Ordenação por comparação[1][2]

Algoritmos de ordenação têm sido extensivamente estudados na computação teórica devido à frequência com que se fazem necessários em aplicações cotidianas. Dentre as diferentes abordagens existentes, os algoritmos de ordenação por comparação constituem uma importante classe, quer por sua natureza didática, quer por sua aplicabilidade prática. Dentro desta classe, destacam-se duas famílias de algoritmos de ordenação que apresentam funcionamento e eficiência bastante relacionados entre si.

A primeira família de algoritmos baseia-se na noção de ordenação incremental. Sendo baseados em princípios bastante simples, a eficiência destes algoritmos compromete bastante sua aplicação prática, sendo utilizados mais comumente em situações didáticas. Fazem parte deste grupo o *bubble sort*, a ordenação por seleção e a ordenação por inserção. Em termos de eficiência, uma exceção nesta classe de algoritmos é o *heap sort* que, apesar de ser conceitualmente idêntico à ordenação por seleção, consegue apresentar uma eficiência muito melhor devido à estrutura de dados usada internamente pelo algoritmo (uma *heap*). A segunda família de algoritmos é composta por métodos baseados na estratégia dividir-para-conquistar, que os proporciona uma eficiência significativamente maior. No entanto, alguns destes algoritmos requerem o uso de estruturas de memória adicionais, ou apresentam complexidade de pior caso idêntica à dos algoritmos de ordenação incremental devido à casos especiais. Fazem parte deste grupo o *merge sort* e o *quick sort*.

Neste roteiro, serão detalhados os algoritmos de ordenação por seleção e por inserção.

1 Ordenação por seleção (*selection sort*)

A ordenação por seleção é um exemplo de algoritmo de ordenação por comparação bastante ineficiente, baseado na idéia de ordenar um elemento a cada iteração assim como todos os algoritmos desta família.

Lógica do algoritmo

A ordenação por seleção trabalha o vetor de entrada decrementalmente, considerando que a cada iteração o elemento de valor mínimo (ou máximo, analogamente) será movido para sua posição correta. A ordenação por seleção executa uma única troca por iteração: enquanto o laço mais interno do algoritmo identifica o elemento máximo do subvetor atual, o laço externo o reposiciona para a posição correta.

Pseudocódigo

O pseudocódigo do algoritmo de ordenação por seleção pode ser visto abaixo. O laço externo (linha 1) controla o tamanho do subvetor a ser analisado a cada iteração. O laço interno (linhas 2–7) encontra o elemento de menor valor no subvetor em questão. Por fim, o elemento mínimo é reposicionado (linha 8).

[1] Roteiro de estudo fornecido na disciplina de Estruturas de Dados Básicas 1 (EDB1) da Universidade Federal do Rio Grande do Norte (UFRN).

[2] Autor: Leonardo Bezerra.

Algorithm 1 Algoritmo de ordenação por seleção

Input: vetor v, tamanho n

```
1: for (i := 0; i < n; i += 1) do
2:     min := i
3:     for (j := i+1; j < n; j += 1) do
4:         if v[j] < v[min] then
5:             min := j
6:         end if
7:     end for
8:     trocar(v[i],v[min])
9: end for
```

Exemplo de execução

Tomemos como exemplo o vetor de entrada 4 – 5 – 1 – 3 – 2. A primeira iteração do algoritmo considera o vetor em seu tamanho original, assegurando que o elemento mínimo será deslocado para a posição final:

$\mathbf{4} - 5 - 1 - 3 - 2$ – Mínimo: 4
$4 - \mathbf{5} - 1 - 3 - 2$ – Mínimo: 4
$4 - 5 - \mathbf{1} - 3 - 2$ – Mínimo: 1
$4 - 5 - 1 - \mathbf{3} - 2$ – Mínimo: 1
$4 - 5 - 1 - 3 - \mathbf{2}$ – Mínimo: 1
$\mathbf{4} - 5 - \mathbf{1} - 3 - 2$ – Troca a ser efetuada
$\mathbf{1} - 5 - \mathbf{4} - 3 - 2$ – Troca efetuada

Uma vez que o elemento 1 já está na sua posição final, a próxima iteração do algoritmo considera o subvetor de tamanho 4:

$1 - \mathbf{5} - 4 - 3 - 2$ – Mínimo: 5
$1 - 5 - \mathbf{4} - 3 - 2$ – Mínimo: 4
$1 - 5 - 4 - \mathbf{3} - 2$ – Mínimo: 3
$1 - 5 - 4 - 3 - \mathbf{2}$ – Mínimo: 2
$1 - \mathbf{5} - 1 - 3 - \mathbf{2}$ – Troca a ser efetuada
$1 - \mathbf{2} - 4 - 3 - \mathbf{5}$ – Troca efetuada

Para este exemplo em específico, o vetor estará ordenado ao final da próxima iteração:

$1 - 2 - \mathbf{4} - 3 - 5$ – Mínimo: 4
$1 - 2 - 4 - \mathbf{3} - 5$ – Mínimo: 3
$1 - 2 - 4 - 3 - \mathbf{5}$ – Mínimo: 3
$1 - 2 - \mathbf{4} - \mathbf{3} - 5$ – Troca a ser efetuada
$1 - 2 - \mathbf{3} - \mathbf{4} - 5$ – Troca efetuada

No entanto, uma vez que este algoritmo não possui nenhum mecanismo para detectar que o algoritmo já está ordenado, sua execução deverá prosseguir até que o critério de parada do laço externo seja atingido:

$1 - 2 - 3 - \mathbf{4} - 5$ – Mínimo: 4
$1 - 2 - 3 - 4 - \mathbf{5}$ – Mínimo: 4
$1 - 2 - 3 - \mathbf{4} - 5$ – Troca a ser efetuada
$1 - 2 - 3 - \mathbf{4} - 5$ – Troca efetuada

Análise da complexidade

O algoritmo de ordenação por seleção é ineficiente para todos os casos (melhor, pior e médio). Sua complexidade assintótica é $O(n^2)$, o que torna seu uso proibitivo quando se trabalha com quantidades razoáveis de dados. No entanto, é possível otimizar este algoritmo consideravelmente fazendo uso de uma *heap*.

Extra! – pesquise e implemente uma heap para compreender o heapsort. A implementação correta do heapsort renderá 1 ponto extra nesta unidade.

2 Ordenação por inserção (*insertion sort*)

A ordenação por inserção pertence à mesma família da ordenação por seleção uma vez que ordena o vetor de forma incremental, mas apresenta eficiência muito maior na prática que os demais algoritmos desta família. Ainda que não seja recomendado para situações onde a quantidade de dados é razoável, a ordenação por inserção é por vezes utilizada como uma rotina auxiliar por algoritmos de ordenação mais eficientes.

Lógica do algoritmo

Diferentemente do método da seleção, a ordenação por inserção trabalha com subvetores ordenados incrementalmente. Assim, este algoritmo nunca considera a totalidade do vetor de entrada em iterações iniciais, o que o torna significativamente mais eficiente que os demais. Em uma iteração i do laço externo, têm-se que os elementos atrás da posição i já estão ordenados, devendo o laço interno encontrar a posição correta do elemento atual no subvetor ordenado.

Pseudocódigo

O pseudocódigo do algoritmo de ordenação por inserção pode ser visto abaixo. O laço externo (linha 1) seleciona o elemento a ser reposicionado na iteração atual, considerando que o subvetor atrás dessa posição está corretamente ordenado. O laço interno (linhas 2–6) percorre o subvetor em questão para encontrar a posição correta do elemento atual no novo subvetor.

Algorithm 2 Algoritmo de ordenação por inserção

Input: vetor v, tamanho n

```
1: for (i := 1; i < n; i += 1) do
2:     for (j := i; j > 0; j -= 1) do
3:         if (v[j] ≥ v[j-1]) then
4:             break
5:         end if
6:     end for
7:     mover(i,j)
8: end for
```

Exemplo de execução

Tomemos como exemplo o vetor de entrada 4 – 5 – 1 – 3 – 2. O algoritmo considera que o subvetor unitário já está ordenado e, portanto, a primeira iteração tenta encontrar a posição adequada do segundo elemento do vetor original neste subvetor:

$$\mathbf{4} - \mathbf{5} - 1 - 3 - 2 \qquad \text{– Posição já adequada: nenhum reposicionamento será efetuado}$$

Na iteração seguinte ($i = 2$), o algoritmo considera que o subvetor de tamanho 2 e origem 0 já está ordenado, e busca a posição correta para o elemento $v[2]$:

$$\begin{aligned}
&4 - \mathbf{5} - \mathbf{1} - 3 - 2 && \text{– Posição: 2}\\
&\mathbf{4} - 5 - \mathbf{1} - 3 - 2 && \text{– Posição: 1}\\
&4 - 5 - \mathbf{1} - 3 - 2 && \text{– Posição: 0 – limite do subvetor atingido}\\
&\mathbf{1} - \mathbf{4} - \mathbf{5} - 3 - 2 && \text{– Reposicionamento efetuado}
\end{aligned}$$

Seguindo a mesmo lógica da iteração anterior, o algoritmo agora busca a posição correta para o elemento $v[3]$:

$1 - 4 - \mathbf{5} - \mathbf{3} - 2$ – Posição: 3
$1 - \mathbf{4} - 5 - \mathbf{3} - 2$ – Posição: 2
$\mathbf{1} - 4 - 5 - \mathbf{3} - 2$ – Posição adequada: 1
$\mathbf{1} - \mathbf{3} - \mathbf{4} - \mathbf{5} - 2$ – Reposicionamento efetuado

Conforme discutido anteriormente, apenas a última iteração considera o tamanho original do algoritmo:

$1 - 3 - 4 - \mathbf{5} - \mathbf{2}$ – Posição: 4
$1 - 3 - \mathbf{4} - 5 - \mathbf{2}$ – Posição: 3
$1 - \mathbf{3} - 4 - 5 - \mathbf{2}$ – Posição: 2
$\mathbf{1} - 3 - 4 - 5 - \mathbf{2}$ – Posição adequada: 1
$\mathbf{1} - \mathbf{2} - \mathbf{3} - \mathbf{4} - \mathbf{5}$ – Reposicionamento efetuado

Análise da complexidade

Conforme mencionado anteriormente, o algoritmo de ordenação por inserção é mais eficiente que os outros métodos vistos até aqui. A complexidade assintótica de melhor caso desse algoritmo é $O(n)$, enquanto sua complexidade assintótica de pior caso é $O(n^2)$. No entanto, apesar de apresentar complexidade de caso médio $O(n^2)$, este algoritmo é particularmente rápido para ordenar quantidades pequenas de dados. De fato, a eficiência deste algoritmo é tanta que outros algoritmos como o *quick sort* o utilizam como uma rotina auxiliar, como se pode verificar na implementação oficial deste algoritmo no padrão ANSI/ISO C.