1 Algoritmos de ordenação

Ordenar elementos é uma das tarefas mais comuns no cotidiano das organizações modernas. Ordenam-se nomes, números, logradouros, telefones, CPF, preços e qualquer tipo de informação, de forma manual ou automática. Começamos ordenando cartas de baralho, cada uma à sua maneira, e, da mesma forma, os computadores possuem diversos métodos para ordenar os dados armazenados.

Vários métodos de ordenação são conhecidos e são aplicados de acordo com o problema a ser resolvido. Grandes massas de dados podem ser ordenadas de uma forma diferente do que conjuntos de números ou mesmo dados semi-ordenados. Um método de ordenação pode ser mais eficiente para um caso e ruim para outro. Por isso é importante conhecer os métodos e entender quando devem ser aplicados.

Este capítulo apresentará alguns métodos de ordenação e vamos discutir sobre a aplicabilidade de cada um deles. Os algoritmos são simples e podem ser comparados a métodos manuais de ordenação, que podem ser aplicados a um baralho de cartas, por exemplo.

O método mais conhecido e, de longe, o mais usado é conhecido como Quicksort. Ele utiliza o princípio de divisão e conquista, o mesmo usado em inúmeras situações cotidianas de projeto de algoritmos. Por essa razão, entender esse método representa aprender uma das técnicas mais importantes para a construção de algoritmos. O algoritmo Quicksort apresenta muitas semelhanças com outro método, conhecido como Mergesort. Ambos serão apresentados.

Os outros métodos a serem apresentados incluem métodos mais simples, mas com aplicações específicas e eficientes, incluindo Seleção e Inserção. Por fim, o método Heapsort também merece ser analisado por utilizar uma estrutura de dados bastante interessante por princípio.

1.1 Definição do Problema

A tarefa de ordenação de dados pode ser algo bastante simples, como ordenar uma sequência de números inteiros. No entanto, pode ser algo também bem mais complexo, como ordenar todos os registros de um banco de dados como, por exemplo, o banco de dados de contribuintes da receita federal.

Para os conjuntos simples de dados, os algoritmos mais simples podem ser tão bons quanto os mais sofisticados. No entanto, para grandes volumes de dados, os algoritmos mais eficientes são necessários para se reduzir o tempo de processamento.

Para tratar o problema de ordenação vamos considerar um conjunto de registros identificados por uma chave, que será usada para a ordenação. A chave pode ser um nome, um número ou uma sequência qualquer de caracteres ou dígitos. Poderia ser o CPF ou um nome.

A ordenação será crescente pelo valor da chave. Assim, se a chave for um nome, os nomes iniciados com a letra ‘A’ devem ficar antes dos nomes iniciados com a letra ‘B’ e assim por adiante, seguindo a sequência de letras a partir da primeira letra inicial.

Os números devem ser considerados por seu valor, sendo os menores apresentados no início da lista após a ordenação.

Para simplificar os exemplos a serem apresentados, apenas sequências de números inteiros positivos serão consideradas. O problema básico apresentado no exemplo será constituído da leitura de 10 números, digitados pelo teclado, e em sua escrita de forma ordenada crescente.

Assim, caso a sequência de números:

31 61 7 89 34 13 19 92 11 23,

o resultado após a ordenação deverá ser:

7 11 13 19 23 31 34 61 89 92

E vai valer para qualquer método de ordenação.

O programa principal será sempre como mostrado a seguir, no qual a função ordena(V) será substituída pela função correspondente ao método de ordenação. Em C++, apresentado a seguir, a estrutura de dados **vector** será utilizada, não sendo necessário indicar o número de elementos a serem considerados, visto que é uma estrutura dinâmica.

Para ordenar estruturas de dados diferentes, basta substituir o tipo <int> na declaração do **vector** V pelo tipo correspondente, e alterar as comparações nos métodos de ordenação. Caso sejam usados registros, as chaves dos registros é que devem ser comparadas.

C++

```
#include <iostream>
#include <vector>
using namespace std;
...
int main() {
      int i, x;
      vector<int> V;
      for (i=0;i<10;i++){
            cin >> x;
            V.push_back(x);
      }
      ordena(V);
      for (int i=0;i<V.size();i++){
            cout << V[i] << endl;
```

```
        }
        return 0;
    }
```

A seguir, segue o código de referência em Java. Uma classe com o nome do método de ordenação será criada, e um método com o mesmo nome. O tipo de dados para a coleção de elementos será um vetor de inteiros (**int [ ]**) que deve ser inicializado com o número máximo de elementos a serem armazenados. A entrada de dados será por um objeto do tipo **Scanner.**

Java

```
import java.util.Scanner;
public class Ordena {

...
   public static void main(String[] args) {
          int i, x;
          int quantidade = 10;
          Scanner jin = new Scanner(System.in);
          int[] V = new int[quantidade];
          for (i=0;i<10;i++){
                V[i] = jin.nextInt();
          }
          ordena(V);
          for (i=0;i<V.length;i++){
                System.out.println(V[i]);
          }
   }
}
```

Os métodos de ordenação Seleção, Inserção, Quicksort e Heapsort serão apresentados a seguir, inseridos nesse contexto de ordenação de um conjunto de 10 inteiros, que podem ser ampliados para qualquer quantidade bem como qualquer tipo de dados.

1.2 Seleção

O primeiro método a ser apresentado é conhecido como Seleção ou SelectionSort. É o método mais simples. Provavelmente, se entregarmos um conjunto de cartas a pessoas para que ordenem de forma manual, a maioria usará esse método, devido à sua simplicidade.

O princípio básico do algoritmo Seleção é localizar, entre os elementos do conjunto, qual é o menor e trocá-lo com o elemento da primeira posição.

A figura X.x ilustra como o método Seleção funciona. A cada passagem, o menor elemento do conjunto é trocado com o elemento que está na primeira posição. Na sequência II, percebe-se a troca da carta de número 2 com a carta de número 7. Na sequência III, a carta de número 3 é que troca de posição com a carta de número 6. Em seguida, sequência IV, a carta de número 6 é trocada com a carta de número. Finalmente, na última sequência, a carta de número 7 é trocada com a carta de número 10, atingindo a ordem numérica crescente.

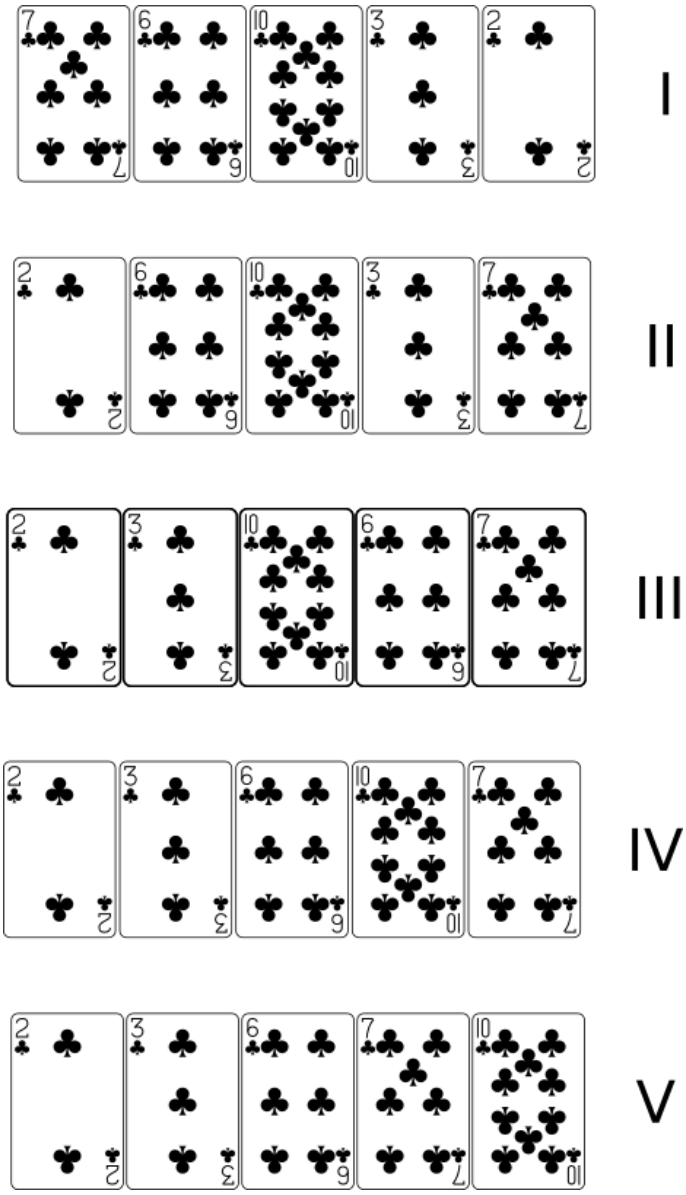

Figura X.x - Exemplo de funcionamento do método Seleção

O código relativo ao método Seleção é apresentado a seguir, em linguagem C++. A função recebe o vetor V, do tipo ***vector***, o qual estará ordenado após o término da execução da função. Como o conteúdo do vetor V será alterado durante a execução da função, ele deve ser passado por referência, usando o símbolo &, como pode ser visto no código.

Uma variável temporária do tipo inteira chamada menor é usada para registrar o índice do menor valor do conjunto. O algoritmo conta com dois laços do tipo **for.** O laço mais interno busca pelo primeiro elemento do conjunto que ainda não foi disposto em sua devida posição. O laço mais externo mantém esse processo até que todos os elementos do conjunto estejam em sua ordem correta.

Quando o menor elemento é encontrado, ele é trocado de posição com a primeira posição ainda não ordenada. A troca é realizada pela função **swap**. Essa função pode trocar dois elementos de posição de qualquer tipo de dados e pode ser mantida na solução mesmo que o tipo de dados a ser ordenado seja um registro com mais dados.

C++

```
void selecao(vector<int> &V){
      int menor;
      for (int i=0; i < V.size();i++){
            menor = i;
            for (int j=i+1;j<V.size();j++){
                  if (V[j] < V[menor])
                        menor = j;
            }
            swap(V[i], V[menor]);
      }
}
```

A seguir, é apresentado o código do método de ordenação Seleção em Java. É exatamente o mesmo algoritmo com as mesmas instruções. A única diferença é que não existe a função **swap** em Java. Assim, a troca deve ser feita usando uma variável temporária, do mesmo tipo de dados que se deseja ordenar.

Java

```
import java.util.Scanner;
public class Selecao {
      public static void selecao(int [] V){
            int menor;
            for (int i=0; i < V.length;i++){
                  menor = i;
                  for (int j=i+1;j<V.length;j++){
                        if (V[j] < V[menor])
                              menor = j;
                  }
                  int temp = V[i];
                  V[i] = V[menor];
                  V[menor]= temp;
            }
      }

      public static void main(String[] args) {
          int i, x;
          int quantidade = 10;
          Scanner cin = new Scanner(System.in);
          int[] V = new int[quantidade];
          for (i=0;i<10;i++){
                 V[i] = cin.nextInt();
          }
          selecao(V);
          for (i=0;i<V.length;i++){
                 System.out.println(V[i]);
          }
      }
}
```

Uma vantagem do método de ordenação Seleção é o pequeno número de trocas realizadas, correspondendo, no máximo, ao número de elementos do conjunto. Quando

aplicado a tipos mais robustos de dados, como registros heterogêneos, o baixo número de trocas é essencial para que o método seja executado com eficiência.

1.3 Inserção

O método de ordenação Inserção é o próximo a ser apresentado. Esse método também é bastante simples e também é um dos primeiros métodos escolhidos quando se deseja ordenar coleções de elementos no mundo real, como as cartas do baralho.

A diferença entre os métodos Inserção e Seleção está apenas na forma como o menor elemento é trazido para o início do conjunto. Enquanto no método Seleção, o menor elemento é trocado com o primeiro elemento ainda não ordenado do conjunto, no método Inserção, o menor elemento é trazido para o início do conjunto deslocando todos os elementos maiores uma posição à frente.

A figura X.X ilustra o método de ordenação Inserção. A cada passada, o primeiro elemento ainda não ordenado é comparado com os anteriores até descobrir qual a sua posição. Na sequência em I, a carta de número seis será escolhida para comparar com as cartas mais à esquerda. Como ela é menor que a carta de número sete, elas serão trocadas.

Em seguida, na sequência II, a carta de número dez será escolhida e comparada com as cartas mais à esquerda. Como ela é maior que a carta à sua esquerda, permanecerá na mesma posição. Em seguida, a carta de número três será selecionada. Comparando a carta de número três com as demais à esquerda, verifica-se que ela é a menor e todas as demais serão, então, deslocadas à direita para abrir espaço no início para a carta de número três.

Na sequência III, a carta de número dois é selecionada e comparada com as cartas à esquerda. Tal qual ocorreu com a carta de número três, ela é a menor de toda a sequência e vai assumir a primeira posição, deslocando todas as demais à direita. A sequência IV apresenta o resultado após a ordenação.

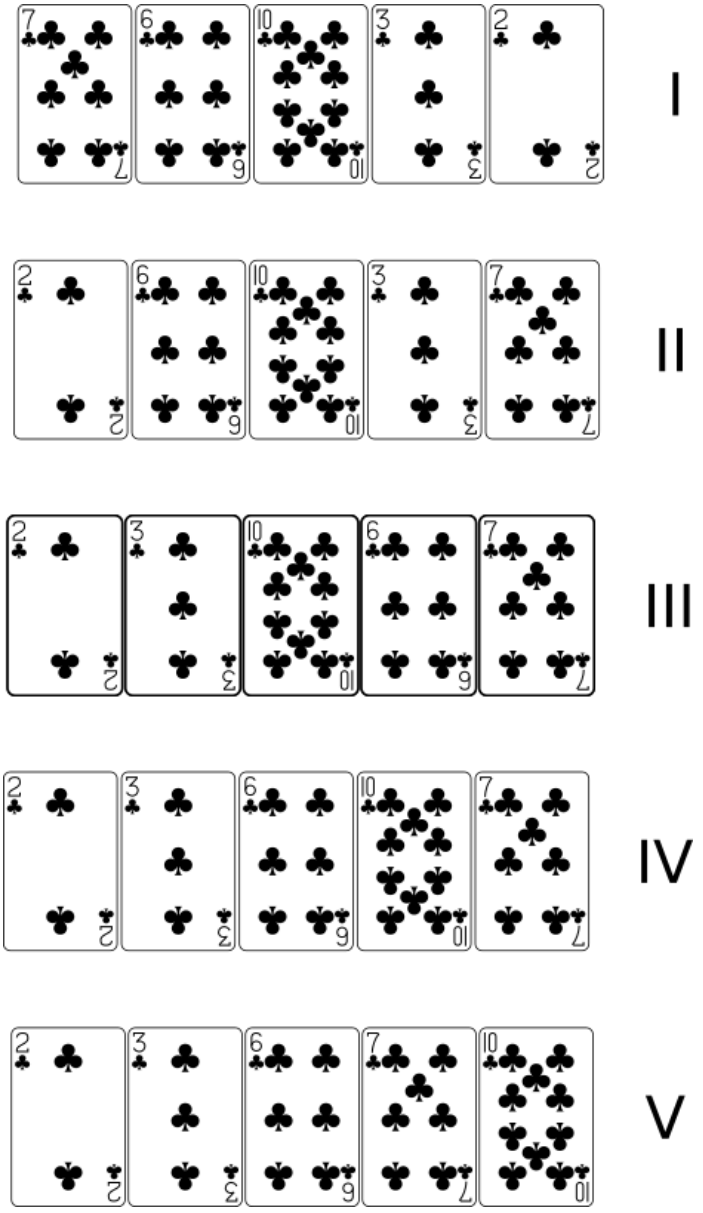

Figura X.x - Exemplo de funcionamento do método Inserção

A seguir, é apresentado o código em linguagem C++. A função **insercao** recebe o vetor V, declarado como **vector**, e promove a ordenação dos seus elementos. Como o valor do vetor V será alterado durante a função, ele deve ser passado por referência, razão pela qual o símbolo & é inserido em sua declaração.

O método conta com dois laços, tal qual o método de ordenação Seleção. O laço externo seleciona o elemento a ser comparado com os demais à direita já ordenados. O laço interno compara o elemento escolhido com os elementos à direita até encontrar um elemento menor, ou o início do vetor. Como a comparação é com os elementos à direita, a variável j de controle do laço é decrementada com o avanço do laço.

C++:

```
#include <iostream>
#include <vector>
using namespace std;
void insercao(vector<int> &V) {
    for (int i = 1; i < V.size(); i++) {
          int escolhido = V[i];
          for (int j=i-1;(j>=0)&&(V[j]>escolhido); j--) {
               V[j + 1] = V[j];
          }
          V[j + 1] = escolhido;
     }
}
int main() {
     int i, x;
     vector<int> V;
     for (i=0;i<10;i++){
          cin >> x;
          V.push_back(x);
     }
     insercao(V);
     for (int i=0;i<V.size();i++){
          cout << V[i] << endl;
     }
     return 0;
}
```

A seguir é apresentado o método de ordenação Inserção em Java. A função insercao é idêntica à função em C++, exceto pelo tipo de dado utilizado, vetor de inteiros (**int []**).

Java:

```
import java.util.Scanner;
public class Insercao {

   public static void insercao(int[] V) {
      int i, j, escolhido;
      for (i = 1; i < V.length; i++) {
           escolhido = V[i];
           for(j=i-1; (j >= 0) && (V[j] > escolhido); j--) {
                V[j + 1] = V[j];
           }
           V[j + 1] = escolhido;
      }
   }
   public static void main(String[] args) {
      int i, x;
      int quantidade = 10;
      Scanner cin = new Scanner(System.in);
      int[] V = new int[quantidade];
      for (i=0;i<10;i++){
         V[i] = cin.nextInt();
      }
      insercao(V);
      for (i=0;i<V.length;i++){
          System.out.println(V[i]);
      }
   }
}
```

O método de ordenação Inserção é bastante simples, mas não muito eficiente, assim como o método Seleção, exceto em alguns casos. Quando os dados já estão semi-ordenados, o método Inserção pode ser o melhor de todos. Esse caso é comum quando em sistemas que mantém os dados ordenados mas algumas inserções e movimentações podem desfazer a ordem em alguns elementos do conjunto. Para esses casos, o método Inserção é muito bom, pois é capaz de reordenar o conjunto com poucas movimentações.

1.4 Quicksort

O método de ordenação conhecido como Quicksort é o mais famoso e amplamente utilizado. É o mais eficiente na maioria das situações, razão pela qual é o método padrão da maioria dos sistemas. Ao ordenar dados em softwares como o Microsoft Excel® ou Word®, o método usado para essa ordenação é o Quicksort.

O princípio básico é conhecido como divisão-e-conquista. Esse termo, usado para definir um princípio básico de projeto de algoritmos, tem raízes históricas nos impérios macedônico e romano, na antiguidade, sendo utilizado por Júlio César (*divide et impera*) e Felipe da Macedônia. Na Antiguidade, o princípio indicava que era melhor fragmentar os territórios para conquistá-los e para mantê-los, visto que a administração e conflitos em territórios muito grandes seriam tarefas muito mais difíceis.

No projeto de algoritmos, o princípio de divisão-e-conquista indica que é mais fácil resolver um grande problema fragmentando em problemas menores. O caso mais comum é o Quicksort e eis o princípio: é mais fácil ordenar dois conjuntos menores do que um grande conjunto com a soma de todos os elementos.

O projeto de algoritmos de divisão-e-conquista deve considerar as seguintes fases:

1. divisão do conjunto em conjuntos menores;
2. processamento dos conjuntos menores; e,
3. reagrupamento no conjunto original, já com todos os dados processados.

Normalmente, os conjuntos são subdivididos até que tenham tão poucos elementos que o seu processamento se torna trivial. É o caso do Quicksort. Os subconjuntos são continuamente subdivididos até que tenham apenas um elemento. Os conjuntos unitários já estariam, assim, ordenados, dispensando a fase de processamento. Nesse caso, o mecanismo principal do algoritmo está no método de divisão ou de reagrupamento dos subconjuntos. Para o Quicksort, o mecanismo de ordenação atua na divisão dos subconjuntos.

Ao dividir os conjuntos, o método Quicksort gera dois subconjuntos já classificados com base em um elemento especial chamado pivô, sendo um subconjunto apenas com os elementos maiores que o pivô e o outro subconjunto apenas com os elementos menores que o pivô. O processo de reagrupamento se torna mais simples, pois basta concatenar os conjuntos de acordo com sua classificação em relação ao pivô.

É possível imaginar a aplicação do método de ordenação Quicksort para ordenar elementos do cotidiano, como as cartas do baralho. O método consiste em:

1. Escolher uma carta de referência, chamada pivô;
2. Dividir o conjunto de cartas em dois subconjuntos: um subconjunto das cartas com valor numérico maior que o pivô e o subconjunto das cartas com valor numérico menor que o pivô;
3. Retornar ao passo um, para cada subconjunto de forma separada, até que os subconjuntos tenham apenas um elemento.
4. Reagrupar os subconjuntos, em ordem, até obter o conjunto original ordenado.

1.4.1 Escolha do pivô

A principal questão a ser resolvida no método de ordenação Quicksort diz respeito à escolha do pivô, que serve de referência para a divisão do conjunto em dois subconjuntos. A escolha do pivô ideal faz com que essa divisão resulte em subconjuntos de igual tamanho, reduzindo os passos adicionais necessários para que se obtenham conjuntos unitários.

Como escolher o pivô ideal? Quando os elementos do conjunto a ser ordenado são bem conhecidos, como as cartas do baralho, essa escolha é mais fácil, pois é possível escolher um elemento intermediário da ordenação. No caso das catorze cartas de um naipe de um baralho, seria possível escolhe a carta de número sete como pivô e obter, assim, um subconjunto de seis e outro de sete cartas: as cartas menores que sete e as cartas maiores que sete. Se o pivô escolho for a carta de número oito, o resultado será bem parecido. No entanto, se a carta escolhida como pivô for o Ás ou o Rei, os subconjuntos ficarão totalmente desbalanceados, pois o conjunto de cartas maiores que o Ás ou menores que o Rei teriam quase totalidade dos elementos, exceção apenas do pivô.

Na grande maioria dos casos, não é possível prever como estão distribuídos os elementos do conjunto de forma que se torna impossível escolher o melhor pivô. Parte-se, então para a tentativa de escolher um bom pivô sem perder muito tempo.

O melhor pivô de um conjunto seria a mediana desse conjunto, ou seja, o elemento para o qual existem tantos elementos maiores quanto menores que ele no conjunto. Encontrar a mediana, no entanto, pode ser uma operação demorada, que não se justifica para esse caso, pois tornaria o método Quicksort lento. Tampouco seria viável calcular a média, além de não ter garantia de dividir o conjunto em partes iguais.

Sem ter um bom método para a escolha do pivô, normalmente toma-se um elemento qualquer do conjunto, que pode ser o primeiro, último ou um elemento tomado aleatoriamente.

Também existe a possibilidade de tomar a média de dois ou três elementos do conjunto, que pode minimizar a chance de obter um pivô próximo aos extremos do conjunto. Esse caso, no entanto, exclui o pivô da ordenação, visto que ele deixa de ser um elemento do conjunto, o que tem implicações práticas no resultado final, visto que a maioria das operações do método Quicksort são realizadas em subconjuntos pequenos, de poucos elementos, para os quais um elemento a mais possa fazer diferença no resultado final.

Um método de escolha do pivô que apresenta bons resultados é escolher a mediana entre três elementos do conjunto, normalmente tomados do início, meio e fim do conjunto. Assim, minimiza a chance de escolher um pivô nos extremos do conjunto e melhora o desempenho do método Quicksort na maioria dos casos.

A figura X.X apresenta uma ilustração da utilização do método de ordenação Quicksort sobre um conjunto de cartas de baralho. Para esse exemplo, o pivô foi escolhido sempre como o último elemento de cada subconjunto.

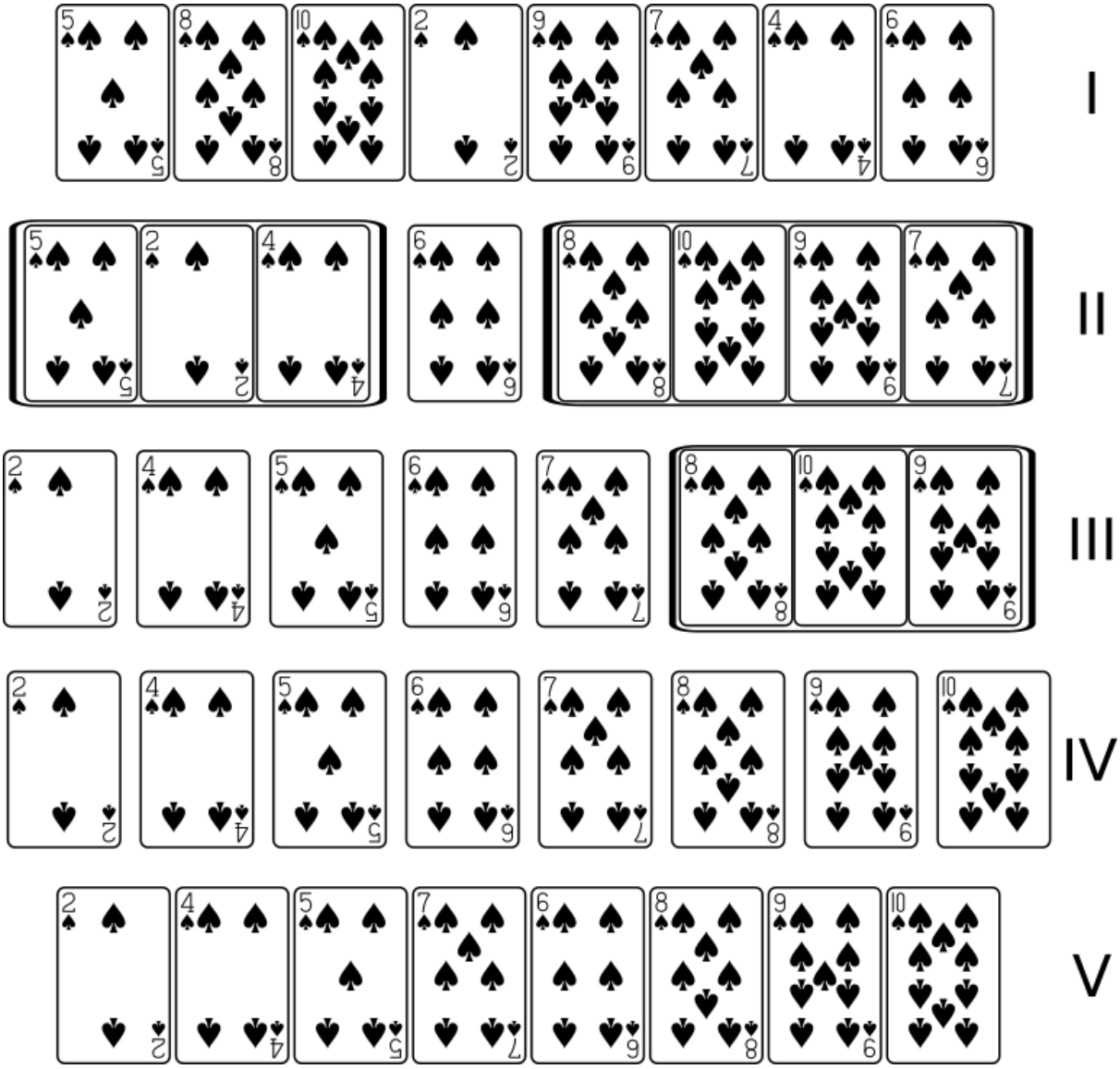

Figura X.X - Exemplo de funcionamento do método Quicksort

As cartas foram consideradas inicialmente como estão em I. O método Quicksort toma o último elemento, a carta de número seis, como pivô e divide o conjunto em dois subconjuntos, como pode ser visto em II: O subconjunto das cartas maiores que seis e o subconjunto das cartas menores que seis. A carta de número seis é considerada à parte e mantida entre os dois subconjuntos.

O próximo passo é subdividir os subconjuntos em II para obter conjuntos ainda menores. Em cada subconjunto, o último elemento é tomado como pivô, sendo a carta de número quatro no primeiro subconjunto e a carta de número sete no segundo subconjunto. O resultado é mostrado em III, o primeiro subconjunto, sendo dividido entre as cartas menores que quatro e as cartas maiores que quatro, terá como resultado dois subconjuntos unitários; e o segundo subconjunto terá como resultado um conjunto vazio, referente às cartas menores que sete, que será ignorado, e o subconjunto das cartas maiores que sete, que vai permanecer com três cartas.

Os subconjuntos unitários chegaram na sua menor subdivisão e já estão prontos para o reagrupamento. O subconjunto com três elementos ainda deve ser dividido mais

uma vez, considerando o último elemento, a carta de número nove, como pivô. Assim em IV, é mostrado o resultado da divisão deste último subconjunto em dois subconjuntos: cartas menores que nove e cartas maiores que nove. Ambos serão conjuntos unitários e o algoritmo já terminou todas as divisões, finalizando apenas com conjuntos unitários.

O último passo é o reagrupamento, como apresentado em V. Os subconjuntos individuais são concatenados em um conjunto com o mesmo número de elementos do conjunto original, porém ordenado.

É possível perceber claramente o efeito da escolha de um pivô ruim. Quando o pivô escolhido foi a carta de número sete, no último subconjunto em II, obtêm-se dois subconjuntos desbalanceados: um conjunto vazio e outro com três elementos.

O pior cenário de escolha do pivô é quando ele é um elemento de um dos extremos do conjunto, primeiro ou último, pois vai gerar um subconjunto vazio, como ocorreu no exemplo verificado. E esse caso ocorre com frequência quando se aplica o método Quicksort em conjuntos já previamente ordenados e o pivô é tomado no extremos, pois sempre vai considerar o maior ou o menor elemento na ordenação.

Para evitar essa situação, uma estratégia razoável é a escolha do pivô como a mediana de três elementos do conjunto. A figura X.X ilustra essa situação.

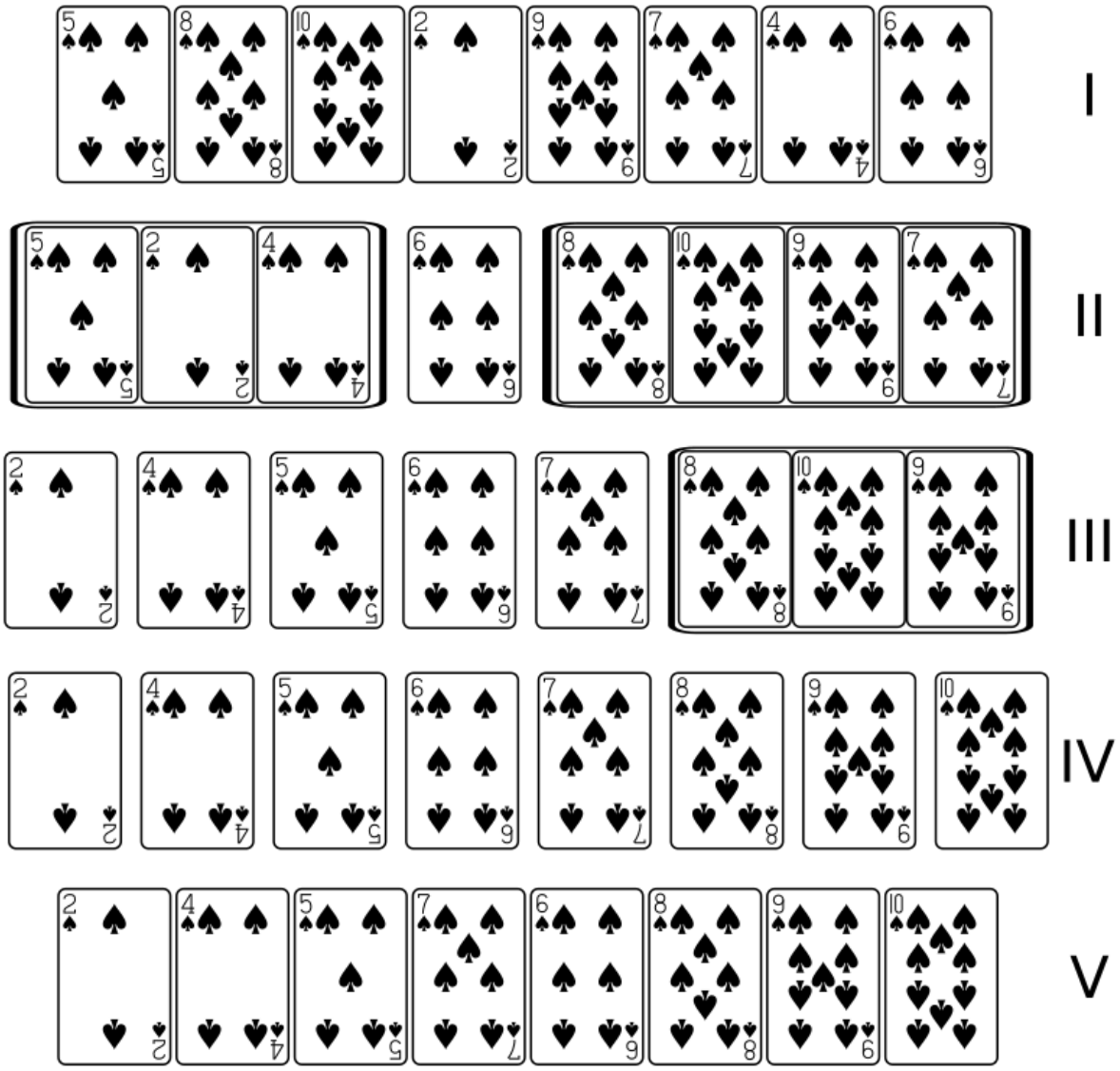

Figura X.X - Exemplo de funcionamento do método Quicksort usando a mediana de três elementos como pivô

O resultado da utilização da mediana de três elementos como pivô não vai diferir muito do resultado da utilização do último elemento como pivô porque o exemplo mostra tem poucos elementos a ordenar. Em conjuntos maiores, a diferença poderia ser mais significativa.

Para este exemplo, os pivôs já se alteram desde a primeira divisão. O pivô a ser escolhido em I será a mediana tomada entre as cartas de números cinco, seis e dois, que representam, respectivamente, o primeiro, o último e o elemento do meio do conjunto. O primeiro pivô será, então, a carta de número cinco, que resultará em um subconjunto com dois elementos e ouro com cinco elementos. Para essa divisão, a escolha do pivô pela mediana gerou dois subconjuntos menos balanceados que seria usando apenas o último elemento como pivô, mas, normalmente, a escolha da mediana é melhor que a escolha de apenas um elemento.

Para a segunda fase de divisão, em II, o último subconjunto, com 5 cartas, terá o pivô a partir da mediana das cartas de números seis, sete e dez, sendo, respectivamente o primeiro, o último e o elemento do meio deste subconjunto, resultando na carta de número sete como pivô. Os dois subconjuntos resultantes desta divisão podem ser vistos em III, sendo um apenas com a carta de número seis e outro com as cartas de número oito, nove e dez. Este último subconjunto será finalmente dividido em conjuntos unitários em IV, usando a carta de número oito como pivô, sendo a mediana entre as cartas de números oito, nove e dez.

A implementação do método de ordenação Quicksort é apresentada a seguir, em linguagem C++ e Java. Esta implementação é recursiva, e faz sucessivas chamadas ao método para cada subconjunto criado até que seja um conjunto unitário.

A essência do algoritmo Quicksort está na função o nome do algoritmo. Basicamente se o subconjunto é unitário, a função pode retornar, pois já concluiu o processo de subdivisão. Do contrário, um pivô é escolhido e a função **particao** é chamada para quebrar o conjunto em dois a partir do pivô. A função **particao** retorna a posição do pivô após a divisão. Os dois subconjuntos poderiam ser assim definidos: o primeiro mantém a posição inicial e tem como posição final a posição anterior ao pivô; o segundo tem como posição inicial a posição posterior ao pivô e mantém a posição final.

O pivô já está no sua devida posição após a função **particao**, pois esta função distribui os elementos menores à esquerda do pivô e maiores à direita. O algoritmo continua a processar os dois subconjuntos criados. E só conclui quando todos são unitários.

O reagrupamento dos vários subconjuntos unitários em um vetor único é uma consequência do algoritmo. Não é necessária nenhuma ação após o fim das partições. Todos os elementos já estão no vetor em sua devida posição ordenada.

A função **particao** realiza seu trabalho, deixando os elementos maiores que o pivô à direita e menores à esquerda. Existem pelo menos três implementações diferentes para a função **particao**. A lógica dela é a essência das diferenças entre as diversas implementações do Quicksort.

Para a implementação apresentada a seguir, os índices **i** e **j** percorrem o vetor a partir de cada extremidade, início e fim, respectivamente. O índice **i** percorre a partir

de **inicio**, procurando por elementos menores que o pivô. O índice **j** percorre a partir do fim, excluindo o pivô, procurando por elementos maiores que o pivô. Quando, e se, o índice **i** encontra um elemento maior que o pivô e o índice **j** encontra um elemento menor que o pivô, esses elementos são trocados. A procura persiste enquanto o índice **i** é menor ou igual ao índice **j**.

Quando os índices se encontram, é sinal que a busca já foi finalizada e os elementos menores que o pivô estão no início do vetor e os maiores já estão ao final do vetor. O último passo é trocar o pivô para a sua posição definitiva. Ao final do particionamento, o elemento de índice **i** é o primeiro elemento maior que o pivô do vetor. Ele pode ser trocado com o pivô, que está na última posição, para que o pivô fique no lugar certo.

C++

```
#include <iostream>
#include <vector>
using namespace std;
int particao( vector<int> &V, int inicio, int fim) {
   int pivo = V[fim];
   int i = inicio;
   int j = fim-1;
   while( i <= j ){
      while( V[i] < pivo )
         i++;
      while( j >= inicio && V[j] >= pivo  )
         j--;
      if( i < j )
         swap(V[i], V[j]);
   }
   swap(V[fim], V[i]);
   return i;
}
void quickSort(vector<int> &V, int inicio, int fim) {
      if (inicio < fim) { // Quando inicio = fim, a partição é
                          // unitária e deve retornar
            int corte = particao(V, inicio, fim);
```

```
            quickSort(V, inicio, corte - 1);
            quickSort(V, corte + 1, fim);
      }
}
int main() {
      int i, x;
      vector<int> V;
      for (i=0;i<10;i++){
            cin >> x;
            V.push_back(x);
      }
      quickSort(V, 0, 9);
      for (int i = 0; i < V.size(); i++)
            cout <<V[i]<<endl;
      return 0;
}
```

A implementação em Java é idêntica à implementação em C++, com exceção da função **swap**, inexistente em Java.

Java:

```
import java.util.Scanner;
public class Quicksort {
   public static int particao(int [] V, int inicio, int fim) {
         int pivo = V[fim];
         int i = inicio;
         int j = fim-1;
         while( i <= j ){
            while( V[i] < pivo )
               i++;
            while( j >= inicio && V[j] >= pivo )
                  j--;
```

```
                if( i < j ) {
                     int temp = V[i];
                     V[i] = V[j];
                     V[j] = temp;
                }
            }
            V[fim] = V[i];
            V[i] = pivo;
            return i;
      }
           public static void quickSort(int [] V, int inicio, int
fim) {
                if (inicio < fim) { // Quando inicio = fim, a
partição é unitária e deve retornar
                     int corte = particao(V, inicio, fim);
                     quickSort(V, inicio, corte - 1);
                     quickSort(V, corte + 1, fim);
                }
           }
           public static void main(String[] args) {
                int i, x;
               const int quantidade = 10;
               Scanner cin = new Scanner(System.in);
               int[] V = new int[quantidade];
               for (i=0;i<quantidade;i++){
                      V[i] = cin.nextInt();
                }
                quickSort(V, 0, quantidade-1);
                for (i=0;i<V.length;i++){
                       System.out.println(V[i]);
                }
           }
      }
```

Uma implementação alternativa da função **particao** é apresentada a seguir. Ela apresenta um código mais compacto, mas pode apresentar um número maior de trocas até chegar no particionamento esperado.

A implementação a seguir, apresentada em C++, utiliza o índice **i** para percorrer o vetor a procura de elementos menores que o pivô. Ao encontrar um elemento menor que o vetor, este é trocado com o elemento de índice **j,** que é incrementado. O índice **j** representa a partição do vetor que só tem elementos menores que o pivô à esquerda. Ele é incrementado sempre que um elemento menor que o pivô é encontrado. Para cada elemento encontrado, ele troca com um elemento de índice **j** à direita do vetor, além de incrementar o **j**.

Nessa implementação, um elemento maior que o pivô que esteja no início do vetor será trocado sucessivamente cada vez que for localizado um elemento menor que o vetor à direita, até que todos os elementos menores já estejam em seus devidos lugares. Essa função pode executar trocas dos elementos nas posições **i** e **j**, mesmo com esses índices iguais, o que resulta em processamento inútil. No entanto, é uma implementação mais compacta.

C++

```
int partition(vector<int> &V, int inicio, int fim) {
      int pivo = V[fim];
      int j = inicio - 1;
      for (int i = inicio; i < fim; i++) {
            if (V[i] < pivo) {
                  j = j + 1;
                  swap(V[j], V[i]);
            }
      }
      V[fim] = V[j + 1];
      V[j + 1] = pivo;
      return (j + 1);
}
```

A implementação em Java da função **particao** alternativa é mostrada a seguir.

Java:

```
public static int particao(int [] V, int inicio, int fim) {
            int pivo = V[fim];
            int j = inicio - 1;
            for (int i = inicio; i < fim; i++) {
                  if (V[i] < pivo) {
                        j = j + 1;
                        int temp = V[j];
                        V[j] = V[i];
                        V[i] =  temp;
                  }
            }
            V[fim] = V[j + 1];
            V[j + 1] = pivo;
            return (j + 1);
}
```

Existem ainda outras implementações para a função de particionamento do Quicksort que podem realizar mais ou menos trocas para chegar no solução particionada. O uso de uma ou outra implementação pode levar à maior eficiência do algoritmo, bem como pode ser uma solução mais compacta.

O algoritmo Quicksort e suas variações tem sido aceito de forma quase unânime como melhor implementação a ser usada em casos gerais, sejam em editores de textos, planilhas ou em gerenciadores de arquivos. É a implementação mais comum e mais utilizada, devido a seus bons resultados. É possível mostrar que sua execução está entre as mais rápidas, o que será visto no capítulo sobre Complexidade Computacional.

1.5 Heapsort

O algoritmo de ordenação *Heapsort* é uma alternativa ao Quicksort cujo tempo de execução é bem próximo para a maioria dos casos. É um algoritmo eficiente. Seu funcionamento é baseado na estrutura de dados conhecida como árvore binária.

Uma árvore binária é uma estrutura de dados abstrata hierárquica no qual cada elemento possui ligação com até dois elementos no nível inferior. O nível mais alto da árvore é conhecido como raiz. A representação da árvore como vetor é denominada *Heap*. O algoritmo *Heapsort* tem como base a construção de uma árvore binária na qual os maiores elementos são superiores na árvore, chamada, nesse caso, de *heap*

máximo. O maior elemento do conjunto sempre estará na posição superior da árvore, que será sempre facilmente reconstruída após sua saída.

A figura X mostra uma representação de um heap máximo. Para facilitar a implementação, o heap será implementado como vetor. A primeira posição do vetor será do elemento raiz. Os filhos de cada elemento *i* da árvore estão nas posições 2*i* e 2*i* + 1, respectivamente.

Figura X - Exemplo de árvore binária que corresponde a um heap máximo

O vetor correspondente à Figura X ficaria da seguinte forma:

1						

O algoritmo *Heapsort* inicia transformando um *heap* qualquer em um *heap* máximo. Isso significa, basicamente, em verificar as posições 2*i* e 2*i* + 1 do vetor, comparando com a posição i. O maior desses elementos deve ser movimentado para a posição *i*. A análise do vetor começa da última para a primeira posição. Assim, ao final da verificação, a primeira posição vai conter o maior elemento desse vetor. Esse elemento pode ser movido para a última posição do vetor e o processo se reinicia, já considerando o vetor apenas até a penúltima posição.

A figura X mostra como o algoritmo funciona. Após a construção do *heap* máximo que pode ser visto em I, os maiores elementos vão sendo enviados para o final do vetor, sendo trocados com o elemento que se encontra na última posição válida e o *heap* máximo é reconstruído. Em II, o elemento 2, que se encontra na última posição do vetor, foi trocado com o meio elemento 11. A seguir, em III, o *heap* máximo é reconstruído, da esquerda para a direita. O elemento 2 será comparado e trocado com 9 e depois com 3. O elemento 3 é, ainda, trocado com o elemento 5 para obter novamente o *heap* máximo.

Em IV, o processo se repete, trocando 9 com 4. Em V, o *heap* máximo é refeito, trocando 4 com 5 e, depois, 5 com 8. Em VI, o elemento 8 é retirado, sendo substituído por 3, tendo o *heap* máximo refeito em VII. Em VIII e X temos novamente a retirada do maior elemento, sendo substituído pelo último elemento do vetor com a reconstrução do *heap* máximo, respectivamente, em IX e XI, até sobrar apenas um elemento, o menor do conjunto.

O código do algoritmo de ordenação Heapsort é apresentado a seguir. A função heapify constroi o *heap* máximo, invertendo a posição dos elementos na árvore. A função heapsort realiza a ordenação. O primeiro passo é construir um *heap* máximo e isso é feito no primeiro laço for. A função *heapify* é chamada sucessivamente, na metade superior do vetor, para que todos os elementos possam ser verificados em sua colocação na árvore.

Após a execução do laço for na função heapsort, o vetor já se encontra na forma de um heap máximo. A partir de então, o primeiro elemento é trocado pelo último, e a função heapify é chamada novamente para recolocar o elemento trocado em seu devido

lugar e refazer o heap máximo. Esse processo é repetido no segundo laço for até que todos os elementos estejam ordenados.

Na implementação em linguagem C++ e Java, como o vetor se inicia no elemento 0, os seus filhos na árvore estarão nas posições 2*i*+1 e 2*i*+2.

C++

```
#include <iostream>
#include <vector>
using namespace std;
void heapify(vector<int> &V, int i)
{
    int maior = i;  // Inicializa o maior como raiz
    int esquerda = 2*i + 1;  // esquerda = 2*i + 1
    int direita = 2*i + 2;  // direita = 2*i + 2
    // Se o filho a esquerda  eh maior que a raiz
    if (esquerda < tam && V[esquerda] > V[maior])
        maior = esquerda;
    // Se o filho a direita eh maior que a raiz
    if (direita < tam && V[direita] > V[maior])
        maior = direita;
    // Se o maior nao eh a raiz
    if (maior != i)
    {
        swap(V[i], V[maior]);
        // Chama recursivamente na subarvore
        heapify(V, tam, maior);
    }
}
// main function to do heap sort
void heapSort(vector<int> &V)
{
    // Constroi o heap
    for (int i = V.size() / 2 - 1; i >= 0; i--)
```

```
            heapify(V, V.size(), i);
        // Extrai os elementos do heap, um a um
        for (int i=V.size()-1; i>=0; i--)
        {
            // Move a raiz corrente para o final
            swap(V[0], V[i]);
            heapify(V, i, 0);
        }
    }
    int main()
    {
         int i, x;
         vector<int> V;
         for (i=0;i<10;i++){
               cin >> x;
               V.push_back(x);
         }
         heapSort(V);
         for (int i = 0; i < V.size(); i++)
               cout <<V[i]<<endl;
         return 0;
    }
```

A seguir, é apresentado o código correspondente em Java. A principal diferença é novamente na ausência da função swap em Java.

Java

```
public class Heapsort {
     public static void heapify(int [] V, int tam, int i)
     {
         int maior = i;  // Inicializa o maior como raiz
```

```
        int esquerda = 2*i + 1;  // esquerda = 2*i + 1
        int direita = 2*i + 2;  // direita = 2*i + 2
        // Se o filho à esquerda  eh maior que a raiz
        if (esquerda < tam && V[esquerda] > V[maior])
            maior = esquerda;
        // Se o filho a  direita eh maior que a raiz
        if (direita < tam && V[direita] > V[maior])
            maior = direita;
        // Se o maior não é a raiz
        if (maior != i)
        {
            int temp = V[i];
            V[i] = V[maior];
            V[maior] = temp;
            // Chama recursivamente na subarvore
            heapify(V, tam, maior);
        }
    }
    // main function to do heap sort
    public static void heapSort(int [] V)
    {
        // Constroi o heap
        for (int i = V.length / 2 - 1; i >= 0; i--)
            heapify(V, V.length, i);
        // Extrai os elementos do heap, um a um
        for (int i=V.length-1; i>=0; i--)
        {
            // Move a raiz corrente para o final
            int temp = V[0];
            V[0] = V[i];
            V[i] = temp;
            heapify(V, i, 0);
        }
```

```
        }
        public static void main(String[] args) {
            int i, x;
            int quantidade = 10;
            Scanner cin = new Scanner(System.in);
            int[] V = new int[quantidade];
            for (i=0;i<10;i++){
                   V[i] = cin.nextInt();
             }
             heapSort(V);
             for (i=0;i<V.length;i++){
                    System.out.println(V[i]);
             }
        }
    }
```

Exercícios

1) Considere a sequência de números a seguir. Mostre a sequência de trocas para que a ordenação seja realizada seguindo os métodos Seleção, Inserção, Quicksort e Heapsort.

   5 9 2 8 11 4 7

2) Considere um sequência previamente ordenada sendo submetida aos algoritmos de ordenação Seleção, Inserção, Quicksort e Heapsort. Qual método executaria mais rapidamente recebendo uma sequência previamente ordenada?