Capítulo 4

Algoritmos de ordenação sofisticados

Com a evolução dos estudos relacionados a algoritmos, muitas técnicas vão se complementando e algumas novas podem surgir. O objetivo é melhorar continuamente o desempenho nas soluções de problemas, tanto nos conhecidos como nos desconhecidos. Em problemas de ordenação, pode-se encontrar uma grande fartura de novas técnicas e ainda obter um bom ambiente para compará-las. Algoritmos de ordenação são explorados nessas situações quando se tem um conhecimento claro do problema a ser resolvido. Isso torna a classificação das técnicas empregadas nos algoritmos o trabalho mais realizado entre os pesquisadores.

Uma das técnicas mais utilizadas para solucionar problemas complexos e que exigem um grande número de comparações é a técnica de dividir e conquistar, que consiste em pegar um problema relativamente grande e quebrá-lo em subproblemas menores até que sua resolução seja praticamente uma única operação computacional (SEDGEWICK;

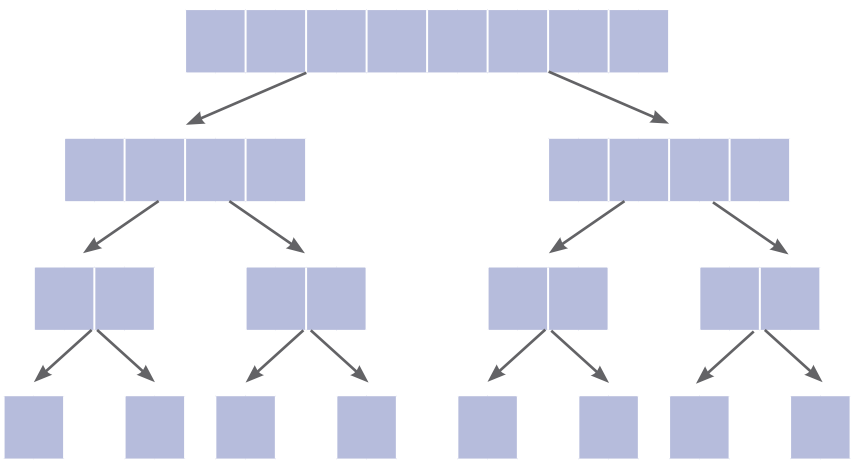
FLAJOLET, 2013). Os subproblemas obtidos através da divisão do problema original devem estar intimamente relacionados a ele (CORMEN *et al.*, 2002). Diversos algoritmos de ordenação utilizam essa técnica, e por isso são considerados algoritmos de ordenação sofisticados.

A técnica de dividir e conquistar é definida com três passos (CORMEN *et al.*, 2002; GOODRICH; TAMASSIA, 2013):

- **Dividir:** o problema é dividido em um determinado número de subproblemas.
- **Conquistar:** os subproblemas são resolvidos recursivamente.
- **Combinar:** junta-se as soluções dos subproblemas combinando-os e obtendo ao final a solução do problema original.

Dividir é o passo mais simples para a maioria dos problemas que serão resolvidos com essa técnica. A figura 1 ilustra o processo de divisão de um vetor com oito posições.

Figura 1 – Processo de divisão de um problema com um vetor de 8 posições, até que ele seja quebrado em 8 subproblemas



Note que o processo de divisão é naturalmente obtido através de um algoritmo recursivo a partir do problema original, que é dividido até chegar a seu caso básico. Após o processo de divisão, o caso básico é concluído, obtendo assim o resultado de cada um dos subproblemas definidos no processo da divisão. Com os resultados dos casos básicos calculados, o processo de conquista também é finalizado. Nesse momento, inicia-se o processo de combinação, que, usando como exemplo o somatório das raízes quadradas, será a soma dos resultados individuais de cada elemento, no momento de retorno de cada chamada recursiva do problema.

A técnica de divisão e conquista é por si mesma recursiva, como observado no exemplo da figura 1. Sendo assim, descobrir de forma precisa seu desempenho é uma tarefa não trivial. Um dos motivos é porque nem sempre a divisão dos problemas será exata, já que em muitos casos os conjuntos classificados podem conter uma quantidade ímpar de elementos. Isso gera uma divisão não muito precisa para se contabilizar de maneira simétrica ambas as partes divididas (SEDGEWICK; FLAJOLET, 2013).

Distinguir a diferença de desempenho quando o total de elementos é ímpar ou par não é uma tarefa precisa quando o conjunto de dados contém milhares de elementos. Porém, em conjuntos menores, a diferença no desempenho é notável. Essa diferença pode ser expressiva se a divisão do subproblema for maior que 2 a cada chamada recursiva. Isso ocorre porque existe uma variação na quantidade de subproblemas chamados de maneira recursiva. Ao juntá-los para a solução do problema original, o custo pode ser alto e ainda pode ocorrer sobreposição de subproblemas, gerando resultados ambíguos (SEDGEWICK; FLAJOLET, 2013).

Contudo, o uso da técnica de divisão e conquista é muito ampla em diversas aplicações. Ela é uma técnica elegante e permite simplificar muito o código da solução, já que utiliza a recursão para isso. Ao longo deste capítulo, serão explorados dois algoritmos de ordenação. Estes são definidos de maneira a aplicar a técnica de dividir e conquistar. São eles: o merge sort e o quicksort.

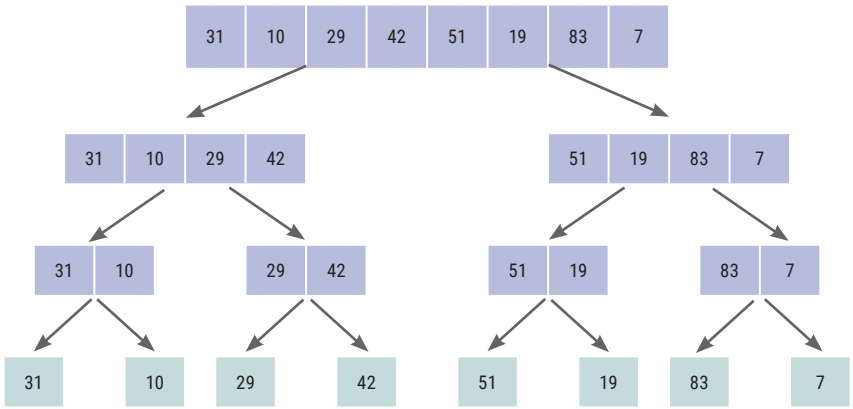
1 Merge sort

Dentre os algoritmos de ordenação que aplicam a técnica de divisão e conquista, o que possui o processo mais simples e eficiente sem dúvidas é o merge sort. Ele é um algoritmo recursivo que se divide em subvetores até obter sua ordenação. Como o merge sort é um processo recursivo, é necessário estabelecer as regras para os casos básico e recursivo (GOODRICH; TAMASSIA, 2013). Veja as regras:

- **Caso básico:** se o vetor ou subvetor possuir apenas um único elemento, ou, se ele for vazio, retornará o próprio vetor, e assim ele será considerado ordenado, uma vez que não existe uma falta de ordem quando há apenas um elemento.
- **Caso recursivo:** se o vetor ou subvetor possuir dois ou mais elementos, este deve ser dividido em dois subconjuntos a partir de sua posição central.

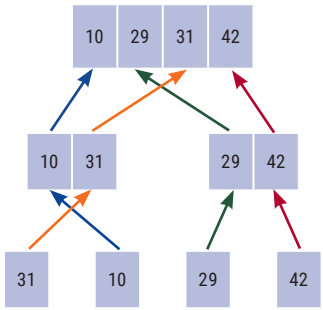
Ao analisar o processo recursivo anterior, pode-se perceber que este é o processo de divisão da técnica. Veja o vetor apresentado na divisão do merge sort, na figura 2.

Figura 2 – Processo de divisão do merge sort (as chamadas recursivas estão representadas pela cor azul, e os casos básicos, pela cor verde, como última etapa da divisão)



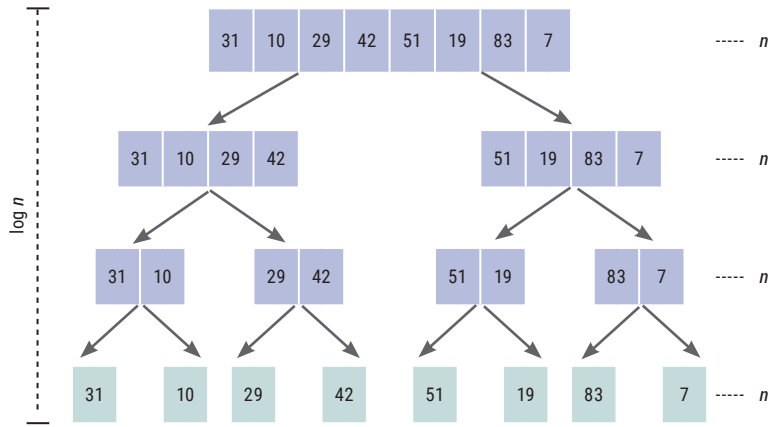
Após atingir o caso básico, o processo de conquista acontece e agora inicia-se o processo de combinação. Nesse momento, ocorre o retorno das funções recursivas para formar o vetor ordenado ao final. Cada subconjunto será concatenado de maneira a formar o subconjunto pai até que o problema original seja resolvido e o resultado final da ordenação seja obtido. A figura 3 apresenta dois níveis do algoritmo de retorno da função.

Figura 3 – Processo de retorno da chamada recursiva do merge sort, em que cada movimentação é representada por uma seta colorida, para cada nível do processo de ordenação (apenas dois níveis de ordenação foram apresentados para facilitar o entendimento)



Ao observarmos o processo do algoritmo merge sort, podemos imaginá-lo como uma árvore binária, a qual chamamos de árvore merge sort. Cada nó dessa árvore representa uma chamada recursiva. Os nós folhas, que seriam o último nível da árvore, são considerados os casos básicos do processo recursivo, como o nível com os nós de cor verde apresentado na figura 2, por exemplo. Neles, os subconjuntos estão totalmente ordenados e o retorno das chamadas recursivas inicia a ordenação durante a navegação aos nós pais, até que o vetor esteja totalmente ordenado, que é quando o nó raiz é atingido (GOODRICH; TAMASSIA, 2013). Com o conhecimento sobre a divisão em forma de árvore, a análise da eficiência desse algoritmo fica mais clara, acompanhe na figura 4.

Figura 4 – Análise do processo de execução do algoritmo merge sort, com base no exemplo da figura 1



Na figura 4, é utilizado o mesmo exemplo da figura 1, no entanto, a ênfase é exatamente na quantidade de operações que são realizadas pelo algoritmo. Agora, a altura da árvore obtida no processo recursivo executa o número de log *n* operações (log na base 2, normalmente omitido nas análises de algoritmos). Cada linha executa *n* operações. Ao combinar as duas funções, obtemos um total de *n* log *n* operações. Sendo assim, o merge sort pode ser considerado mais rápido que os algoritmos mais simples, como bubble sort, insertion sort e selection sort.

Nesse momento, após a compreensão do algoritmo, vamos realizar a sua implementação utilizando a linguagem de programação Java:

```
1.  import java.util.Arrays;
2.
3.  public class merge sort {
4.
5.     public static void sort(int X[], int inicio, int
fim) {
6.         if (inicio < fim) {
7.             int meio = (inicio + fim) / 2;
8.             sort(X, inicio, meio);
9.             sort(X, meio + 1, fim);
```

```
10.            merge(X, inicio, meio, fim);
11.            System.out.printf("Passo recursivo: %s \n",
Arrays.toString(X));
12.        }
13.    }
14.
15.    private static void merge(int X[], int inicio, int
meio, int fim) {
16.        int i, esquerda, direita;
17.        int aux[] = new int[X.length];
18.
19.        for (i = inicio; i <= fim; i++) aux[i] = X[i];
20.
21.        esquerda = inicio;
22.        direita = meio + 1;
23.        i = inicio;
24.
25.        while (esquerda <= meio && direita <= fim) {
26.            if (aux[esquerda] <= aux[direita]) X[i++] =
aux[esquerda++];
27.            else X[i++] = aux[direita++];
28.        }
29.
30.        while (esquerda <= meio) X[i++] =
aux[esquerda++];
31.    }
32.
33.    public static void main(String[] args) {
34.            int X[] = {31, 10, 29, 42, 51, 19, 83, 7};
35.            sort(X, 0, X.length - 1);
36.            System.out.printf("Ordenado: %s \n", Arrays.
toString(X));
37.    }
38. }
```

O algoritmo é dividido em dois métodos, o método *sort* e o *merge*. No método *sort*, entre as linhas 5 e 13, ocorrem apenas as chamadas recursivas, dividindo o vetor ao meio enquanto a posição de início for menor que o fim. O método que exige mais capacidade de processamento é o *merge*. A parte mais onerosa da ordenação está entre as linhas 25 e 28,

nas quais as posições são comparadas e copiadas no local correto da ordenação.

Perceba que esse algoritmo apresenta uma implementação complexa e o uso da recursão é imprescindível para obter um melhor entendimento. Outro algoritmo em que ocorre essa situação é o quicksort, que será apresentado no próximo tópico.

2 Quicksort

Apesar de seu nome pressupor rapidez, em seu pior caso, o tempo de execução do algoritmo quicksort terá um número de operações igual à quantidade de elementos ao quadrado. Contudo, espera-se que seu tempo de execução médio seja $n \log n$, assim como o merge sort. Diferentemente deste, o processo mais custoso do quicksort em relação ao seu desempenho está na separação dos elementos para a chamada recursiva. Esse trabalho é realizado com base em um elemento de índice p, denominado de pivô. A determinação do elemento pivô pode ser realizada de maneira aleatória ou arbitrária (CORMEN *et al*., 2002).

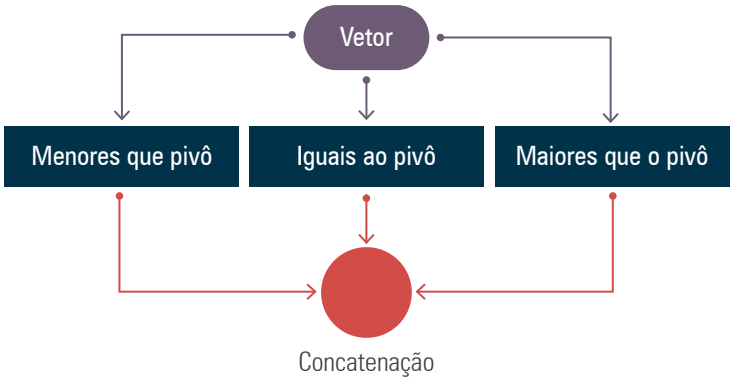
Apesar dessas considerações, o quicksort é um dos algoritmos mais empregados na tarefa de ordenação (GOODRICH; TAMASSIA, 2013). Seu desempenho é tão significativo que as soluções desenvolvidas em ambientes virtuais utilizam-no como algoritmo de ordenação (CORMEN *et al*., 2002).

O processo de divisão e conquista do quicksort ocorre a partir do particionamento do vetor em três outros (GOODRICH; TAMASSIA, 2013):

- um vetor com os elementos menores que o pivô;
- um vetor com os elementos iguais ao pivô;
- um vetor com os elementos maiores que o pivô.

Veja a ilustração do processo de partição e concatenação do vetor em ordenação pelo algoritmo quicksort, na figura 5.

Figura 5 – Processo de partição do vetor original em três outros vetores, a partir de um elemento pivô, e a concatenação, já na posição ordenada

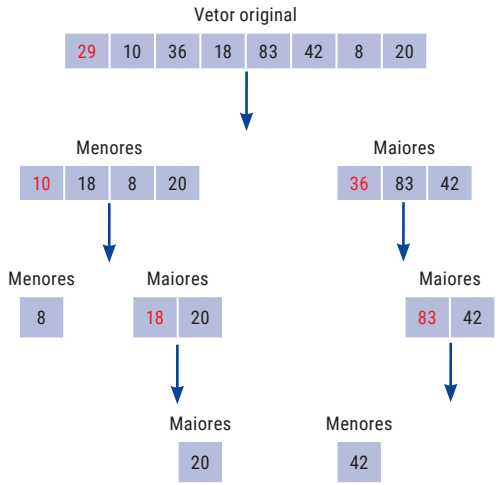


Fonte: adaptado de Goodrich e Tamassia (2013).

O pivô, como já observamos, pode ser obtido de maneira aleatória ou arbitrária, fixa, a partir das posições de início (0) ou fim (tamanho do vetor – 1). Não se recomenda utilizar uma posição ao centro do vetor, pois, em algum momento durante o processo de ordenação, essa posição pode não mais existir. Portanto, se não quiser utilizar as posições nas extremidades dos vetores, trabalhe com posições relativas (CORMEN *et al.*, 2002).

A figura 6 apresenta o exemplo de separação do vetor a partir do particionamento com base em um pivô. Para o exemplo, o pivô será considerado como o primeiro elemento do vetor, o elemento armazenado na posição 0.

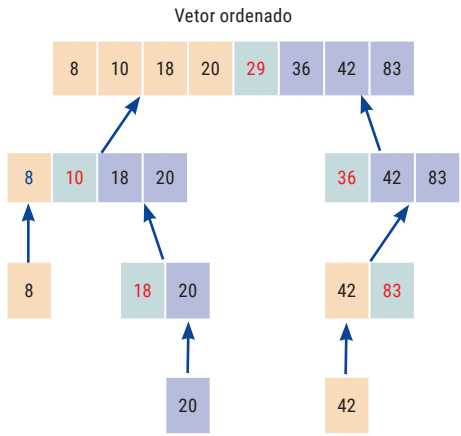
Figura 6 – Processo de particionamento adotando como pivô o primeiro elemento de cada conjunto (os pivôs estão sinalizados na cor vermelha)



O processo de partição é realizado até que os subconjuntos tenham o tamanho de um único elemento.

Após a separação, é necessário unir os subconjuntos para que retornem o vetor original de maneira ordenada. Como os vetores são separados por menores, iguais e maiores, a combinação entre eles é basicamente uma operação simples de junção já na sequência ordenada. A ordenação assim acontece de maneira automática (GOODRICH; TAMASSIA, 2013). A figura 7 apresenta a sequência de união dos vetores.

Figura 7 – Processo de união realizado pelo algoritmo quicksort após finalizar o particionamento do vetor (os quadrados em laranja representam os menores elementos, os verdes representam o pivô selecionado e, por fim, os azuis indicam os elementos maiores que o pivô)



Como no processo de particionamento eles foram totalmente separados, a união ocorre tranquilamente e pode-se observar que é um processo de concatenação simples entre os elementos menores, o pivô e os elementos maiores nessa sequência específica.

A implementação do algoritmo quicksort é apresentada a seguir na linguagem de programação Java:

```
1. import java.util.Arrays;
2.
3. public class quicksort {
4.
5.     public static void sort(int X[], int inicio, int
fim) {
6.         if (inicio < fim) {
7.             int pivot = divide(X, inicio, fim);
8.             sort(X, inicio, pivot - 1);
9.             sort(X, pivot + 1, fim);
10.        }
11.    }
12.
13.    public static int divide(int X[], int inicio, int
fim) {
```

```
14.         int pivot = X[inicio];
15.         int postPivot = inicio;
16.         for (int i = inicio + 1; i <= fim; i++) {
17.             if(X[i] < pivot) {
18.                 X[postPivot] = X[i];
19.                 X[i] = X[postPivot + 1];
20.                 postPivot++;
21.             }
22.         }
23.         X[postPivot] = pivot;
24.         return postPivot;
25.     }
26.
27.     public static void main(String[] args) {
28.             int X[] = {29, 10, 36, 18, 83, 42, 8, 20};
29.             sort(X, 0, X.length - 1);
30.             System.out.printf("Ordenado: %s \n", Arrays.
toString(X));
31.     }
32.}
```

Assim como no algoritmo merge sort, o quicksort é dividido entre dois métodos. No método *sort* ocorrem as chamadas recursivas, após a identificação do pivô (linhas 5 a 11). No método *divide*, ocorre a divisão, entre as linhas 13 e 25. O método *divide* posiciona o pivô ao centro do vetor e coloca nas posições anteriores a ele os menores elementos, enquanto os maiores são colocados nas posteriores. Este é um trabalho realizado pela referência de posição do vetor, o que facilita o uso de memória de armazenamento. O desempenho é bem próximo ao de se criar vetores novos, mas, ao criar vetores, utiliza-se mais espaço de memória.

A visualização da execução do quicksort é também como uma árvore binária. Entretanto, a altura da árvore no quicksort pode ser linear, em seu pior caso. O pior caso do algoritmo acontece quando ele é aplicado em um conjunto de dados totalmente distintos e já ordenado. Considerando que o pior caso é algo raro de ocorrer, o quicksort tem se mostrado a melhor opção quando o assunto é ordenação.

3 Exercícios de fixação

Para praticar, segue uma lista de exercícios.

1. Execute o algoritmo merge sort para os vetores abaixo e compare as diferenças entre os resultados:
   a. [36, 15, 75, 2, 16, 48, 42, 51]
   b. [13, 45, 38, 20, 23, 7, 6, 5, 10, 11]
   c. [9, 8, 7, 6, 5, 4, 3, 2, 1]
2. Execute o algoritmo quicksort para os vetores abaixo e compare as diferenças entre os resultados:
   a. [36, 15, 75, 2, 16, 48, 42, 51]
   b. [13, 45, 38, 20, 23, 7, 6, 5, 10, 11]
   c. [9, 8, 7, 6, 5, 4, 3, 2, 1]
3. Compare o resultado entre os dois algoritmos.

Considerações finais

Os algoritmos demonstrados ao longo deste capítulo apresentam características mais sofisticadas para solucionar o problema da ordenação. Nos algoritmos apresentados, a técnica utilizada é a de dividir e conquistar, cuja intenção é dividir o problema complexo em pequenos subproblemas de resolução mais simples. Apesar de o cenário ser aplicado para a ordenação de dados, essa técnica pode ser empregada em qualquer cenário de soluções de problemas.

Quando se trata de ordenação, a técnica de divisão e conquista está amplamente vinculada ao processo de algoritmos recursivos, que nessa situação é a melhor opção para diminuir a complexidade da solução e deixar o código mais compreensível. Os dois algoritmos apresentados,

merge sort e quicksort, são considerados os algoritmos mais rápidos para ordenação, de modo geral. Outros algoritmos possuem tempos de execução mais rápidos, como o radix sort, mas limitam-se a alguns cenários, como o de dicionário de dados.

Referências

CORMEN, Thomas H. *et al*. **Algoritmos**: teoria e prática. Rio de Janeiro: Editora Campus, 2002.

FORBELLONE, André Luiz Villar; EBERSPACHER, Henri Frederico. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Prentice Hall, 2005.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados & algoritmos em Java**. Porto Alegre: Bookman Editora, 2013.

SEDGEWICK, Robert; FLAJOLET, Philippe. **An introduction to the analysis of algorithms**. [*S. l.*]: Pearson Education India, 2013.