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 *merge sort* e *quick sort*.

1 *Merge sort*

O algoritmo *merge sort* faz parte da família dos algoritmos de ordenação por comparação baseados em métodos dividir-para-conquistar. Comparado à eficiência dos algoritmos baseados em ordenação incremental, o *merge sort* é um procedimento extremamente eficaz, ainda que existam outros algoritmos dividir-para-conquistar mais eficientes que este. Nesta seção, detalharemos este algoritmo, discutindo sua eficiência em seguida.

Lógica do algoritmo

Diferentes implementações do algoritmo *merge sort* divergem quanto à forma de implementá-lo, mas o princípio por trás deste algoritmo é bastante simples: ordenar subconjuntos de dados é computacionalmente mais barato que ordenar o conjunto inteiro. Assim, algoritmos *merge sort top-down* inicialmente dividem o vetor de entrada de forma recursiva, até cair no caso base de vetores unitários. Em seguida, o *backtracking* da recursão mescla os subvetores já ordenados oriundos dos níveis inferiores da recursão (daí o nome *merge sort*).

Pseudocódigo

O pseudocódigo do algoritmo *merge sort* pode ser visto abaixo nos Algoritmos 1 e 2. O modelo geral da recursão é mostrado no Algoritmo 1, onde o caso base é tratado nas linhas 2–3. As linhas 5–6 executam o procedimento recursivo, aprofundando a recursão em ambos os lados. Por fim, a linha 7 efetua a mescla dos subvetores ordenados, com o auxílio do procedimento merge, descrito no Algoritmo 2.

[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.

Algoritmo 1 mergeSort (vetor v, índices $\langle inicio,\ fim \rangle$)

```
1: n ← fim - inicio
2: if n < 2 then
3:    return
4: else
5:    mergeSort(v, 0, inicio + ⌊n/2⌋)
6:    mergeSort(v, inicio + ⌊n/2⌋, n)
7:    merge(v, inicio, fim)
8: end if
```

No Algoritmo 1, as linhas 1–2 definem os índices usados para percorrer os subvetores ordenados e preencher o vetor auxiliar onde os dados ordenados serão armazenados. O laço apresentado nas linhas 5–11 é executado enquanto os índices não atingem o fim dos subvetores, armazenando no vetor auxiliar os elementos em ordem. Mais precisamente, o teste efetuado na linha 6 define se o próximo elemento a ser armazenado no vetor auxiliar deverá vir do subvetor esquerdo ou direito.

Por sua vez, o laço apresentado nas linhas 14–20 é executado quando um dos índices atingiu o fim de um subvetor, mas ainda há elementos restando no outro subvetor. Neste caso, não há necessidade de se comparar elementos, bastando copiar todos os elementos restantes no subvetor que ainda não foi mesclado por inteiro. Note que o teste efetuado na linha 15 é responsável por identificar qual subvetor ainda apresenta elementos a serem adicionados ao vetor auxiliar.

Por fim, o laço das linhas 23–26 copia os dados ordenados armazenados no vetor auxiliar para o vetor original.

Algoritmo 2 merge(vetor v, índices $\langle inicio,\ fim \rangle$)

```
1: n ← fim - inicio, meio ← inicio + ⌊n/2⌋
2: i ← inicio, j ← meio, k ← 0
3:
4: // percorre os subvetores enquanto nenhum houver sido esgotado
5: while i < meio and j < fim do
6:    if v[i] < v[k] then
7:       aux[k] ← v[i]; i ← i + 1
8:    else
9:       aux[k] ← v[j]; j ← j + 1
10:   end if
11: end while
12:
13: // percorre o subvetor que ainda não foi esgotado
14: while i < meio or j < fim do
15:   if i < meio then
16:      aux[k] ← v[i]; i ← i + 1
17:   else
18:      aux[k] ← v[j]; j ← j + 1
19:   end if
20: end while
21:
22: // copia o vetor ordenado para o vetor original
23: i ← inicio; k ← 0
24: while i < fim do
25:   v[i] ← aux[k]; i ← i + 1; k ← k + 1
26: end while
```

Exemplo de execução

Tomemos como exemplo o vetor de entrada 4 – 5 – 1 – 3 – 2. A recursão é efetuada como descrita a seguir:

$4-5-1-3-2$	– Descendo na recursão para o subvetor v[0:2]
$4-5 \quad 1-3-2$	– Descendo na recursão para o subvetor v[0:1]
4 **5** $1-3-2$	– Caso base atingido para v[0] e v[1]
4 – 5 $1-3-2$	– *Backtracking*: mesclando os subvetores v[0] e v[1]
4 – 5 $1-3-2$	– *Backtracking*: descendo na recursão para o subvetor v[2:5]
4 – 5 **1** $3-2$	– Caso base atingido para v[2]
4 – 5 **1** 3 2	– *Backtracking*: descendo na recursão para o subvetor v[3:5]
4 – 5 **1** 3 2	– Caso base atingido para v[3] e v[4]
4 – 5 **1** **2 – 3**	– *Backtracking*: mesclando os subvetores v[3] e v[4]
4 – 5 **1 – 2 – 3**	– *Backtracking*: mesclando os subvetores v[2] e v[3:5]
1 – 2 – 3 – 4 – 5	– *Backtracking*: mesclando os subvetores v[0:2] e v[2:5]

Análise da complexidade

Conforme discutido anteriormente, algoritmos dividir-para-conquistar apresentam complexidade assintótica melhor que algoritmos baseados em ordenação incremental. As complexidades de melhor, pior e caso médio para o *merge sort* são todas $\Theta(n \log n)$. No entanto, a complexidade de melhor caso deste algoritmo pode ser melhorada quando outras variantes do mesmo são usadas, como o *merge sort* natural.

2 *Quick sort*

O algoritmo *quick sort* é provavelmente o algoritmo de ordenação mais utilizado em aplicações cotidianas, uma vez que sua eficiência na prática supera dos demais algoritmos de ordenação baseados na estratégia dividir-para-conquistar. Em particular, diversos mecanismos têm sido investigados para aumentar ainda mais a eficiência deste algoritmo. Nesta seção, apresentamos o algoritmo tradicional e detalhamos alguns destes mecanismos ao analisar a complexidade deste método de ordenação.

Lógica do algoritmo

Ainda que também seja baseado na idéia de dividir-para-conquistar, o *quick sort* segue uma lógica um pouco diferente da usada pelo *merge sort*. No *quick sort*, cada nível de recursão têm por objetivo particionar os elementos do subvetor atual em dois subconjuntos disjuntos. Assim, dado um determinado nível da recursão, seleciona-se um elemento para atuar como pivô e promove-se uma série de trocas para assegurar que todos os elementos à esquerda do pivô sejam menores que o mesmo. Uma vez que o subvetor atual obedeça este padrão, a recursão prossegue até que, atingindo todos os casos base, o vetor original estará ordenado.

Pseudocódigo

O pseudocódigo do algoritmo *quick sort* pode ser visto nos Algoritmos 3 e 4. O modelo geral da recursão é apresentado no Algoritmo 3, onde as linhas 1–2 cuidam do caso base. O caso recursivo é tratado nas linhas 3–7, onde a linha 4 efetua o particionamento do vetor atual e as linhas 5–6 aprofundam a recursão para cada subvetor. Mais precisamente, o procedimento partition assegura que os elementos nas posições *inicio* até p sejam menores que o pivô (que estará na posição p), enquanto os elementos nas posições p até *fim* sejam maiores que o pivô.

Algorithm 3 quickSort (vetor v, índices $\langle inicio, fim \rangle$)

```
1: if fim - inicio < 2 then
2:     return
3: else
4:     p ← partition(v, inicio, fim)
5:     quickSort(v, inicio, p)
6:     quickSort(v, p+1, fim)
7: end if
```

O Algoritmo 4 descreve o procedimento partition. Para compreender essa operação, os índices-limite i e j devem ser considerados fronteiras das partições que se pretende criar nos extremos do subvetor atual. Os laços

das linhas 3–5 e 6–8 incrementam/decrementam esses índices até que sejam encontrados elementos que não pertençam à respectiva partição. Neste ponto, os elementos presentes nessas posições são trocados (linhas 9–12). Mais geralmente, o laço das linhas 2–13 continua até que os índices se alcancem (particionamento concluído). Ao final deste procedimento (linhas 14–15), o elemento escolhido como pivô é posicionado entre as duas partições e sua posição é retornada.

Algoritmo 4 partition(vetor v, índices $\langle inicio, fim \rangle$)

```
 1: aux ← inicio; i ← inicio + 1; j ← fim - 1
 2: while i ≤ j do
 3:    while i ≤ j and v[i] ≤ v[aux] do
 4:       i ← i + 1
 5:    end while
 6:    while i ≤ j and v[j] > v[aux] do
 7:       j ← j - 1
 8:    end while
 9:    if v[i] > v[j] then
10:       swap(v[i], v[j])
11:       i ← i + 1, j ← j - 1
12:    end if
13: end while
14: swap(v[aux], v[j])
15: return j
```

Exemplo de execução

Tomemos como exemplo o vetor de entrada 4 – 5 – 1 – 3 – 2. Considerando que o primeiro elemento do subvetor atual será usado como pivô, a recursão é efetuada como descrita a seguir:

4 – 5 – 1 – 3 – 2	– Selecionado pivô: 4 (v[0])
4 **2** – 1 – 3 – **5**	– Troca realizada (5 e 2)
3 – 2 – 1 **4** 5	– Reposicionado pivô
3 – 2 – 1 **4** 5	– Chamando recursão para subvetor v[0:2]
3 2 – 1 **4** 5	– Selecionado pivô: 3 (v[0])
3 2 – 1 **4** 5	– Nenhuma troca necessária
1 – 2 **3** **4** 5	– Reposicionando pivô
1 – 2 **3** **4** 5	– Chamando recursão para subvetor v[0:1]
1 – 2 **3** **4** 5	– Selecionando pivô: 1 (v[0])
1 **2** **3** **4** 5	– Nada a fazer
1 **2** **3** **4** 5	– *Backtracking*: chamando recursão para subvetor [4:5]
1 **2** **3** **4** **5**	– Nada a fazer

Análise da complexidade

Assim como o algoritmo *merge sort*, a complexidade assintótica de melhor caso e de caso médio do *quick sort* é $O(n \log n)$. No entanto, a complexidade de pior caso deste algoritmo é $O(n^2)$, assim como a dos algoritmos de ordenação incremental. Na prática, contudo, o *quick sort* é o algoritmo de ordenação por comparação mais rápido graças a diferentes técnicas usadas para melhorar seu desempenho. Dentre as principais, destacam-se (i) a seleção aleatória de pivôs e (ii) o uso de algoritmos como a ordenação por inserção quando os subvetores atingem um tamanho mínimo. Uma série de outras abordagens pode ser encontradas em Sedgewick (1998).