Universidade Positivo Sistemas de Informação 11/02/2019 - 12:35:19.8
Estruturas de Dados e Arquivos Prof Dr P Kantek (pkantek@up.edu.br)
Algoritmos de ordenação VIVO511b, V: 1.01
1
Exercício : 1 __ / __ / __

Algoritmos de ordenação

- Na década de 70, James Martin sugeriu que 40% do tempo de TODOS os computadores é gasto neste algoritmo.
- O conceito de ordem é fundamental em programação
- Excelente tópico para estudar algoritmos e estudar comportamentos e complexidades

Antes de começarmos a estudar os diversos algoritmos de classificação, vamos definir o problema:

Seja X um conjunto composto por i elementos entre os quais se pode estabecer uma relação de ordem. Dados X_i e X_j com $i \neq j$, sempre pode-se estabelecer $X_i > X_j$ ou $X_i = X_j$ ou $X_i < X_j$.

- X estará em ordem crescente se e somente se $X_i \leq X_j, \ \forall i < j$.
- X estará em ordem estritamente crescente se e somente se $X_i < X_j, \ \forall i < j$.
- X estará em ordem decrescente se e somente se $X_i \geq X_j, \ \forall i < j$.
- X estará em ordem estritamente decrescente se e somente se $X_i > X_j, \ \forall i < j$.

Um algoritmo de ORDENAÇÃO é aquele que recebe um X qualquer (possivelmente desordenado), permuta seus elementos e devolve X em ordem.

Tipicamente fazem parte de X_i um conjunto de informações. Neste caso, haverá uma parte de X_i denominada CHAVE é identificada por k (k=key) pela qual se fará a ordenação. Embora devamos ter em mente a existência dos outros campos, apenas a chave será tratada nos algoritmos.

Os algoritmos de ordenação, grosso modo, podem ser divididos em 2 categorias: aqueles de complexidade $O(n^2)$ e os de complexidade $O(n \times log_2 n)$, ambas as expressões – como sempre – a menos das constantes multiplicativas.

O estudo da primeira família se justifica pelas seguintes razões:
1. Constantes multiplicativas eventualmente menores
2. Abundância de recursos de hardware
3. Irrelevância do desempenho para pequenas instâncias
4. Simplicidade (inclusive conceitual) dos algoritmos envolvidos

Principais algoritmos

Bolha Trata-se do algoritmo mais simples que existe. Pares de vizinhos são comparados e se eles estiverem desordenados, são invertidos. A cada passada, o elemento mais leve é levado para a ponta. Daí o nome.

Bolha* O mesmo algoritmo acima, mas com um pequeno truque para encerrar o algoritmo mais cedo caso o vetor a ordenar seja do tipo "quasi-ordenado".

Inserção O mesmo algoritmo usado pelo jogador de buraco ao ordenar as cartas que recebe. Há um monte de cartas desordenadas. As cartas, uma a uma, vão sendo retiradas desse monte e colocadas em ordem na mão do freguês. Quando o monte terminar, a mão está ordenada.

Seleção O menor elemento do vetor a ordenar é intercambiado com o primeiro elemento do vetor. Depois, o segundo menor com o segundo elemento, e assim por diante.

Shell Tem este nome devido ao seu descobridor, o dr Shell em 1959. Parecido com o BOLHA no sentido de que ele troca vizinhos, mas não próximos e sim distantes. Adicionalmente aos demais algoritmos, este precisa uma seqüência de controle, como operador adicional. Pode ter quantos números se quiser, desde que o último seja sempre 1.

HeapSort Usa como estrutura auxiliar um heap. Tem desempenho excelente, e se o programador souber bem o que é um heap, sua implementação é fácil e o seu conceito mais ainda.

Quick Devido a C.R.Hoare em 1962, este algoritmo é o campeão. Ninguém até hoje conseguiu ser mais rápido que ele. Sua estratégia passa pela escolha de pivots, que dividem o conjunto sendo classificado em 2 partes. Uma contém os elementos menores que o pivot e a outra os maiores. Levando esse procedimento ao limite, o conjunto estará classificado. Este é um excelente exemplo da estratégia "dividir e conquistar".

Algoritmo Bolha

```
1: inteiro I,J,AUX
2: I ← tamanho do vetor a ordenar
3: enquanto (I ≥ 2)
4:     J ← 2
5:     enquanto (J ≤ I)
6:         se V[J − 1] > V[J]
7:             AUX ← V[J − 1]
8:             V[J − 1] ← V[J]
9:             V[J] ← AUX
10:        fim{se}
11:        J + +
12:    fim{enquanto}
13:    I − −
14: fim{enquanto}
```

Note as duas características que este algoritmo tem: uma ineficiência grande aliada à simplicidade do código. A grande maioria dos programadores profissionais, só conhece "de cabeça"este algoritmo.

Algoritmo Bolha-Estrela

```
1: inteiro I,J,AUX
2: lógico MUDOU ← .V.
3: I ← tamanho do vetor a ordenar
4: enquanto (I ≥ 2) ∧ MUDOU
5:     J ← 2
6:     MUDOU ← .F.
7:     enquanto (J ≤ I)
8:         se V[J − 1] > V[J]
9:             AUX ← V[J − 1]
10:            V[J − 1] ← V[J]
11:            V[J] ← AUX
12:            MUDOU ← .V.
13:        fim{se}
14:        J + +
15:    fim{enquanto}
16:    I − −
17: fim{enquanto}
```

A única diferença em relação ao original é a presença da variável MUDOU que determina o fim do loop externo, quando nenhuma mudança mais for feita.

Algoritmo de Inserção

```
1: inteiro I, J, X
2: I ← 2
3: TAM ← Tamanho do vetor a ordenar
4: enquanto (I ≤ TAM)
5:     X ← V[I]
6:     J ← I − 1
7:     enquanto (J ≥ 1) ∧ (V[J] > X)
8:         V[J + 1] ← V[J]
9:         J − −
10:    fim{enquanto}
11:    V[J + 1] ← X
12:    I + +
13: fim{enquanto}
```

Na linha 5, o elemento que vai ser colocado em seu lugar definitivo é sacado do vetor. Com isso ele abre um espaço para remanejamento dentro do vetor. Isso é feito enquanto $V[J] > X$ nas linhas 8 e 9. Se o sujeito que foi sacado é o menor do vetor, a condição do enquanto é encerrada pela outra condição ($J \geq 1$). Depois que o remanejamento foi feito, o elemento sacado volta para o vetor, agora no seu lugar certo (linha 11).

Algoritmo Seleção

```
1: inteiro I,J,COR,INUI,AUX
2: TAM ← Tamanho do vetor a ordenar
3: I ← 1
4: enquanto (I ≤ TAM)
5:     COR ← V[I]
6:     INUI ← I
7:     J ← I + 1
8:     enquanto (J ≤ TAM)
9:         se V[J] < COR
10:            COR ← V[J]
11:            INUI ← J
12:        fim{se}
13:        J + +
14:    fim{enquanto}
15:    AUX ← V[I]
16:    V[I] ← V[INUI]
17:    V[INUI] ← AUX
18:    I + +
19: fim{enquanto}
```

O menor elemento é procurado (linhas 8 a 14). Encontrado (agora ele é apontando por $INUI$, ele é trocado de lugar com o elemento apontado por I (linhas 15 a 17).

Algoritmo Shell

```
1: inteiro k, h, i, x, j
2: TAM ← Tamanho do vetor a ordenar
3: DECR ← {seqüência de controle desejada}
4: para K = 1 até tamanho(DECR)
5:     H ← DECR[K]
6:     I ← H
7:     enquanto (I < TAM)
8:         X ← V[I + 1]
9:         J ← I
10:        enquanto (J ≥ H) ∧ (V[1 + J − H] > X)
11:            V[1 + J] ← V[1 + J − H]
12:            J ← J − H
13:        fim{enquanto}
14:        V[1 + J] ← X
15:        I + +
16:    fim{enquanto}
17: fim{para}
```

Algoritmo Heap Sort

```
1: inteiro I,J
2: TAM ←Tamanho do vetor a ordenar
3: I ← ⌊(TAM ÷ 2)
4: para J = I até 1, passo = −1
5:     desce J,TAM
6: fim{para}
7: para J = TAM até1, passo = −1
8:     V[J] ← exclui J
9: fim{para}
10: devolva V
```

```
1: inteiro função exclui (int J)
2: devolva V[1]
3: V[1] ← V[J]
4: desce (1, J)
```

```
1: função desce (int K, TAM)
2: inteiro J
3: J ← K × 2
4: se (J ≤ TAM)
5:     se (J < TAM)
6:         se (V[J + 1] > V[J])
7:             J + +
8:         fim{se}
9:     fim{se}
10:    se (V[K] < V[J])
11:        AUX ← V[K]
12:        V[K] ← V[J]
13:        V[J] ← AUX
14:        desce (J, TAM)
15:    fim{se}
16: fim{se}
```

Algoritmo QUICK

```
1: função RAP(inteiro I, F)
2: inteiro ANT,DEP,MEI,AUX
3: ANT ← I
4: DEP ← F
5: MEI ← V[⌊(I + F) ÷ 2]
6: repita
7:     enquanto (V[ANT] < MEI)
8:         ANT + +
9:     fim{enquanto}
10:    enquanto (V[DEP] > MEI)
11:        DEP − −
12:    fim{enquanto}
13:    se (ANT ≤ DEP)
14:        AUX ← V[ANT]
15:        V[ANT] ← V[DEP]
16:        V[DEP] ← AUX
17:        ANT + +
18:        DEP − −
19:    fim{se}
20: até (ANT > DEP)
21: se (I < DEP)
22:     RAP (I, DEP)
23: fim{se}
24: se (ANT < F)
25:     RAP (ANT, F)
26: fim{se}
27: fim função
28: TAM ← Tamanho do vetor a ordenar
29: RAP (1,TAM)
```

Desempenho Compare os desempenhos dos algoritmos estudados: [1]

Vetor aleatório

alg	100	500	1K	2.5K
bol	0.6	16	68	429
bol*	0.7	17	70	436
ins	0.3	7	31	198
sel	0.4	10	41	255
shell	0.1	1	2	8
heap	0.1	1	2	6
quick	0.06	0.5	1	3

Vetor quasi-ordenado

alg	100	500	1K	2.5K
bol	0.4	11	47	294
bol*	0.06	0.1	0.3	0.6
ins	0.01	0.1	0.2	0.5
sel	0.4	10	40	255
shell	0.1	0.5	1	3
heap	0.1	1	2	6
quick	0.06	0.3	0.8	2

☞ Para você fazer

Estude o que acontece quando os dados a ordenar não cabem todos ao mesmo tempo na memória do computador.

[1] Medidas feitas com um computador Pentium 500MHz, com Windows Millenium usando a linguagem APL plus, versão 6.0