tamanho da entrada
$O(n^2)$	proporcional ao quadrado da entrada
$O(n^3)$	proporcional ao cubo da entrada
$O(k^n)$	exponencial em relação a entrada
$O(log_2 n)$	proporcional ao logaritmo da entrada
$O(n.log_2 n)$	a entrada vezes o logaritmo da entrada

Alguns exemplos: $O(n)$=leitura e impressão de um arquivo; $O(n^2)$= classificação bolha, classificação por seleção, menor caminho (dijkstra); $O(n^3)$=menor caminho (floyd-warshall); $O(k^n)$=caixeiro viajante; $O(log_2 n)$=busca em uma árvore binária, busca binária (veja a seguir); $O(n.log_2 n)$=quick sort.

Busca Binária

A busca binária é um algoritmo que busca encontrar um elemento em uma lista. A lista deve estar ordenada, e se o elemento pertence à lista, a resposta é o seu índice (ou seja: em que lugar da lista este elemento aparece). Se ele não pertence a resposta é um índice inválido que sinaliza este fato. Nós usamos um algoritmo parecido quando procuramos uma palavra num dicionário, ou olhamos uma lista de aprovados para ver se estamos lá, ou em buscas similares.

Vamos estudar o algoritmo. Suponha que você tem que descobrir um número que eu inventei, entre 1 e 1000. Depois de cada chute eu responderei: acertou, mais alto ou mais baixo, dependendo de onde estiver o número que eu inventei em relação ao seu chute.

Se você começar em 1, 2, 3 e assim por diante, a cada palpite apenas um número é excluído e se eu pensei em um número próximo de 1000 serão quase 1000 tentativas.

A busca binária começa chutando o meio do intervalo: 500. Se eu responder "acima", de cara são eliminados 500 candidatos. O próximo chute deve ser 750, chute este que elimina mais a metade do restante. De metade em metade chega-se ao número escolhido em no máximo 10 tentativas. É muito melhor do que as quase 1000 da estratégia anterior.

De um modo geral a busca binária gasta $log_2 N$ tentativas, onde N é o tamanho da lista. Você pode ter esquecido o que é o logaritmo, mas relembre: Em $log_x Y$ =? a pergunta a responder é quantas vezes x deve ser multiplicado por x até chegar ao valor Y. Daqui que $log_2 1024 = 10$ já que 2 deve ser multiplicado por 2 10 vezes para chegar a 1024. Acompanhe o algoritmo em Python

```
def pesqbin(lista,item):
  baixo=0
  alto=len(lista)-1
  while baixo <= alto:
    metade = (baixo+alto)//2
    chute=lista[meio]
    if chute==item:
      return meio
    if chute>item:
      alto=meio-1
    else:
      baixo=meio+1
  return -1
```

Alocação sequencial e encadeada

Imagine a memória do computador como sendo um enorme armário de gavetas. Cada gaveta corresponde a um byte e pode conter 256 valores (binários) diferentes. Cada gaveta tem um nome (um endereço) e por meio dele que se examina o que tem lá (leitura) ou se modifica esse conteúdo (gravação). Cada vez que você (seu programa) precisa armazenar algum dado ele pede memória ao sistema operacional. Há duas maneiras desse pedido ser feito: memória sequencial e memória encadeada. Na alocação sequencial, os endereços pedidos estão contíguos. Na alocação encadeada não existe contiguidade. Na alocação sequencial:
1. Todos os dados estão juntos. Podem ser processados sequencialmente de maneira muito fácil. Podem também ser acessados individualmente muito rápido. 2. Não é fácil crescer a estrutura, já que o espaço ao final dela pode estar sendo já usado por outrém. Neste caso, todos os dados precisam ser copiados. 3. Inclusões e exclusões são mais demoradas, pois há necessidade de efetuar deslocamentos dos dados remanescentes.
Na alocação encadeada:
1. Os dados estão em qualquer lugar da memória. Para chegar no próximo, o anterior precisa conter um apontador para o próximo. Isto é o encadeamento. 2. A estrutura pode crescer à vontade até o limite da memória. 3. Não é possível acessar um dado diretamente. A cadeia toda tem que ser percorrida. 4. Inclusões e exclusões (depois de localizado o local exato) são muito rápidas.

Recursão

A recursão é um processo de repetir um procedimento sucessivas vezes até resolver o problema. A novidade é que o procedimento é sempre o mesmo (embora com parâmetros cada vez menores). É odiado por muitos programadores, mas adorado por outros tantos. No paradigma funcional (Haskell, Lisp, APL,...) é praticamente o único mecanismo de repetição empregado. Um algoritmo recursivo sempre pode ser reescrito sem recursão (o que chamamos interativo), mas com a palavra Leigh Caldwell do *Stack Overflow: loops podem melhorar o desempenho do seu programa. A recursão melhora o desempenho do seu programador. Escolha o que for mais importante para a sua situação.*

Um programa recursivo tem 3 características:
1. chama a si próprio
2. apresenta um caso básico, quando não há recursão
3. na chamada recursiva, a cada vez o universo é menor, em direção ao caso básico
veja-se um exemplo de uma função escrita na forma recursiva e na interativa

```
def fati(x):
  r=1
  while x>1:
    r=r*x
    x=x-1
  return r

def fatr(x):
  if x>1:
    return x*fatr(x-1)
  else:
    return 1
```

Dividir para conquistar

Suponha que você tem uma grande área de terra retangular para construir canteiros de legumes. Cada canteiro precisa estar em uma área quadrada do maior tamanho possível. Qual a solução ótima ? D-C é uma possível resposta. Como a técnica é sempre recursiva, vamos buscar o caso base: pode ser quando a dimensão maior é múltipla da menor. Nesta hora divide-se o terreno em quadrados e está resolvido o problema. Enquanto este caso base não aparecer vai-se recortar um quadrado de aresta igual à menor aresta e rechamar o código. Acompanhe o algoritmo em Python

```
def dcfazenda(x,y):
  print("gerará "+str(x//y)+" modulos
     de "+str(y)+" por "+str(y))
  if x%y==0:
    return
  else:
    return dcfazenda(y,x%y)
```

Quicksort

É o algoritmo campeão de desempenho para ordenar listas. Lembram-se da importância da lista estar ordenada na busca binária ? Aliás, em francês um computador chama *ordinateur* e em espanhol *ordenador*. Escolhido um elemento aleatório da lista aqui chamado de *pivot* a lista é subdividida em duas sub-listas: a dos elementos menores do que o pivot e outra sublista dos elementos maiores do que ele. O algoritmo é chamado recursivamente para cada uma das sublistas e no final a resposta é a sublista menores (já ordenada) + pivot + sublista maiores (já ordenada). Acompanhe o algoritmo

```
def quick(lis):
  if len(lis)<2:
    return lis
  else:
    pivot=lis[0]
    menor=[i for i in lis[1:] if i<=pivot]
    maior=[i for i in lis[1:] if i>pivot]
    return quick(menor)+[pivot]+quick(maior)
```

No caso médio este algoritmo é muito rápido. Ele só deixa de sê-lo no caso particular de que o vetor já esteja (quase) ordenado. Neste caso o pivot (quase) sempre vai ser o menor elemento e a cada recursão o vetor resultante só será uma unidade menor. Daí que normalmente escolhe-se como pivot alguém do meio do vetor original. O comando melhor seria: `pivot=lista[len(lista)//2]`.

☞ Para você fazer

Conduza uma pesquisa na Internet e faça um resumo pessoal dos seguintes algoritmos/sites:

1. Troca de Chaves Diffie-Hellman
2. Algoritmo HyperLogLog
3. Algoritmo Flajolet-Martin
4. Algoritmo Hash e SimHash (um hash que tem sensibilidade local)
5. o site better explained
6. o algoritmo de similaridade do cosseno
7. o algoritmo da maior subsequência comum
8. algoritmo da mochila (aplicação do alg. guloso)