Notas de aula 23/05

ORDENAÇÃO

Ordenar é rearranjar um conjunto de itens em ordem ascendente ou descendente. Seu principal objetivo é facilitar a recuperação posterior dos itens do conjunto ordenado. Ex.: Uma lista telefônica: se a lista não fosse ordenada em ordem alfabética, ficaria bem mais difícil encontrar o contato desejado.

Os métodos de ordenação podem ser classificados em dois grupos: métodos simples (ou elementares) e métodos eficientes:

- Os métodos simples são adequados para arquivos pequenos, pois conseguem realizar a ordenação a um custo $O(n^2)$.
- Os métodos eficientes podem ser aplicados a arquivos maiores, pois conseguem realizar a ordenação a um custo O(n log n).

  Iniciaremos os estudos de métodos de ordenação com os métodos simples.

Métodos simples / elementares

Esses métodos produzem programas pequenos e fáceis de entender, que ilustram com simplicidade os princípios de ordenação. Além disso, existe um grande número de situações onde é melhor usar métodos simples em vez de métodos eficientes. Pois, pesar de métodos mais sofitisticados conseguirem ordenar um conjunto de dados usando menos comparações, estas comparações são mais complexas nos detalhes, fazendo com que métodos simples sejam mais eficientes para pequenas entradas.

Alguns dos métodos simples mais utilizados são: ordenação por seleção (*selection sort*), ordenação por inserção (*insertion sort*) e bolha (*bubble sort*)

Ordenação por seleção

É um dos mais simples. Funciona da seguinte maneira: imagine um vetor *v* de tamanho *N*:

1. O menor item de *v[0, …, N - 1]* é selecionado e troca de lugar com o 1º item do vetor.
2. O menor item de *v[1, …, N - 1]* é selecionado e troca de lugar com o 2º item do vetor.
3. O menor item de *v[2, …, N - 1]* é selecionado e troca de lugar com o 3º item do vetor, e assim por diante até restar o último item do vetor.

Pseudocódigo:

```
ORDENAÇÃO - SELEÇÃO:
para i ← 0 até N - 1
        min ← i
        para j ← i até N
                se vetor[j] < vetor[min], então
                        min ← j
        troca(vetor, min, i)
```

O link abaixo traz uma visualização de como este método funciona (clique no botão Selection Sort).

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

É possível ver que o algoritmo de ordenação por seleção contém 2 laços. O primeiro laço começa em i = 0 e vai até N - 1. Já o laço mais interno começa em i e vai até N. Isso significa que:

- Para a 1ª iteração do laço mais externo, as instruções do laço mais interno se repetem *n* vezes.
- Para a 2ª iteração do laço mais externo, as instruções do laço mais interno se repetem *n - 1* vezes.
- Para a 3ª iteração do laço mais externo, as instruções do laço mais interno se repetem *n - 2* vezes.
- …
- Para a última iteração do laço mais externo, as instruções do laço mais interno são executadas 1 vez.

Portanto o custo em relação ao tempo de execução pode ser definido por:

$T(n) = n + (n - 1) + (n - 2) + \dots + 3 + 2 + 1$

Sendo assim, é possível dizer que T(n) + T(n) é igual a:

$$
\begin{array}{llllllll}
 & n + & (n - 1) + & (n - 2) + & \dots + & 3 + & 2 + & 1 \\
+ & & & & & & & \\
 & n + & (n - 1) + & (n - 2) + & \dots + & 3 + & 2 + & 1
\end{array}
$$

Que pode ser reescrito como:

$$
\begin{array}{llllllll}
 & n + & (n - 1) + & (n - 2) + & \dots + & 3 + & 2 + & 1 \\
+ & & & & & & & \\
 & 1 + & 2 + & 3 + & \dots + & (n - 2) + & (n - 1) + & n \\
\hline
 & (n + 1) + & (n + 1) + & (n + 1) + & \dots + & (n + 1) + & (n + 1) + & (n + 1)
\end{array}
$$

Logo:

$T(n) + T(n) = n(n + 1)$
$2T(n) = n(n + 1)$
$T(n) = n(n + 1) / 2$
$T(n) = (n^2 + n) / 2$

Isso quer dizer que o algoritmo terá uma complexidade de tempo $O(n^2)$. Essa é complexidade é válida tanto no pior caso (quando o vetor estiver em ordem decrescente), quando no melhor caso (quando o vetor já estiver ordenado). Isso porque o algoritmo não reconhece essas situações e executada o mesmo número de operações independente do caso. Por isso temos que:

Pior caso - Seleção = $O(n^2)$
Melhor caso - Seleção = $O(n^2)$

Ordenação por inserção

O vetor é percorrido em ordem. Cada item do vetor é selecionado e é movido para sua posição correta.

Pseudocódigo:

```
ORDENAÇÃO - INSERÇÃO:
para i ← 1 até N
    atual ← vetor[i]
    j ← i - 1
    enquanto j >= 0 e vetor[j] > atual
        desloca_direita(vetor, j) // para liberar espaço para o atual
        j ← j - 1
    vetor[j + 1] = atual
```

O link abaixo traz uma visualização de como este método funciona (clique no botão Insertion Sort).

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Considere um vetor em ordem decrescente: [5 | 4 | 3 | 2 | 1]
Quando i = 1, para que o 4 entre no lugar correto, o laço interno precisa:

1. deslocar o 5;

[4 | 5 | 3 | 2 | 1]
Quando i = 2, para que o 3 entre no lugar correto, o laço interno precisa:

1. deslocar o 5;
2. deslocar o 4;

[3 | 4 | 5 | 2 | 1]
Quando i = 3, para que o 2 entre no lugar correto, o laço interno precisa:

1. deslocar o 5;
2. deslocar o 4;
3. deslocar o 3;

[2 | 3 | 4 | 5 | 1]
Quando i = 4, para que o 1 entre no lugar correto, o laço interno precisa:

1. deslocar o 5;
2. deslocar o 5;
3. deslocar o 3;
4. deslocar o 2;

[1 | 2 | 3 | 4 | 5]

Ou seja o número de trocas é: 1 + 2 + 3 + 4, que pode ser generalizado para:
$T(n) = 1 + 2 + 3 + \ldots + (n - 2) + (n - 1) + n$

Que como foi visto no exemplo anterior, leva a uma complexidade $O(n^2)$.
Considere agora um vetor em ordem crescente: [1 | 2 | 3 | 4 | 5]
Quando i = 1, o item atual é menor que o item na posição j, o laço interno não executa;
Quando i = 2, o item atual é menor que o item na posição j, o laço interno não executa;
Quando i = 3, o item atual é menor que o item na posição j, o laço interno não executa;
Quando i = 4, o item atual é menor que o item na posição j, o laço interno não executa;

Ou seja, quando o vetor está ordenado, apenas o laço externo é executado. O laço externo vai de 1 até N, ou seja, N comparações serão feitas, levando a uma complexidade O(n).

Podemos dizer então, que quando o vetor está em ordem decrescente, temos o pior caso do algoritmo, pois ele precisa fazer até n^2 comparações. Já quando o vetor está em ordem crescente, o algoritmo reduz o número de comparações para n, constituindo o melhor caso para o algoritmo. Sendo assim:

Pior caso - Inserção = $O(n^2)$
Melhor caso - Inserção = O(n)

Bolha

O vetor vai sendo percorrido em ordem, comparando pares de elementos. Sempre que o par de elementos estiver desordenado, os elementos trocam de lugar. Ao fim de cada iteração, o último item do vetor já está na posição certa.

Pseudocódigo:

```
ORDENAÇÃO - BOLHA:
para i ← N - 1 até 1
    para j ← 0 até i
        se vetor[j] > vetor[j + 1], então
            troca(vetor, j, j + 1) // vai empurrando o maior para o final
```

O laço mais interno começa em N - 1 e se repete até que i seja >= 1. O laço mais interno vai de 0 até i. Isso significa que a cada iterção do laço mais externo, o laço mais interno precisa ser executado uma vez a menos. Isso quer dizer que:

- Para a 1ª iteração do laço mais externo, as instruções do laço mais interno se repetem *n* vezes.
- Para a 2ª iteração do laço mais externo, as instruções do laço mais interno se repetem *n - 1* vezes..
- …
- Para a última iteração do laço mais externo, as instruções do laço mais interno são executadas 1 vez.

Portanto o custo em relação ao tempo de execução pode ser definido por:

$T(n) = n + (n - 1) + (n - 2) + \ldots + 3 + 2 + 1$

Que já foi verificado que leva a um custo $O(n^2)$. Este custo é o mesmo quando o vetor está em ordem crescente ou decrescente, então a complexidade será $O(n^2)$ tanto no melhor quanto no pior caso.

Entretanto, é possível melhorar o algoritmo com o uso de uma flag:

```
ORDENAÇÃO - BOLHA:
flag ← 0
para i ← N - 1 até 1
    Se flag == 0, então
        para j ← 0 até i
            se vetor[j] > vetor[j + 1], então
                troca(vetor, j, j + 1) // vai jogando o maior para o final
```

flag ← 0
Se não, então
flag ← 1

Considere um vetor em ordem decrescente: [5 | 4 | 3 | 2 | 1]. A flag inicia com 0, portanto o laço mais interno é executado. Neste laço, vetor[j] sempre é maior que vetor[j + 1], e a flag sempre vai receber o valor 0. Ao sair da primeira iteração do laço mais interno, a flag está valorizada com 0, fazendo o laço executar novamente e assim por diante. Neste caso, a complexidade irá permanecer $O(n^2)$.

Considere agora um vetor em ordem crescente: [1 | 2 | 3 | 4 | 5]. A flag inicia com 0, portanto o laço mais interno é executado. Neste laço, vetor[j] sempre é menor que vetor [j + 1], e a flag sempre vai receber o valor 1. Ao sair da primeira iteração do laço mais interno, a flag está valorizada com 1, não entrando mais no laço interno. Neste caso o algoritmo vai fazer N comparações para o laço mais externo + N comparações da única execução do laço mais interno, ou seja 2N comparações. Portanto, com o uso da flag, é possível diminuir o custo para O(n) quando o vetor está em ordem crescente.

Desta maneira, um vetor em ordem decrescente constitui o pior caso do algoritmo, enquanto um vetor em ordem crescente constitui o melhor caso do algoritmo. Podemos dizer então que:

Pior caso - Bolha = $O(n^2)$
Melhor caso - Bolha = O(n)

O link abaixo traz uma visualização de como este método funciona (clique no botão Bubble Sort).

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Links

Complexidades dos principais algoritmos: http://www.bigocheatsheet.com/

Referências

Ziviani, N. “Projeto de algoritmos: com implementações em Pascal e C”. 3ª ed, São Paulo, Cengage Learning, 2017.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. “Algoritmos: teoria e prática”. 6ª ed. Rio de Janeiro, Elsevier, 2002.