1 - Heapsort

O algoritmo de ordenação heapsort (https://pt.wikipedia.org/wiki/Heapsort) é melhor aplicado a uma estrutura do tipo array, pois a forma de percorrer os elementos se baseia em simular o comportamento de uma árvore binária usando os índices do array, veja o exemplo da Figura 1, onde os elementos estão representados na cor vermelha e os índices na cor azul.

Considerando que o 1º elemento do array está na posição 1 - não é zero, como nas linguagens baseadas no padrão C, tem-se que:

- O elemento raiz está no índice 1;
- Se um pai está no índice *p* então o seu filho esquerdo *fe* estará na posição 2**p* e o seu filho direito *fd* estará na posição 2**p* + 1. De forma equivalente pode-se dizer que um filho (esquerdo ou direito) que possui índice *f* possui um pai que está no índice (int)(*f*/2).

A operação fica mais simples ao considerar que o 1º índice do conjunto está na posição 1. Porém, para implementar essa ideia no Java é necessário somar 1 no índice do elemento pai e subtrai 1 no índice dos elementos filhos, pois o 1º elemento do array estará no índice 0.

O algoritmo heapsort possui as seguintes características:

- Não necessita de um array auxiliar para ser implementado;
- Ele não é estável – muda a ordem relativa de elementos com valores iguais;
- Ordem de complexidade: O(n log n) em todos os casos.

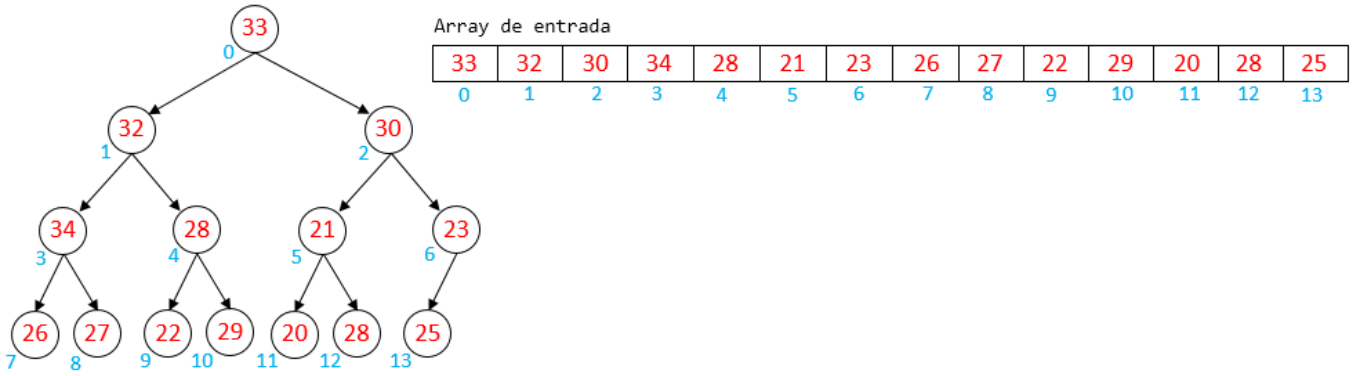

Figura 1 – Interpretação de um array como árvore binária.

A Figura 2 mostra uma implementação do algoritmo, a Figura 3 mostra o caso de teste da classe Heapsort e a Figura 4 o resultado desse teste. A seguir tem-se as operações do algoritmo:

- O 1º passo do algoritmo é criar um heap (amontoar) do array, onde cada elemento pai possui valor maior que os seus filhos, veja como exemplo a representação da Figura 5;
- O 2º passo do algoritmo é encontrar sucessivamente o maior valor e colocar no final do array:
  - Na 1ª iteração é encontrado o maior valor em [0..n-1] e coloca-o na posição n-1 do array (Figura 6);
  - Na 2ª iteração é encontrado o maior valor em [0..n-2] e coloca-o na posição n-2 do array (Figura 7);
  - Na 3ª iteração é encontrado o maior valor em [0..n-3] e coloca-o na posição n-3 do array (Figura 8).

  Veja que a cada iteração o conjunto possui n-1 elementos que na iteração anterior. A Figura 9 mostra o array ordenado após todas as iterações.

```
public class Heapsort {
   public void heapsort(int[] v) {
      System.out.print("Array de entrada:\n");
      for( int a:v ){
         System.out.print( a +" " );
      }

      criaHeap(v); /* ao final desse passo o pai será maior que os filhos em toda a árvore */

      System.out.print("\nArray heap:\n");
      for( int a:v ){
         System.out.print( a +" " );
      }

      for(int i = v.length - 1, n=v.length; i > 0; i--) {
         troca(v, i, 0); /* troca o elemento i (em ordem decrescente) com o raiz */

         System.out.print("\nIteração i="+i+":\n");
         for( int a:v ){
            System.out.print( a +" " );
         }
         /* Percorre a árvore de cima para baixo.
         * Ao final do 1a iteração o maior valor estará na última posição,
         * ao final da 2a iteração o 2o maior valor estará na penúltima posição e
         * assim sucessivamente */
         heap(v, 0, --n); /* note que o intervalo 0..n diminui a cada iteração */
      }
   }

   /* percorre a 1a metade do array */
   private void criaHeap(int[] v){
      for(int i = v.length/2-1; i >= 0; i--){
         heap(v, i, v.length);
      }
   }

   /* rearranja os elementos */
   private void heap(int[] v, int pai, int n){
      int fe = 2 * pai + 1, fd = fe + 1;
      int maior = fe; /* índice do maior valor entre os filhos esquerdo e direito */
        if( fe < n ){

            if( fd < n && v[fe] < v[fd] ){ /* para ordenar no outro sentido basta substituir v[fe] > v[fd] */
                maior = fd; /* obtém o maior valor entre os filhos esquerdo e direito */
            }
            if( v[maior] > v[pai] ){ /* para ordenar no outro sentido basta substituir v[fe] < v[pai] */
               troca(v, maior, pai); /* o maior valor entre os filhos sobe para o pai */
               heap(v, maior, n);
            }
        }
   }

   private void troca(int[] v, int a, int b){
      int aux = v[a];
      v[a] = v[b];
      v[b] = aux;
   }
}
```

Figura 2 – Classe Heapsort.

```
public class Principal {
   public static void main(String[] args) {
      int[] v1 = {14,8,13,8,15,17,10,-10,4,15,8,12,17,12,-1};
      int[] v2 = {-10,-1,4,8,8,8,10,12,12,13,14,15,15,17,17};
      int[] v3 = {33,32,30,34,28,21,23,26,27,22,29,20,28,25};

      Heapsort hs = new Heapsort();
      hs.heapsort(v3);
   }
}
```

Figura 3 – Classe Principal para testar o algoritmo Heapsort.

```
Array de entrada:
33 32 30 34 28 21 23 26 27 22 29 20 28 25
Array heap:
34 33 30 32 29 28 25 26 27 22 28 20 21 23
Iteração i=13:
23 33 30 32 29 28 25 26 27 22 28 20 21 34
Iteração i=12:
21 32 30 27 29 28 25 26 23 22 28 20 33 34
Iteração i=11:
20 29 30 27 28 28 25 26 23 22 21 32 33 34
Iteração i=10:
21 29 28 27 28 20 25 26 23 22 30 32 33 34
Iteração i=9:
21 28 28 27 22 20 25 26 23 29 30 32 33 34
Iteração i=8:
23 27 28 26 22 20 25 21 28 29 30 32 33 34
Iteração i=7:
21 27 25 26 22 20 23 28 28 29 30 32 33 34
Iteração i=6:
23 26 25 21 22 20 27 28 28 29 30 32 33 34
Iteração i=5:
20 23 25 21 22 26 27 28 28 29 30 32 33 34
Iteração i=4:
22 23 20 21 25 26 27 28 28 29 30 32 33 34
Iteração i=3:
21 22 20 23 25 26 27 28 28 29 30 32 33 34
Iteração i=2:
20 21 22 23 25 26 27 28 28 29 30 32 33 34
Iteração i=1:
20 21 22 23 25 26 27 28 28 29 30 32 33 34
```

Figura 4 – Resultado do código da Figura 3.

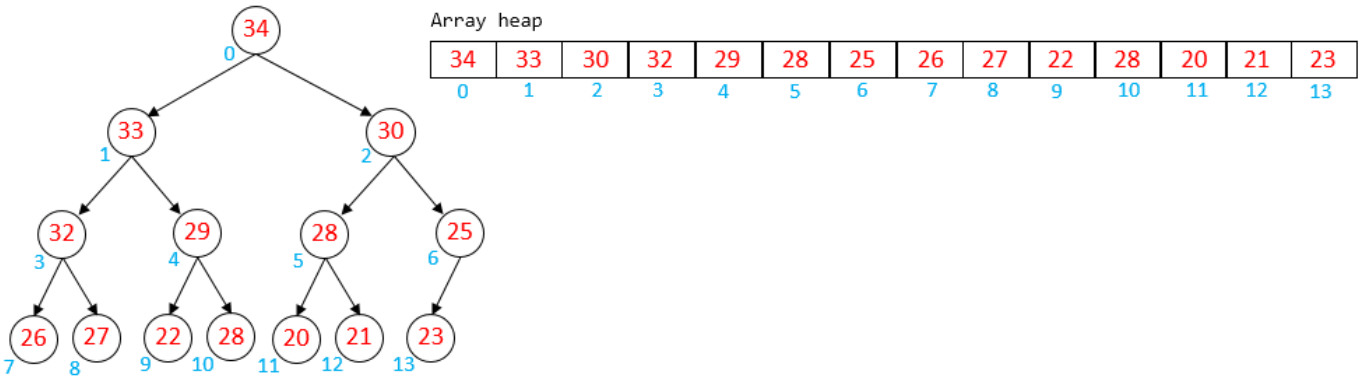

Figura 5 – Array da Figura 1 após a operação de heap – todas os nós pai são maiores que os filhos.

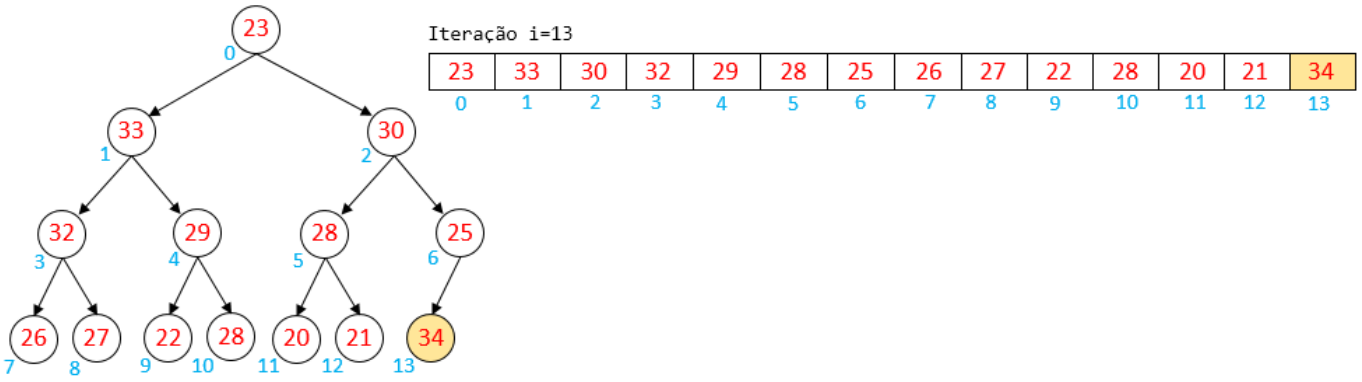

Figura 6 – Array da Figura 5 após a 1ª iteração para rearranjar os elementos –o último elemento está ordenado.

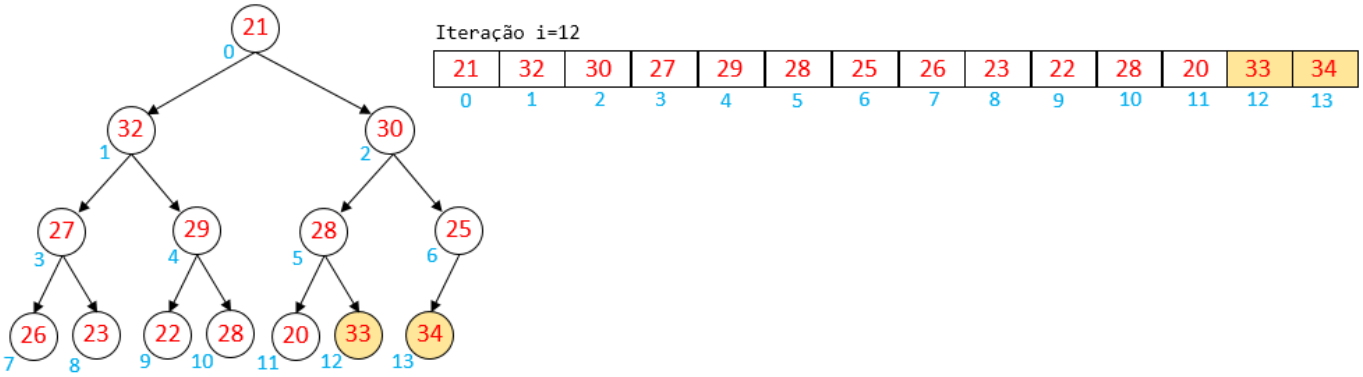

Figura 7 – Array da Figura 5 após a 2ª iteração para rearranjar os elementos –os elementos [i..n] estão ordenados.

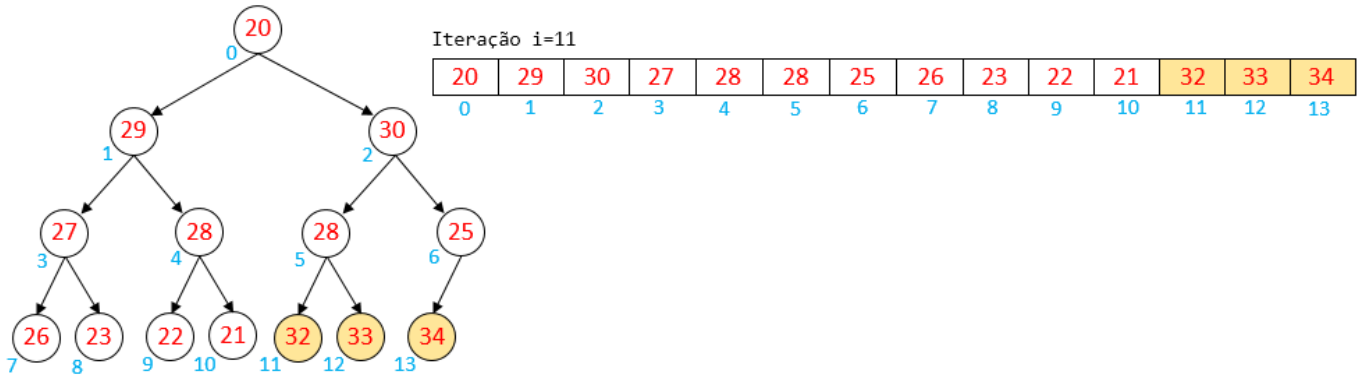

Figura 8 – Array da Figura 5 após a 3ª iteração para rearranjar os elementos –os elementos [i..n] estão ordenados.

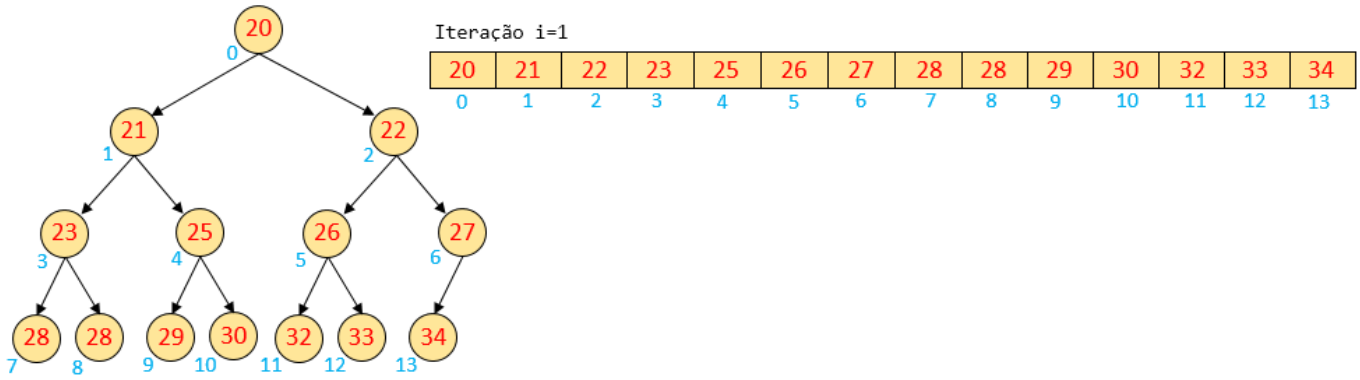

Figura 9 – Array da Figura 5 após a nª iteração para rearranjar os elementos –os elementos [0..n] estão ordenados.

2 - Mergesort

O algoritmo de ordenação mergesort (https://pt.wikipedia.org/wiki/Merge_sort) é melhor aplicado a uma estrutura do tipo array, pois a forma de percorrer os elementos se baseia em dividir sucessivamente o array pela metade para ordenar os elementos dos subarrays. A Figura 10 mostra uma implementação desse algoritmo e a Figura 11 um caso de teste do algoritmo.

A quantidade de elementos do array é reduzida à metade a cada passo da recursão.

- Na 1ª iteração, o número de elementos do array será reduzido por 2: v[0..n/2-1] e v[n/2..n-1];

- Na 2ª iteração, o número de elementos do array será reduzido por 4: v[0..n/4-1], v[n/4..n/2-1], v[n/2..3n/4-1] e v[3n/4..n-1].

E assim sucessivamente, até que, na última iteração, cada conjunto tenha no máximo 1 elemento. O número total de iterações é aproximadamente log n.

O algoritmo usa o conceito de dividir e conquistar:

- Ao dividir o array em subarrays, o número de elementos a serem manipulados em cada subarray será menor, como resolver as duas metades de um problema é mais simples que resolver ele por inteiro e
- Conquistar tem o significado de unir os subarrays ordenados.

O algoritmo mergesort possui as seguintes características:

- Melhor aplicado para grandes conjuntos de dados e ruim para pequenos conjuntos, já que não se justifica o custo de particionar e juntar o array;
- Ele necessita de um array auxiliar para ser implementado;
- Ele é estável – não muda a ordem relativa de elementos com valores iguais;
- Ordem de complexidade: O(n log n) em todos os casos.

```java
public class Mergesort {

   public void mergesort(int[] v){
      if( v != null && v.length > 1 ){
         mergesort(v, 0, v.length);
      }
   }

   /* divide recursivamente o array */
   private void mergesort(int v[], int inicio, int fim){
      if( inicio < fim-1 ) {
         int meio = (inicio + fim)/2;
         mergesort(v, inicio, meio);
         mergesort(v, meio, fim);
         merge(v, inicio, meio, fim);
      }
   }

   /* merge é a junção de dois arrays */
   private void merge(int v[], int inicio, int meio, int fim){
      int i, j;
      /* array auxiliar para receber os elementos a serem ordenados */
      int[] aux = new int[fim-inicio];
      /* copia o trecho do array a ser ordenado */
      for(i = inicio; i < meio; i++){
         aux[i-inicio] = v[i];
      }
      for(j = meio; j < fim; j++){
         aux[(fim-inicio)+meio-j-1] = v[j];
      }

      /* escreve o resultado ordenado no array */
      i = 0; /* será o índice para a 1a metade do array auxiliar */
      j = fim-inicio-1; /* será o índice para a 2a metade do array auxiliar*/
      for (int k = inicio; k < fim; k++){
         if( aux[i] <= aux[j] ){
            v[k] = aux[i++];
         }
         else{
            v[k] = aux[j--];
         }
      }
   }
}
```

Figura 10 – Classe Mergesort.

```
public class Principal {
   public static void main(String[] args) {
      int[] v1 = {14,8,13,8,15,17,10,-10,4,15,8,12,17,12,-1};
      int[] v2 = {-10,-1,4,8,8,8,10,12,12,13,14,15,15,17,17};
      int[] v3 = {33,32,30,34,28,21,23,26,27,22,29,20,28,25};

      Mergesort ms = new Mergesort();
      ms.mergesort(v3);
   }
}
```

Figura 11 – Classe Principal para testar o algoritmo Mergesort.

3 - Quicksort

Assim como o a algoritmo mergesort, o algoritmo de ordenação quicksort (https://pt.wikipedia.org/wiki/Quicksort) usa o conceito de dividir e conquistar.

A estratégia consiste em escolher um elemento no array, chamado de pivô, e rearranjar os demais elementos de modo que todos os valores menores que o pivô sejam colocados à esquerda do pivô e todos aqueles que sejam maiores sejam colocados à direita. Ao final dessa iteração:

- Repete o procedimento nos subarrays a esquerda e a direita do pivô.

O desafio é encontrar um bom elemento como pivô, onde o bom é aquele valor que está no meio da sequência de valores – ou seja, não exatamente de posição (índice). Um pivô que após o rearranjo fica no início ou final do array não é um bom pivô, pois fará com que sejam necessários mais iterações recursivas para ordenar os elementos dos subarrays.

A Figura 12 mostra uma implementação desse algoritmo, a Figura 13 um caso de teste e a Figura 14 mostra o processo de ordenação do array em cada iteração. Nesse exemplo de implementação o pivô é o elemento que está no meio do array, isto pode se tornar uma péssima escolha se o valor desse elemento está nos limites inferior ou superior do conjunto de valores do array.

O algoritmo quicksort possui as seguintes características:

- Aplicado para grandes conjuntos de dados;
- Não necessita de um array auxiliar para ser implementado;
- Ele não é estável – muda a ordem relativa de elementos com valores iguais;
- Ordem de complexidade:
  - O(n log n) no melhor e no caso médio;
  - O(n^2) no pior caso.

```
public class Quicksort {
   public void quicksort(int[] v){
      if( v != null && v.length > 1 ){
         quicksort(v, 0, v.length-1);
      }
   }

   /* escolhe como pivô o elemento que está no meio do array v[inicio..fim] e
    * faz com que os valores à esquerda do pivô sejam menores que o pivô e
    * os valores à direita sejam maiores que o pivô */
   public void quicksort(int v[], int inicio, int fim){
       int pivo, i, j;
       /* escolhe o valor pivô */
      pivo = v[(inicio+fim)/2];
      i = inicio;
      j = fim;
```

```
        System.out.print("\nInício:"+inicio+" Pivo:"+pivo + " Fim:"+ fim+"\n");
        imprimir(v);

        /* ao final de cada iteração dois elementos serão trocados de posição */
        while(i <= j){
            /* identifica um valor maior que o pivô à esquerda do pivô */
            while(v[i] < pivo){
                i++;
            }
            /* identifica um valor menor que o pivô à direita do pivô */
            while(v[j] > pivo){
                j--;
            }

            if(i <= j){
                troca(v, i, j); /* troca o menor e maior valor de posição */
                i++;
                j--;
            }
        }
        /* particiona o array e faz nova organização nos elementos */
        if(j > inicio){
            quicksort(v, inicio, j);
        }
        if(i < fim){
            quicksort(v, j+1, fim);
        }
    }

    private void troca(int[] v, int a, int b){
        int aux = v[a];
        v[a] = v[b];
        v[b] = aux;
    }

    public void imprimir(int[] v){
        for(int a:v){
            System.out.print(a + " ");
        }
    }
}
```

Figura 12 – Classe Quicksort.

```
public class Principal {
    public static void main(String[] args) {
        int[] v1 = {14,8,13,8,15,17,10,-10,4,15,8,12,17,12,-1};
        int[] v2 = {-10,-1,4,8,8,8,10,12,12,13,14,15,15,17,17};
        int[] v3 = {33,32,30,34,28,21,23,26,27,22,29,20,28,25};

        Quicksort qs = new Quicksort();
        System.out.print("\nArray de entrada:\n");
        qs.imprimir(v3);
        qs.quicksort(v3);
        System.out.print("\nArray ordenado:\n");
        qs.imprimir(v3);
    }
}
```

Figura 13 – Classe Principal para testar o algoritmo Quicksort.

```
Array de entrada:
33 32 30 34 28 21 23 26 27 22 29 20 28 25
Início:0 Pivo:23 Fim:13
33 32 30 34 28 21 23 26 27 22 29 20 28 25
Início:0 Pivo:22 Fim:3
20 22 23 21 28 34 30 26 27 32 29 33 28 25
Início:0 Pivo:20 Fim:1
20 21 23 22 28 34 30 26 27 32 29 33 28 25
Início:2 Pivo:23 Fim:3
20 21 23 22 28 34 30 26 27 32 29 33 28 25
Início:4 Pivo:27 Fim:13
20 21 22 23 28 34 30 26 27 32 29 33 28 25
Início:4 Pivo:27 Fim:6
20 21 22 23 25 27 26 30 34 32 29 33 28 28
Início:4 Pivo:25 Fim:5
20 21 22 23 25 26 27 30 34 32 29 33 28 28
Início:7 Pivo:29 Fim:13
20 21 22 23 25 26 27 30 34 32 29 33 28 28
Início:7 Pivo:28 Fim:9
20 21 22 23 25 26 27 28 28 29 32 33 34 30
Início:8 Pivo:28 Fim:9
20 21 22 23 25 26 27 28 28 29 32 33 34 30
Início:10 Pivo:33 Fim:13
20 21 22 23 25 26 27 28 28 29 32 33 34 30
Início:10 Pivo:32 Fim:11
20 21 22 23 25 26 27 28 28 29 32 30 34 33
Início:12 Pivo:34 Fim:13
20 21 22 23 25 26 27 28 28 29 30 32 34 33
Array ordenado:
20 21 22 23 25 26 27 28 28 29 30 32 33 34
```

Figura 14 – Resultado do código da Figura 13 – as marcações em vermelho indicam os elementos pivô em cada iteração e as marcações em azul indicam os elementos do array em manipulação.

4 - Exercícios

1 – Alterar a classe Heapsort para ordenar textos em ordem crescente.

2 – Alterar a classe Mergesort para ordenar textos em ordem crescente.

3 – Alterar a classe Quicksort para ordenar textos em ordem crescente.