DESCRIÇÃO

Apresentação dos algoritmos de ordenação avançados (merge sort, quick sort e shell sort) e discussão sobre suas complexidades.

PROPÓSITO

Entender o funcionamento dos algoritmos de ordenação básicos e avançados é essencial para os profissionais de Tecnologia da Informação (TI) escolherem corretamente o algoritmo mais adequado à solução do problema a ser tratado.

OBJETIVOS

MÓDULO 1

Reconhecer os algoritmos de ordenação elementares

MÓDULO 2

Descrever o funcionamento do algoritmo de ordenação por intercalação (*merge sort*)

MÓDULO 3

Descrever o funcionamento do algoritmo de ordenação rápida (*quick sort*)

MÓDULO 4

Descrever o funcionamento do algoritmo de ordenação *shell sort*

INTRODUÇÃO

No mundo da computação, as operações essenciais, e muito analisadas, são a ordenação e a pesquisa por determinado elemento em um conjunto de valores. Essas tarefas são usadas em muitos programas, além de compiladores, interpretadores, banco de dados e sistemas operacionais.

Neste tema, apresentaremos os fundamentos dos principais métodos de ordenação e analisaremos sua complexidade.

MÓDULO 1

⊙ **Reconhecer os algoritmos de ordenação elementares**

PROCESSO DE ORDENAÇÃO

Ordenação corresponde ao método de organizar um conjunto de objetos em uma ordem (ascendente ou descendente).

É uma atividade relevante e fundamental na área de computação e tem como objetivo facilitar a recuperação dos itens do conjunto, como, por exemplo, a recuperação de nomes em uma lista telefônica.

Em outras palavras, o processo de ordenação nada mais é do que o ato de colocar um conjunto de dados em determinada ordem predefinida. Estando os elementos ordenados conforme o critério definido, a ordenação permitirá que o acesso aos dados seja feito de forma mais eficiente.

A ordenação de um conjunto de dados é feita utilizando como base uma chave chamada de **chave de ordenação**, que é o campo do item utilizado para comparação.

É por meio dele que sabemos se determinado elemento está à frente ou não de outros no conjunto de dados.

Para realizar a ordenação, podemos usar qualquer tipo de chave, desde que exista uma regra de ordenação bem definida. Existem vários tipos possíveis de ordenação.

Os mais comuns são:

Ordenação numérica

1, 2, 3, 4, 5;

Ordenação lexicográfica (ordem alfabética)

Ana, Bianca, Michele, Ricardo.

TIPOS DE ALGORITMOS DE ORDENAÇÃO

Um **algoritmo de ordenação** é aquele que coloca os elementos de dada sequência em certa ordem predefinida. Existem vários algoritmos para realizar a ordenação dos dados. Eles podem ser classificados como:

ALGORITMOS DE ORDENAÇÃO INTERNA (*IN-PLACE*)

O conjunto de dados é pequeno e cabe todo na memória principal. Todo o processo de ordenação é realizado internamente na memória principal. Os dados a serem ordenados estão em um vetor ou em uma tabela, quando a ordenação é feita por um campo chamado de **chave**, que identifica cada elemento do vetor que forma a tabela.

ALGORITMOS DE ORDENAÇÃO EXTERNA

O conjunto de dados não cabe completamente em memória principal. Os dados a serem ordenados estão em algum disco de armazenamento ou fita, em arquivos no formato de registros, que são acessados sequencialmente ou em blocos.

Mais especificamente, há diversos tipos de algoritmos de ordenação, entre os quais podemos destacar:

Métodos	Ordenação por inserção	Ordenação por troca	Ordenação por seleção

Tipos de algoritmos de ordenação.

ORDENAÇÃO POR INSERÇÃO

Inserção direta (*insertion sort*);

Incrementos decrescentes.

ORDENAÇÃO POR TROCA

Bolha (*bubble sort*);

Troca e partição (*quick sort*).

ORDENAÇÃO POR SELEÇÃO

Seleção direta (*selection sort*);

Seleção em árvore (*heap sort*).

A seguir, vamos abordar alguns dos principais algoritmos básicos de ordenação de dados armazenados em *arrays* (ordenação interna).

BUBBLE SORT

Também conhecido como ordenação por “bolhas”, o *bubble sort* é um dos algoritmos de ordenação mais conhecidos e um dos mais simples.

O algoritmo *bubble sort* funciona da seguinte forma:

ETAPA 01

Em cada etapa, cada elemento é comparado com o próximo, e haverá uma troca se o elemento não estiver na ordem correta.

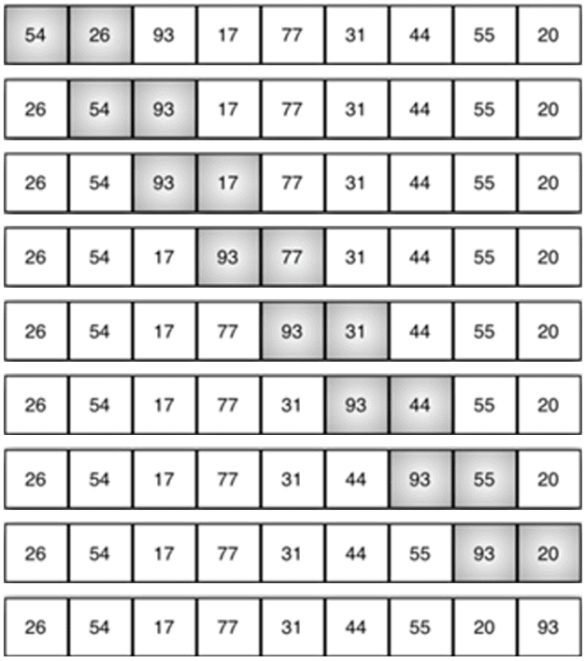
ETAPA 02

A comparação é novamente realizada até que as trocas não sejam mais necessárias.

Para entendermos melhor como utilizar esse algoritmo, vamos apresentar um exemplo.

I. Primeiro, vamos colocar os elementos representados em um vetor, com Item[1] mais à esquerda e Item[n] mais à direita.

II. Em cada passo, o maior elemento é deslocado para a direita até encontrar um elemento maior ainda, conforme mostra a figura ao lado:



Autor: Edgar Gurgel

📷 Funcionamento *bubble sort* – primeira passagem.

Aqui, é importante observar que o maior valor na lista está em alguma comparação.

> Esse valor será continuamente empurrado até o fim da passagem.
>
> Inicialmente, o valor 54 é empurrado.
>
> Depois, o valor 93 é empurrado até o final, e o valores menores vão subindo como bolhas.
>
> Os itens sombreados são aqueles que estão sendo comparados para verificar se estão fora de ordem.

Após o teste com o primeiro elemento do vetor, deve-se repetir o algoritmo com o os elementos subsequentes (Item[2], item[3],... Item[n].

Se imaginarmos que o vetor esteja na posição vertical, com o item[1] na parte inferior e o item[n] na superior, a cada comparação o maior elemento vai “subir”, semelhante ao que ocorreria com uma bolha em um tudo com líquido.

O código em C do algoritmo é o seguinte:

```
void bolha (int *v)
{
    int troca=1;
    int i=0;
    int aux;
    while (troca)
```

```
    {
        troca = 0;
        while (i < TAMANHO - 1)
        {
            if (v[i] > v[i+1])
            {
                aux = v[i];
                v[i] = v[i+1];
                v[i+1] = aux;
                troca = 1;
            }
            i++;
        }
        i = 0;
    }
}
```

Método da bolha em C.

COMENTÁRIO

O esforço computacional despendido pela ordenação de um vetor pode ser determinado pelo número de comparações, que também serve para estimar o número máximo de trocas possíveis de se realizar.

Na primeira passada, fazemos **n-1** comparações; na segunda, **n-2**; na terceira, **n-3**; e assim por diante.

Logo, o tempo total gasto pelo algoritmo é proporcional a:

(n-1) + (n-2) + … + 2 + 1.

A soma desses termos é proporcional ao quadrado de n. Portanto, o desempenho computacional desse algoritmo varia de forma quadrática em relação ao tamanho do problema.

Em geral, usamos a notação **Big-O** para expressar como a complexidade de um algoritmo varia com o tamanho do problema.

Assim, nesse caso, em que o tempo computacional varia de forma quadrática com o tamanho do problema, trata-se de um algoritmo de **ordem quadrática**, e expressamos isso escrevendo **$O(n^2)$**.

ATENÇÃO

No melhor caso, quando o vetor fornecido estiver quase ordenado, o procedimento poderá ser capaz de ordenar em uma única passada. No entanto, esse fato não pode ser usado para fazer uma análise de desempenho do algoritmo, pois o melhor caso representa uma situação muito particular.

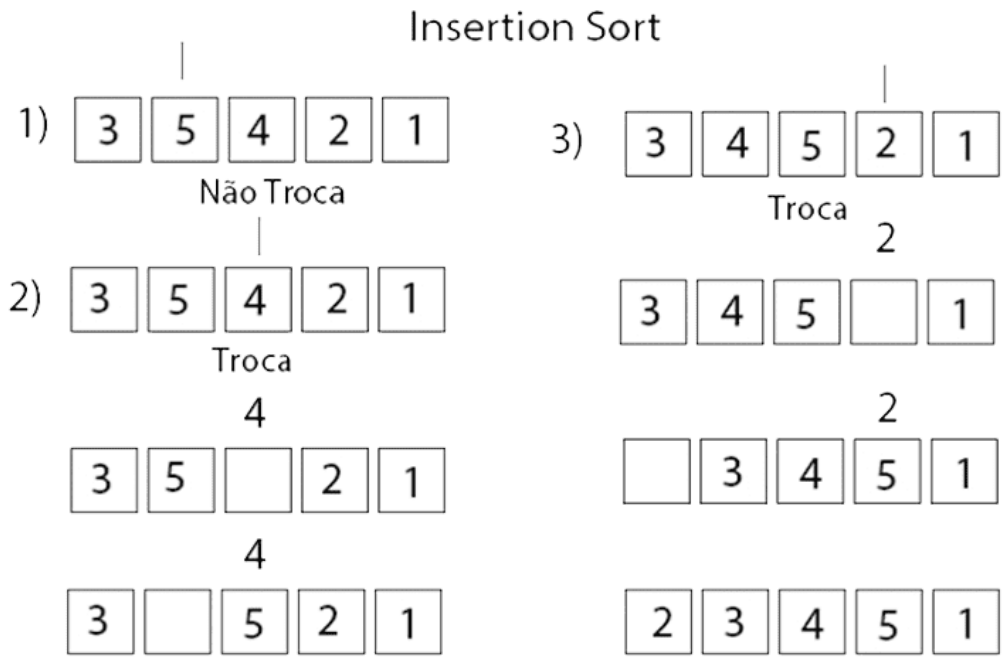
INSERTION SORT (INSERÇÃO)

Também conhecido como **ordenação por inserção**, o *insertion sort* é outro algoritmo de ordenação bastante simples.

Ele tem esse nome, pois se assemelha ao processo de ordenação de um conjunto de cartas de baralho com as mãos: pega uma carta de cada vez e a insere em seu devido lugar, sempre deixando as cartas da mão em ordem.

COMENTÁRIO

Na prática, esse algoritmo possui um desempenho superior quando comparado com outros algoritmos, como o *bubble sort* e o *selection sort*.



Autor: Edgar Gurgel

Funcionamento insertion sort.

O algoritmo insertion sort funciona da seguinte forma:

1. Os elementos são divididos em uma sequência de destino (a_1, ..., a_{i-1}) e em uma sequência fonte (a_i, ..., a_n).

2. Em cada passo, a partir de i = 2, o i-ésimo item da sequência fonte é retirado e transferido para a sequência destino, quando é inserido na posição adequada.

Veja que o algoritmo *insertion sort* percorre uma *array* e, para cada posição X, verifica se seu valor está na posição correta.

Isso é feito andando para o começo do *array*, a partir da posição X, e movimentando uma posição para frente os valores que são maiores do que o valor da posição X. Desse modo, teremos uma posição livre para inserir o valor da posição X em seu devido lugar.

As vantagens de utilização desse algoritmo são:

I. O número mínimo de comparações e movimentos ocorre quando os itens já estão originalmente ordenados.

II. O número máximo ocorre quando os itens estão originalmente em ordem reversa, o que indica um comportamento natural para o algoritmo.

O código que implementa o algoritmo de inserção é o seguinte:

```
void insertion (int *v)
{
     int i, j, aux;
     for (i=0; iv[j] && j > 0)
          {
               aux = v[j-1];
               v[j-1]=v[j];
               v[j]=aux;
               j--;
          }
     }
}
```

📷 Algoritmo de ordenação por inserção.

Ao contrário da ordenação bolha e da ordenação por seleção, o número de comparações que ocorrem durante a ordenação por inserção depende de como a lista está inicialmente ordenada. Se a lista estiver em ordem, o número de comparações será n-1. Se estiver fora de ordem, o número de comparações será $1/2(n^2 + n)$.

O número de troca para cada caso é o seguinte:

Melhor = 2 (n – 1);

Médio = $1/4(n^2 – n)$;

Pior = $1/2(n^2 + n)$.

Portanto, para o pior caso, a ordenação por inserção é tão ruim quanto a ordenação bolha e a ordenação por seleção, ou seja, sua complexidade é $O(n_2)$. Para o caso médio, é somente um pouco melhor.

No entanto, a ordenação por inserção tem duas vantagens:

PRIMEIRA VANTAGEM

Ela se comporta naturalmente, isto é, trabalha menos, quando a matriz já está ordenada, e o máximo, quando a matriz está ordenada no sentido inverso. Isso torna a ordenação excelente para listas que estão quase em ordem.

SEGUNDA VANTAGEM

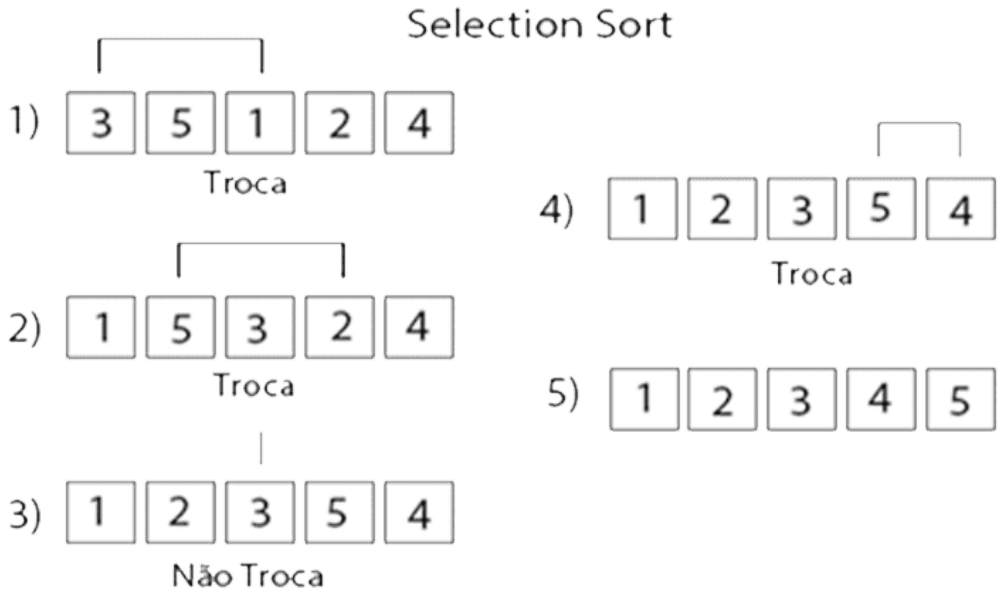
Ela não rearranja elementos de mesma chave. Isso significa que uma lista que é ordenada por duas chaves permanece ordenada para ambas as chaves após uma ordenação por inserção.

SELECTION SORT (SELEÇÃO)

Também conhecido como ordenação por seleção, o *selection sort* é outro algoritmo de ordenação bastante simples. Ele tem esse nome, pois, a cada passo, seleciona o melhor elemento (maior ou menor, dependendo do tipo de ordenação) para ocupar aquela posição do *array*.

COMENTÁRIO

Na prática, esse algoritmo possui um desempenho quase sempre superior quando comparado com o *bubble sort*.



Autor: Edgar Gurgel

📷 Funcionamento *selection sort*.

O algoritmo *selection sort* funciona da seguinte forma:

1. Selecione o elemento com a chave de **menor** valor.

2. Troque-o com o primeiro elemento da sequência.

3. Repita essas operações com os n-1 elementos restantes; depois, com os n-2 elementos; e assim sucessivamente, até restar um só elemento (o maior deles), conforme mostra a figura a seguir:

O algoritmo divide o *array* em duas partes:

A parte ordenada, à esquerda do elemento analisado.

A parte que ainda não foi ordenada, à direita do elemento.

Para cada elemento do *array*, começando do primeiro, o algoritmo procura na parte não ordenada (direita) o menor valor (ordenação crescente) e troca os dois valores de lugar.

Em seguida, o algoritmo avança para a próxima posição do *array*. Esse processo é feito até que todo o *array* esteja ordenado.

O código que implementa o algoritmo de seleção é o seguinte:

```
void selecao (int *v)
{
    int i,j,aux, minimo, pos_minimo;
    for (i=0; i < TAMANHO-1; i++)
```

```
    {
        minimo = v[i];
        pos_minimo = i;
        for (j=i+1; j < TAMANHO; j++) // Passo 1
        {
            if (minimo > v[j])
            {
                minimo = v[j];
                pos_minimo = j;
            }
        }
        if (pos_minimo != i)   // Passo 2
        {
            aux = v[pos_minimo];
            v[pos_minimo] = v[i];
            v[i] = aux;
        }
    }
}
```

📷 Método da seleção – implementação estável.

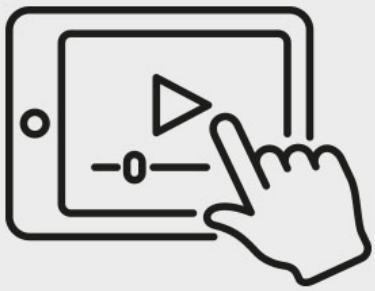
O tempo de execução do *selection sort* é $O(n^2)$ para o melhor e o pior caso.

Independentemente do vetor de entrada, o algoritmo se comportará da mesma maneira. Lembre-se de que a notação indica que o tempo de execução do algoritmo é limitado superior e inferiormente pela função n^2. Ele é ineficiente para grandes conjuntos de dados.

A seguir, será apresentado os três algoritmos de ordenação elementares, demonstrando a complexidade (notação O) de cada um deles.

A COMPLEXIDADE DOS ALGORITMOS DE ORDENAÇÃO ELEMENTARES

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. ALGORITMOS COMO *BUBBLE SORT* E *SELECTION SORT* TÊM COMO FINALIDADE:

A) A verificação da integridade de vetores.

B) O armazenamento de vetores.

C) A ordenação de vetores.

D) A recuperação de vetores.

E) A exclusão de vetores.

2. O ALGORITMO A SEGUIR, DESCRITO EM PSEUDOCÓDIGO, PODE SER UTILIZADO PARA ORDENAR UM VETOR V[1..N] EM ORDEM CRESCENTE:

```
Algoritmo (V, n)
1.   k = n-1
2.   para i = 1 até n faça
3.      j = 1
4.      enquanto j <= k faça
5.         se V[j] > V[j+1] então
6.            aux = V[j]
7.            V[j] = V[j+1]
8.            V[j+1] = aux
9.         j = j+1
10.     k = k-1
```

FONTE: SHUTTERSTOCK

ESSE ALGORITMO É CONHECIDO COMO:

A) *Insertion sort*

B) *Selection sort*

C) *Merge sort*

D) *Quick sort*

E) *Bubble sort*

GABARITO

1. Algoritmos como *bubble sort* e *selection sort* têm como finalidade:

A alternativa **"C "** está correta.

Os dois algoritmos (*bubble sort* e *selection sort*) são voltados para a ordenação de estruturas de vetores, de acordo com o critério que foi definido. As duas formas mais comuns são a ordenação lexicográfica e a numérica.

2. O algoritmo a seguir, descrito em pseudocódigo, pode ser utilizado para ordenar um vetor V[1..n] em ordem crescente:

```
Algoritmo (V, n)
1.  k = n-1
2.  para i = 1 até n faça
3.      j = 1
4.      enquanto j <= k faça
5.          se V[j] > V[j+1] então
6.              aux = V[j]
7.              V[j] = V[j+1]
8.              V[j+1] = aux
9.          j = j+1
10.     k = k-1
```

Fonte: Shutterstock

Esse algoritmo é conhecido como:

A alternativa **"E "** está correta.

O *bubble sort* realiza múltiplas passagens por uma lista. Ele compara itens adjacentes e troca aqueles que estão fora de ordem. Cada passagem pela lista coloca o próximo maior valor em sua posição correta. Em essência, cada item se desloca como uma “bolha” para a posição à qual pertence. Se existem n itens na lista, então, há $n-1$ pares de itens que precisam ser comparados na primeira passagem.

MÓDULO 2

- **Descrever o funcionamento do algoritmo de ordenação por intercalação (*merge sort*)**

ALGORITMO DE ORDENAÇÃO POR INTERCALAÇÃO (*MERGE SORT*)

Também conhecido como ordenação por intercalação, o ***merge sort*** é um algoritmo recursivo que usa a ideia de dividir para conquistar, a fim de ordenar os dados de um array.

Esse algoritmo parte do princípio de que é mais fácil ordenar um conjunto com poucos dados do que um conjunto com muitos dados.

Dessa forma, o algoritmo divide os dados em conjuntos cada vez menores para, depois, ordená-los e combiná-los por meio de intercalação (*merge*).

CARACTERÍSTICAS

Nesta forma de ordenação, o vetor é dividido em vetores com a metade do tamanho do original por meio de um procedimento recursivo. A divisão ocorre até que o vetor fique com somente um elemento, e estes estejam ordenados e intercalados.

A técnica recursiva de dividir para conquistar, aplicada ao algoritmo de *merge sort*, é formada por três etapas:

DIVIDIR

1. **Dividir** o problema em certo número de subproblemas – dividir a sequência de n elementos a serem ordenados em duas subsequências de n/2 elementos;

CONQUISTAR

2. **Conquistar** os subproblemas, solucionando-os recursivamente (se os tamanhos dos subproblemas são suficientemente pequenos, então, solucionar os subproblemas de forma simples) – ordenar duas subsequências recursivamente, utilizando a ordenação por intercalação;

COMBINAR

3. **Combinar** as soluções dos subproblemas na solução de problema original – intercalar as duas subsequências ordenadas por intercalação.

Esse algoritmo apresenta algumas vantagens:

- Requer menos acesso à memória;

É ótimo candidato para o emprego da programação paralela – caso exista mais de um processador, certamente, ele terá um bom desempenho.

EXECUÇÃO

A execução do algoritmo *merge sort* pode ser representada com o uso de uma árvore binária da seguinte forma:

Cada nó representa uma chamada recursiva do *merge sort*.

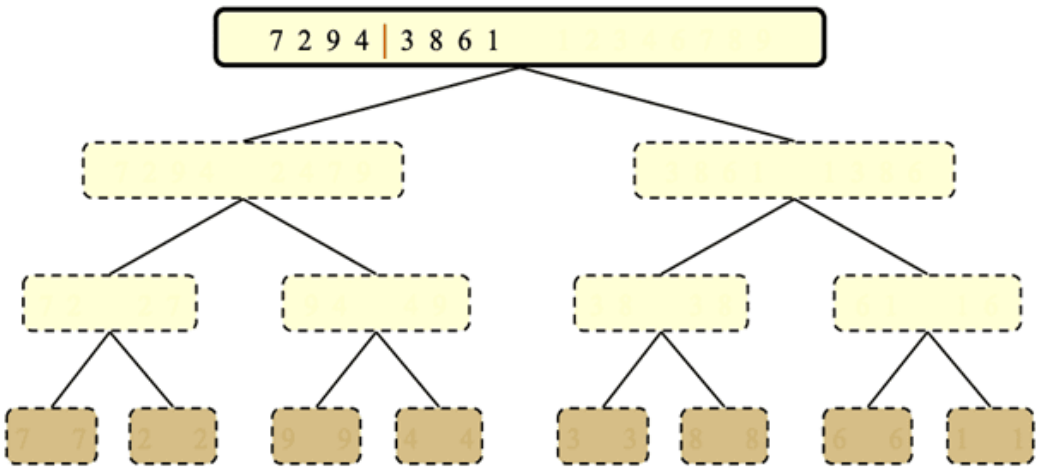
O nó raiz é a chamada inicial.

Os nós folhas são vetores de 1 ou 2 números, que são os casos base.

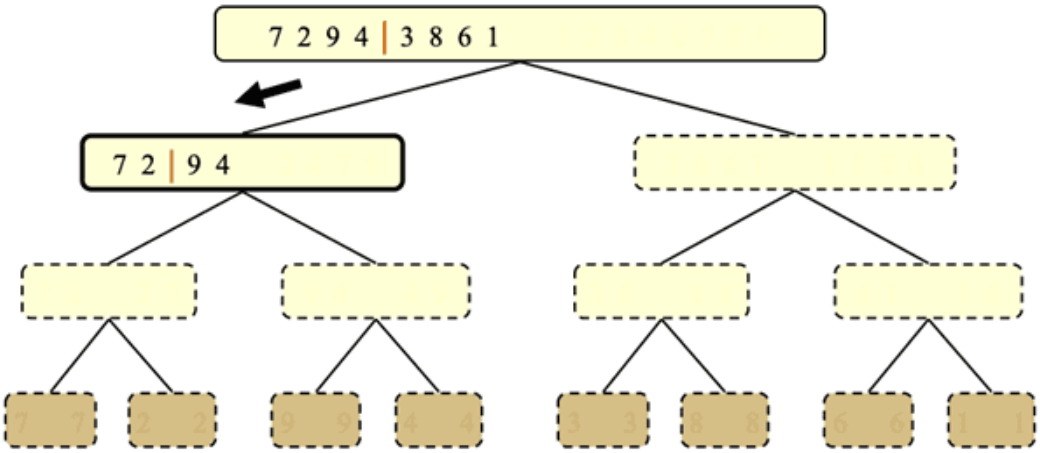
Agora, vamos mostrar um exemplo de execução do merge sort para ordenar o seguinte vetor:

7	2	9	4	3	8	6	1

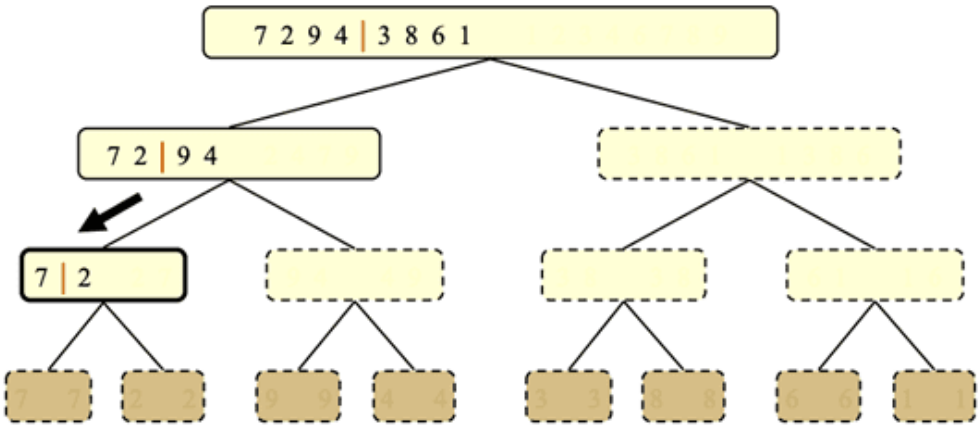
O primeiro passo é a partição do problema, sempre no meio do vetor. Dessa forma, o vetor inicial é particionado em duas partes (7 2 9 4 e 3 8 6 1), conforme mostra a figura a seguir:



Em seguida, vamos fazer uma chamada recursiva para a primeira partição do vetor 7 2 9 4, que será particionada em duas partes (7 2 e 9 4):

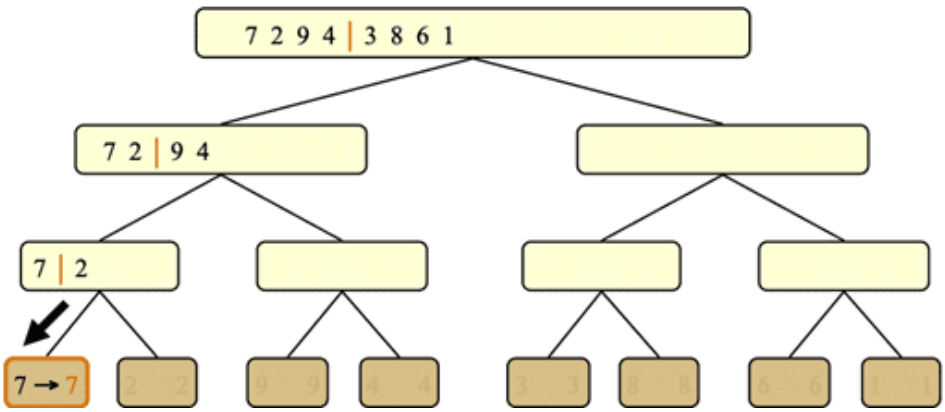


Uma nova chamada recursiva é feita para essa primeira partição 7 2, de tal forma que vamos particioná-la em duas partes (7 e 2) por meio de uma chamada recursiva, conforme mostra a figura a seguir:

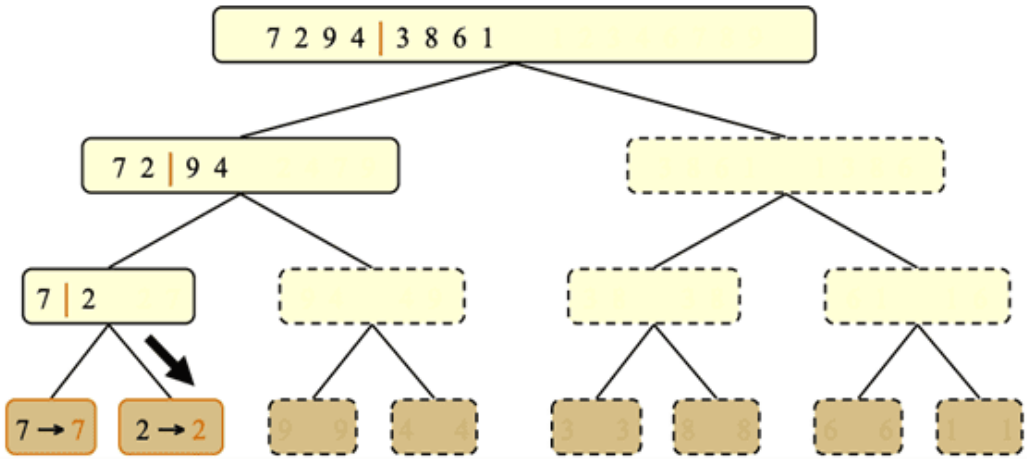


Em seguida, faremos duas chamadas recursivas para o caso base encontrado (7 e 2).

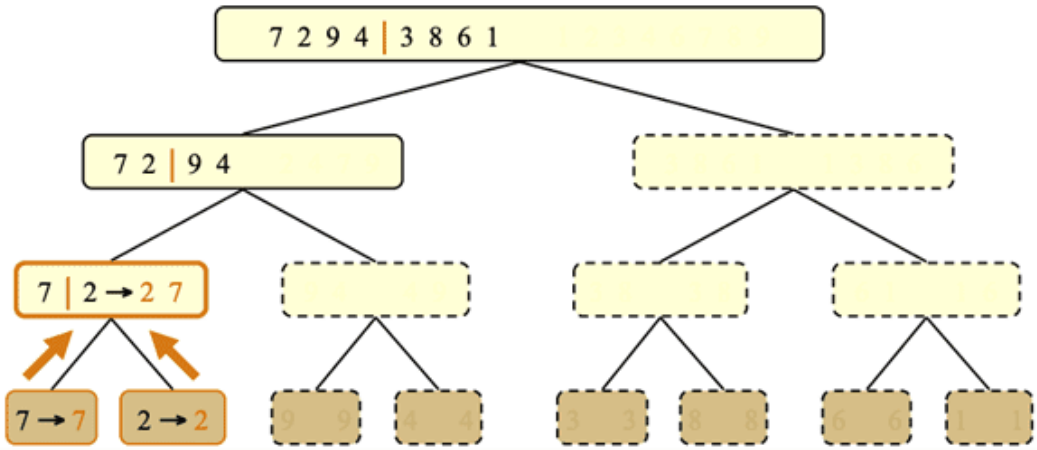
A primeira chamada recursiva é representada na figura a seguir:



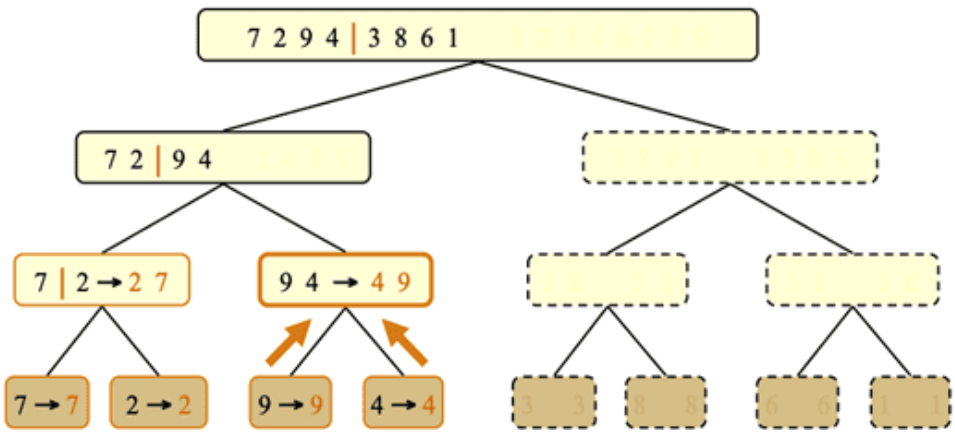
A segunda chamada recursiva para o caso base (7 e 2) é representada na figura a seguir:



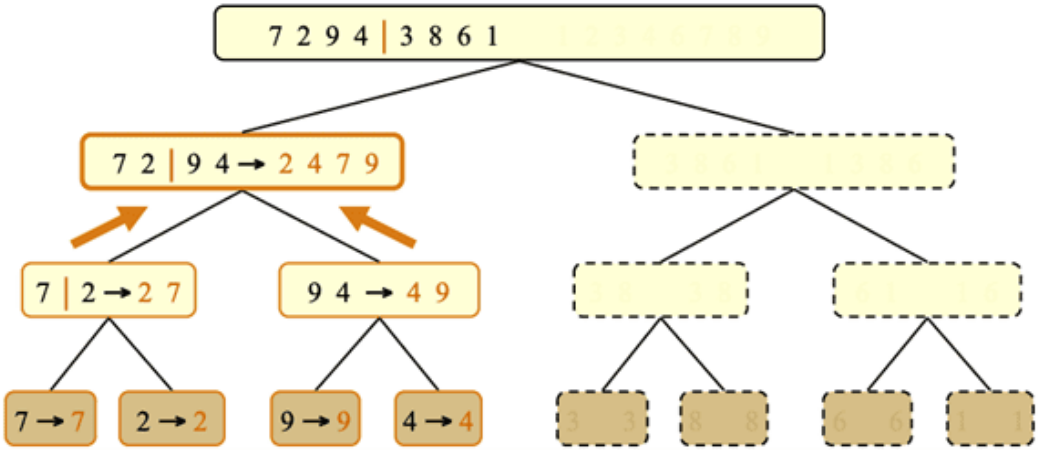
Em seguida, é realizada uma operação de intercalação para ordenar o caso base (7 e 2), conforme mostra a figura a seguir:



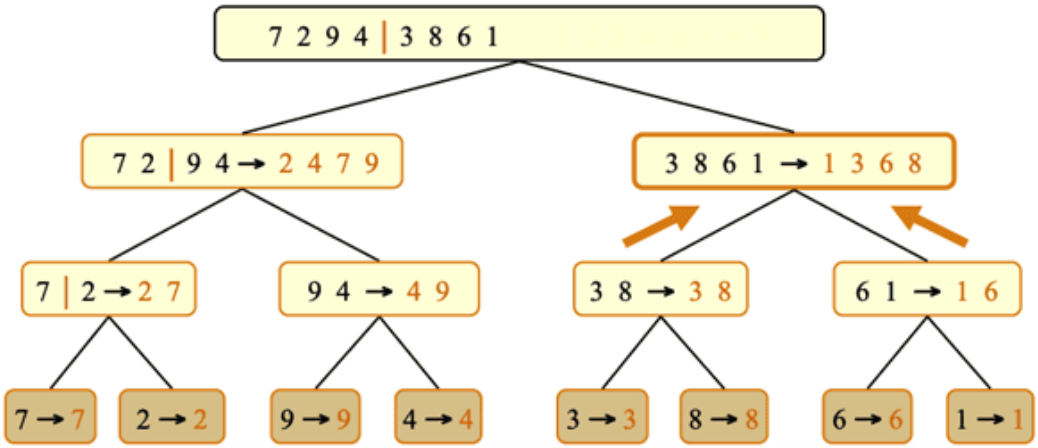
A figura a seguir já mostra as chamadas recursivas para a partição 9 4, particionando-a em duas partes (9 e 4); depois, as duas chamadas recursivas para o caso base; e, por fim, a intercalação para o caso base 9 e 4:



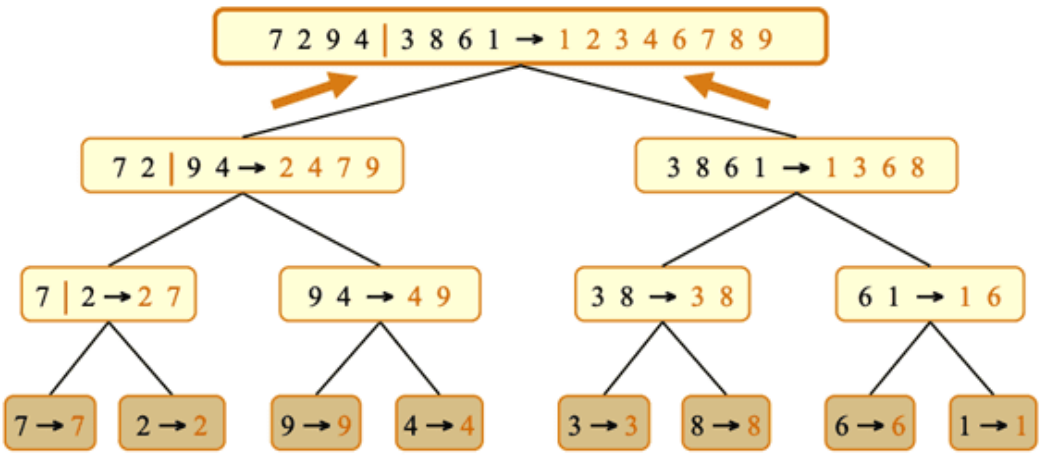
Agora, será realizada uma intercalação (*merge*) para ordenar a primeira partição do vetor:



Em seguida, vamos executar os mesmos procedimentos para ordenar a segunda partição do vetor, que ficará da seguinte forma, conforme mostra a figura a seguir:



Por fim, o último merge servirá para ordenar as duas partições do vetor. Dessa forma, teremos as duas partições (2 4 7 9 e 1 3 6 8) ordenadas, conforme mostra a figura a seguir:



IMPLEMENTAÇÃO

O algoritmo *merge sort* será implementado pelo programa a seguir e por suas funções. Primeiro, vamos apresentar o programa principal:

```
#include <stdio.h>
#include <stdlib.h>
// funções
void mergesort(int *v, int n);
void sort(int *v, int *c, int i, int f);
void merge(int *v, int *c, int i, int m, int f);
// programa principal
int main (void) {
   int i;
   int v[8] = { -1, 7, -3, 11, 4, -2, 4, 8 };
   mergesort(v, 8);
   for (i = 0; i < 8; i++)
printf("%d ", v[i]);
   putchar('\n');
   return 0;
}
```

Agora, vamos apresentar suas funções.

Dado um vetor de inteiros v e um inteiro n >= 0, ordena o vetor v[0..n-1] em ordem crescente:

```
void mergesort(int *v, int n) {
      int *c = malloc(sizeof(int) * n);
      sort(v, c, 0, n - 1);
      free(c);
   }
```

Dado um vetor de inteiros v e dois inteiros i e f, ordena o vetor v[i..f] em ordem crescente. O vetor c é utilizado internamente durante a ordenação:

```
void sort(int *v, int *c, int i, int f) {
      if (i >= f)
      return;
      int m = (i + f) / 2;
      sort(v, c, i, m);
      sort(v, c, m + 1, f);
      /* Se v[m] <= v[m + 1], então v[i..f] já está ordenado. */
      if (v[m] <= v[m + 1])
      return;
merge(v, c, i, m, f);
}
```

Dado um vetor v e três inteiros i, m e f, sendo v[i..m] e v[m+1..f] vetores ordenados, coloca os elementos desses vetores, em ordem crescente, no vetor em v[i..f]:

```
void merge(int *v, int *c, int i, int m, int f) {
    int z, iv = i, ic = m + 1;
    for (z = i; z <= f; z++)
  c[z] = v[z];
    z = i;
    while (iv <= m && ic <= f) {
      /* Invariante: v[i..z] possui os valores de v[iv..m] e v[ic..f] em ordem crescente. */
      if (c[iv] < c[ic])
  v[z++] = c[iv++];
      else  /* if (c[iv] > c[ic]) */
  v[z++] = c[ic++];
    }
    while (iv <= m)
  v[z++] = c[iv++];
    while (ic <= f)
  v[z++] = c[ic++];
  }
```

ANÁLISE DE COMPLEXIDADE

O trecho deste pseudocódigo relevante do algoritmo *merge sort* é o seguinte:

```
função merge (x, inicio, fim)
início
var
meio: numérico;
Se (inicio < fim)
então
inicio
meio ← parteinteira (incio + fim) / 2;
merge (x, inicio, meio);
merge (x, meio+1, fim);
intercala (x, inicio, fim, meio);
fim;
fim-se;
fim-função-merge.
```

Para calcular o tempo de execução do *merge sort*, é preciso calcular, inicialmente, o tempo de execução da função intercala.

O algoritmo da função intercala, que realiza a intercalação de dois vetores, cujos tamanhos suponhamos que sejam $\mathbf{m_1}$ e $\mathbf{m_2}$, respectivamente, faz a varredura de todas as posições dos dois vetores, gastando, com isso, tempo $n = m_1 + m_2$.

Analisando o trecho de código anterior, verificamos que a função **merge** possui três chamadas de função.

As duas primeiras são chamadas recursivas e recebem metade dos elementos do vetor passado, e a outra é a chamada para função que realiza a intercalação das duas metades.

A linha de comparação Se e a linha da atribuição gastam tempo constante O(1). Para calcularmos o tempo de execução de um programa recursivo, inicialmente, deveremos obter a expressão de recorrência da função *merge sort*, que é dada por:

$$T(N)$$

Atenção! Para visualização completa da equação utilize a rolagem horizontal

Onde:

2T (n/2) = duas chamadas recursivas;

n = tempo gasto com a intercalação das duas metades do vetor.

O tempo gasto nas demais linhas da função **(O(1))**, ou seja, o tempo constante, é menor que o gasto pela função intercala. Por isso, somamos apenas **n** à expressão de recorrência.

A recorrência obtida anteriormente será solucionada pelo método master, mostrado a seguir. Os valores necessários para a resolução por esse método são: a = b = 2, f(n) = n.

$$f(n) = \theta\ (n^{\log_b\ a})$$

$$n = \theta\ (n^{\log_2\ 2})$$

$$n = \theta\ (n^1)$$

Atenção! Para visualização completa da equação utilize a rolagem horizontal

Então, sua complexidade é a seguinte:

$$T(n) = \theta (n \log n)$$

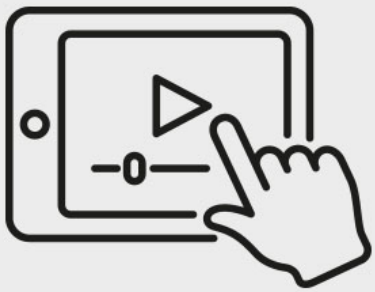
Atenção! Para visualização completa da equação utilize a rolagem horizontal

No *merge sort*, qualquer que seja o vetor de entrada, o algoritmo trabalhará da mesma maneira, dividindo o vetor ao meio, ordenando cada metade recursivamente e intercalando as duas metades ordenadas.

A seguir, será apresentado com exemplos práticos, o algoritmo de ordenação *merge sort*, demonstrando a análise de complexidade.

ALGORITMO DE ORDENAÇÃO MERGE SORT

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. AS ESTRATÉGIAS DE DIVISÃO E CONQUISTA SÃO UTILIZADAS PELO SEGUINTE ALGORITMO DE ORDENAÇÃO:

A) *Selection sort*

B) *Insertion sort*

C) *Bubble sort*

D) *Shell sort*

E) *Merge sort*

2. UMA FÁBRICA DE *SOFTWARE* FOI CONTRATADA PARA DESENVOLVER UM PRODUTO DE ANÁLISE DE RISCOS. EM DETERMINADA FUNCIONALIDADE DESSE *SOFTWARE*, É NECESSÁRIO REALIZAR A ORDENAÇÃO DE UM CONJUNTO FORMADO POR MUITOS NÚMEROS INTEIROS. QUE ALGORITMO DE ORDENAÇÃO OFERECE MELHOR COMPLEXIDADE DE TEMPO (NOTAÇÃO O) NO PIOR CASO?

A) *Merge sort*

B) *Insertion sort*

C) *Bubble sort*

D) *Half sort*

E) *Selection sort*

GABARITO

1. As estratégias de divisão e conquista são utilizadas pelo seguinte algoritmo de ordenação:

A alternativa **"E "** está correta.

A ideia básica do *merge sort* consiste em dividir o problema em vários subproblemas e resolver esses subproblemas por meio da recursividade, bem como conquistar, ou seja, unir as resoluções dos subproblemas após todos terem sido resolvidos.

2. Uma fábrica de *software* foi contratada para desenvolver um produto de análise de riscos. Em determinada funcionalidade desse *software*, é necessário realizar a ordenação de um conjunto formado por muitos números inteiros. Que algoritmo de ordenação oferece melhor complexidade de tempo (notação O) no pior caso?

A alternativa **"A "** está correta.

Para o *merge sort*, não importa a maneira como os elementos estão organizados no vetor. Sua complexidade será sempre a mesma. Este algoritmo é ideal para aplicações que precisam de ordenação eficiente, que não toleram desempenho ruim no pior caso e que possuem espaço de memória extradisponível.

MÓDULO 3

- **Descrever o funcionamento do algoritmo de ordenação rápida (*quick sort*)**

ALGORITMO DE ORDENAÇÃO RÁPIDA (*QUICK SORT*)

O algoritmo de ordenação *quick sort* foi proposto em 1960 pelo cientista da computação britânico Charles Antony Richard Hoare, mas só publicado em 1962.

O ***quick sort*** é o algoritmo de ordenação interna mais rápido e, provavelmente, o mais utilizado.

Ele se caracteriza por dividir o vetor em duas partes, utilizando um procedimento recursivo. Essa divisão se repetirá até que o vetor tenha apenas um elemento, enquanto os demais ficam ordenados ao passo que ocorre o particionamento.

É um algoritmo que também se baseia na técnica de divisão e conquista, que é aplicada da seguinte forma:

DIVIDIR

O vetor X[p..r] é particionado (rearranjado) em dois subvetores não vazios X[p..q] e X[q+1..r], tais que cada elemento de X[p..q] é menor ou igual a cada elemento de X[q+1..r].

O índice q é calculado como parte do processo de particionamento. Para encontrar esse índice, o elemento que está na metade do vetor original – chamado de pivô – é escolhido, e os elementos do vetor são reorganizados. Os que ficarem à esquerda de q são menores ou iguais ao pivô, e os que ficarem à direta de q são maiores ou iguais ao pivô.

CONQUISTAR

Chamadas recursivas ao quick sort são usadas para ordenar os dois subvetores: X[p..q] e X[q+1..r].

COMBINAR

Durante o procedimento recursivo, os elementos são ordenados no próprio vetor. Nesta fase, nenhum processamento é necessário.

De forma geral, para o particionamento, o algoritmo pode ser descrito da seguinte forma:

1

Escolha arbitrariamente um pivô **x**.

2

Percorra o vetor a partir da esquerda até que v[i] >= **x**.

3

Percorra o vetor a partir da direita até que v[j] <= **x**.

4

Troque **v[i] com v[j]**.

5

Continue esse processo até os apontadores **i** e **j** se cruzarem.

Ao final do procedimento, o vetor X[p..r] estará dividido em dois segmentos:

I. Os itens em X[p..q], X[p+1], …., X[i] são menores ou iguais a **x**.

II. Os itens em X[q+1..r], X[q+1], …., X[j] são maiores ou iguais a **x**.

MÉTODO PARA PARTICIONAMENTO

Considere o pivô como o elemento da posição final que queremos encontrar.

> ## DICA
>
> Utilizar o último elemento do vetor ou o primeiro elemento é só um mecanismo para facilitar a implementação.

Inicialize dois ponteiros, denominados alto e baixo, como o limite inferior e superior do vetor que será analisado. Em qualquer momento da execução, elementos **acima de alto** serão **maiores que x**, e elementos **abaixo de baixo** serão **menores que x**.

Vamos mover os ponteiros alto e baixo um em direção ao outro, de acordo com os seguintes passos:

ETAPA 01

01. Baixo deve ser incrementado em uma posição até que a[baixo] >= pivô;

ETAPA 02

02. Alto deve ser decrementado em uma posição até que a[alto] < pivô.

ETAPA 03

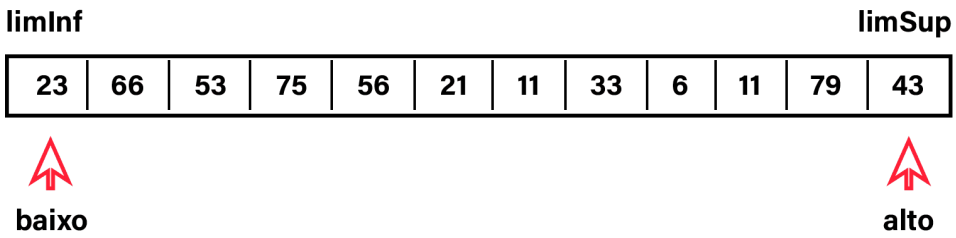
03. Se alto > baixo, devemos trocar as posições de a[baixo] por a[alto].

> ## ATENÇÃO
>
> Esse procedimento deve ser repetido até que a condição colocada no passo 3 falhe, ou seja, quando alto <= baixo. Assim, a[alto] será trocado pelo pivô, que é o objetivo da procura.

EXECUÇÃO

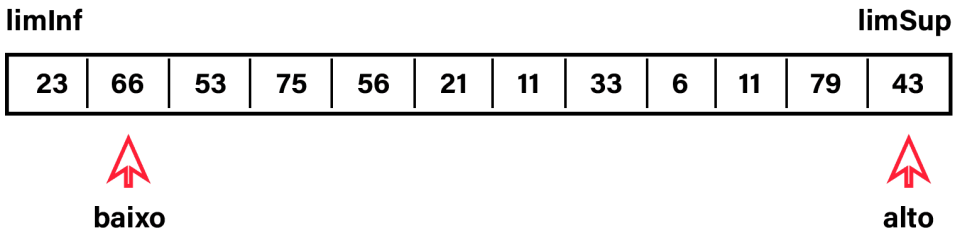
Para entendermos melhor o funcionamento do algoritmo *quick sort*, vamos simular a execução do vetor a seguir, considerando o limite superior e inferior, pivô, alto e baixo. O objetivo é ordenar esse vetor.

O limInf será escolhido como o pivô. Inicialmente, o vetor será dividido em duas partes: a parte da esquerda terá os elementos menores que o pivô, enquanto a parte da direita terá os elementos maiores que o pivô. Observe:



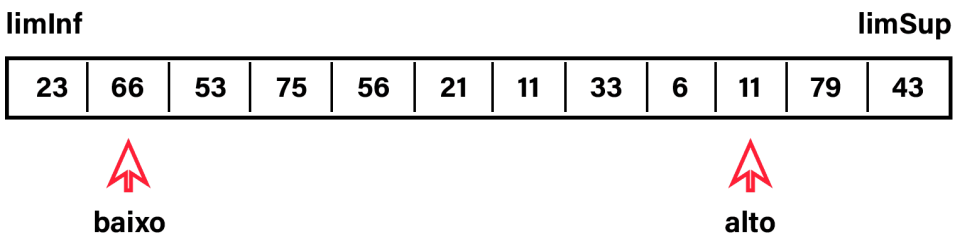
Autor: Gabriele Passeri

Vamos incrementar baixo até que seja encontrado um valor maior que o pivô escolhido 23. Então, encontramos o valor 66:



Autor: Gabriele Passeri

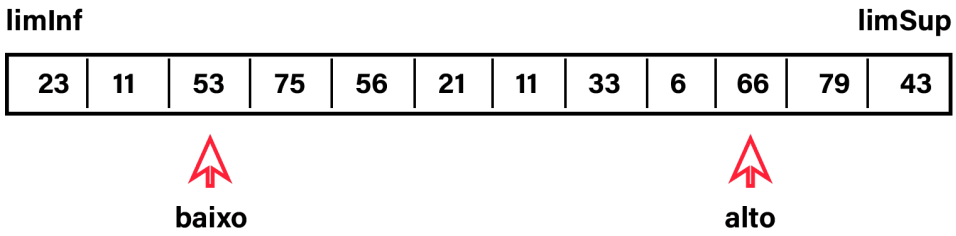
Agora, vamos decrementar alto até que seja encontrado um valor menor que o pivô escolhido 23. Então, encontramos o valor 11:



Autor: Gabriele Passeri

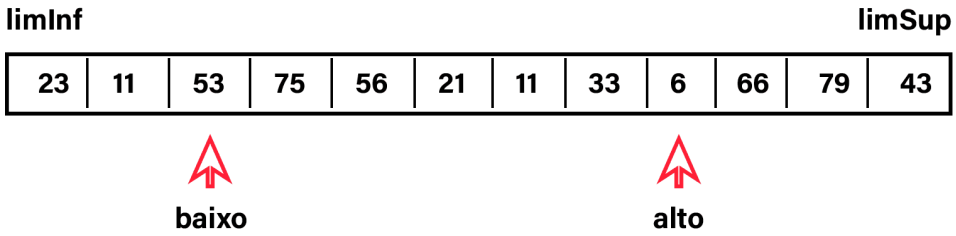
Em seguida, vamos trocar os valores de a[alto] e a[baixo], ou seja, 66 e 11. Após a troca desses valores, vamos incrementar baixo até que um valor maior que o pivô 23 seja encontrado. Então, encontramos o

valor 53:



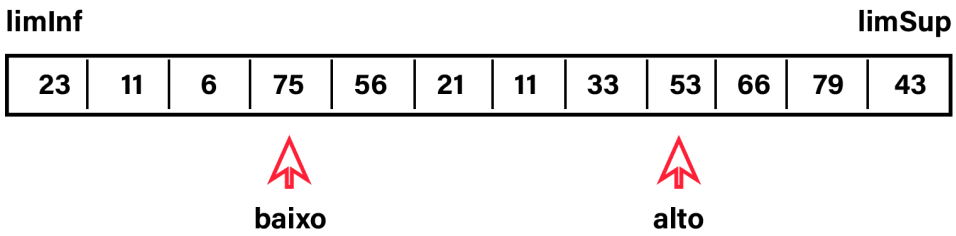
Autor: Gabriele Passeri

Agora, vamos decrementar alto até que um valor menor que o pivô 23 seja encontrado. Então, encontramos o valor 6:



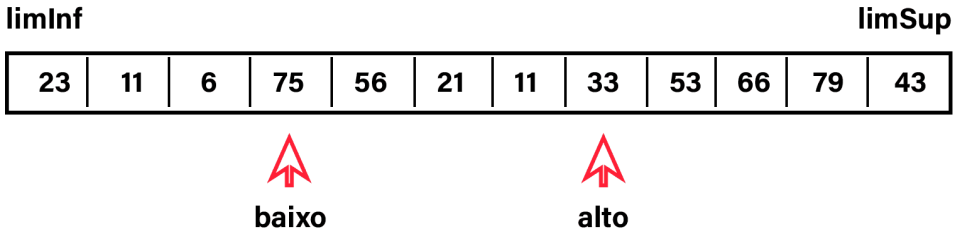
Autor: Gabriele Passeri

Em seguida, vamos trocar os valores de a[alto] e a[baixo], ou seja, 53 e 6. Após a troca desses valores, vamos incrementar baixo até que um valor maior ou igual ao pivô 23 seja encontrado. Então, encontramos o valor 75:



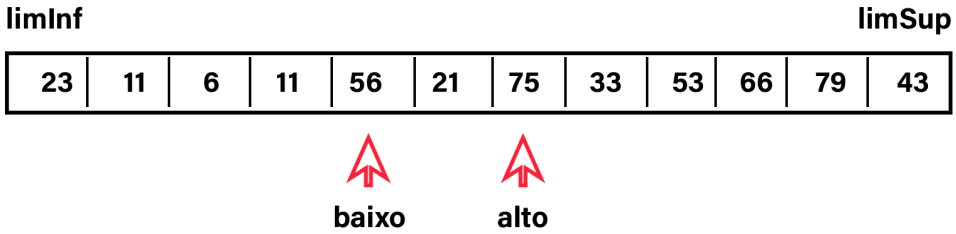
Autor: Gabriele Passeri

Agora, vamos decrementar alto até que um valor menor que o pivô 23 seja encontrado. Então, encontramos o valor 11:



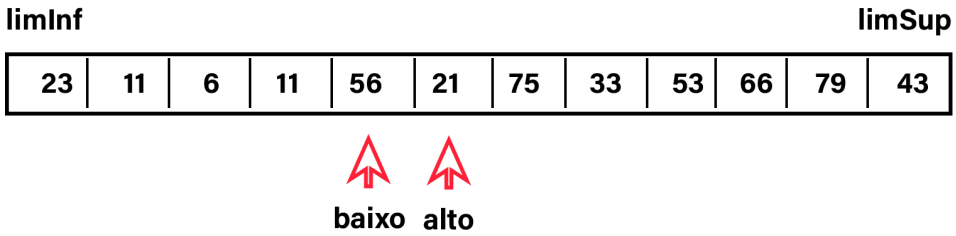
Autor: Gabriele Passeri

Em seguida, vamos trocar os valores de a[alto] e a[baixo], ou seja, 75 e 11. Após a troca desses valores, vamos incrementar baixo até que um valor maior ou igual ao pivô 23 seja encontrado. Então, encontramos o valor 56:



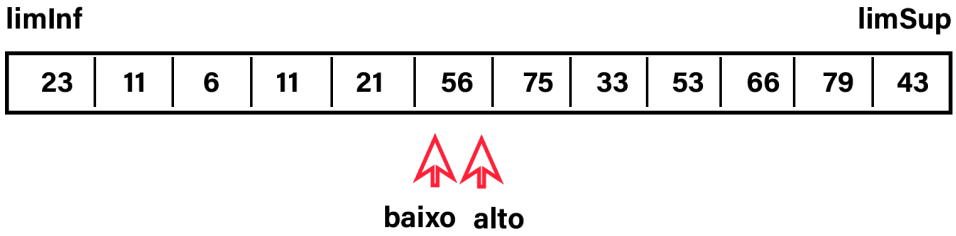
Autor: Gabriele Passeri

Agora, vamos decrementar alto até que um valor menor que o pivô 23 seja encontrado. Então, encontramos o valor 21:



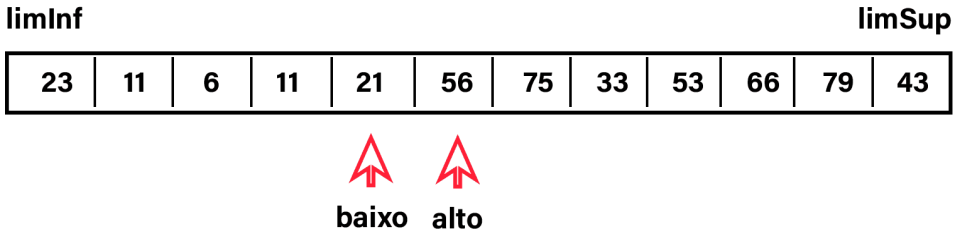
Autor: Gabriele Passeri

Em seguida, vamos trocar os valores de a[alto] e a[baixo], ou seja, 56 e 21. Após a troca desses valores, vamos incrementar baixo até que um valor maior ou igual ao pivô 23 seja encontrado. Então, encontramos o valor 56 novamente:



Autor: Gabriele Passeri

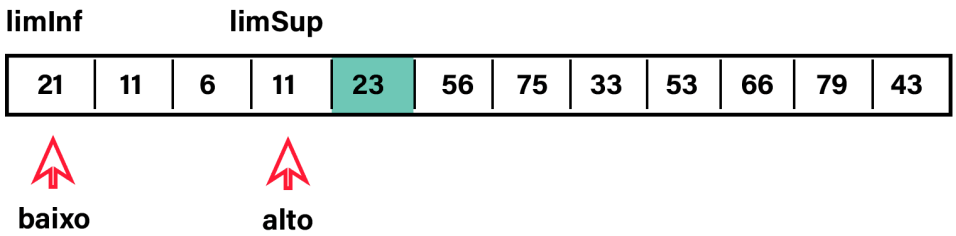
Agora, vamos decrementar alto até que um valor menor que o pivô 23 seja encontrado. Então, encontramos o valor 21 novamente. Entretanto, veja que atingimos a condição de parada do algoritmo, pois alto 21 é menor que baixo 56:



Autor: Gabriele Passeri

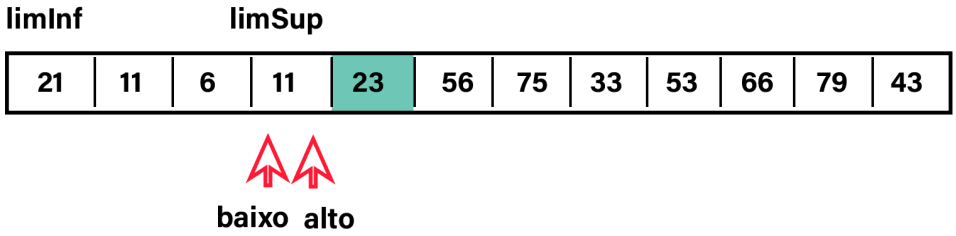
Em seguida, vamos trocar o pivô a[limInf]= 23 com a[alto] = 21. Assim, a divisão do vetor está concluída.

Agora, devemos particionar o subvetor esquerdo e reorganizar os elementos para esse subvetor da seguinte forma, onde limSup = alto – 1 = 4:



Autor: Gabriele Passeri

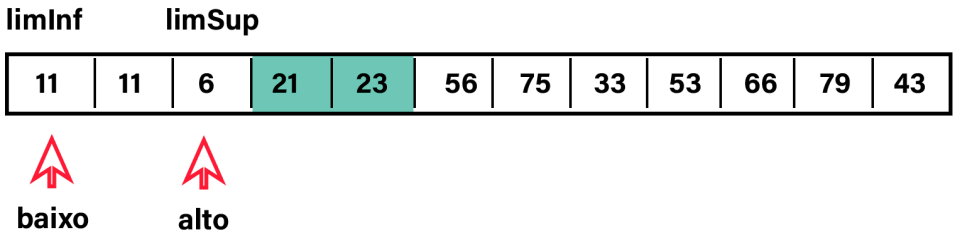
Em seguida, vamos incrementar baixo até que um valor maior ou igual ao pivô 21 seja encontrado. Então, encontramos o valor alto 11. E vamos decrementar alto até que um valor menor que o pivô 21 seja encontrado. Então, encontramos o valor baixo 11:



Autor: Gabriele Passeri

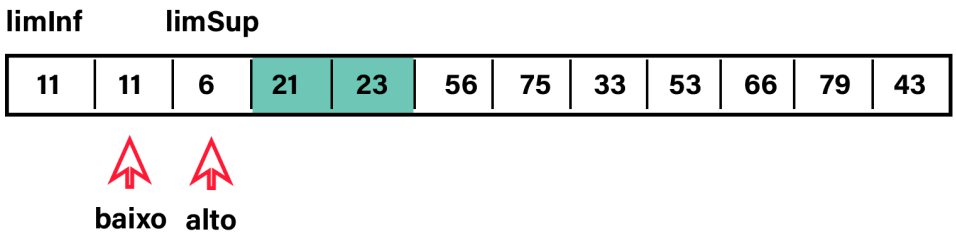
Depois, vamos trocar o pivô a[limInf]= 21 com a[alto] = 11. Assim, a divisão do vetor está concluída.

Agora, devemos particionar o subvetor esquerdo e reorganizar os elementos para esse subvetor da seguinte forma, onde limSup = alto – 1 = 3:



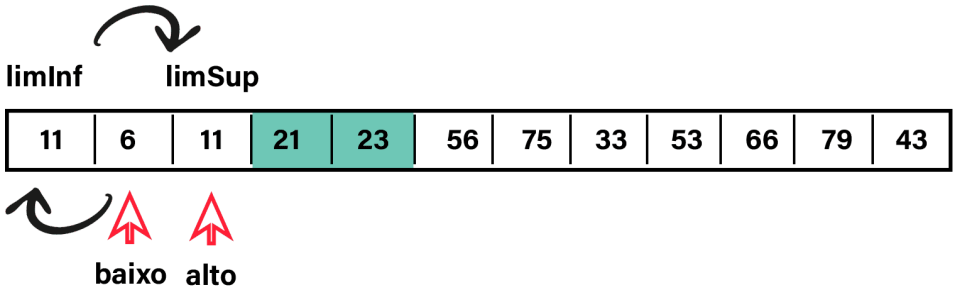
Autor: Gabriele Passeri

Em seguida, vamos incrementar baixo até que um valor maior ou igual ao pivô 11 seja encontrado. Então, encontramos o valor baixo 11. E vamos decrementar alto até que um valor menor que o pivô 11 seja encontrado. Então, encontramos o valor alto 6:



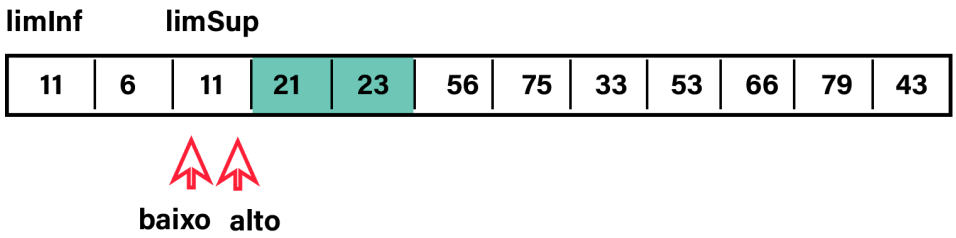
Autor: Gabriele Passeri

Depois, vamos trocar os valores de a[alto] e a[baixo], ou seja, 11 e 6:



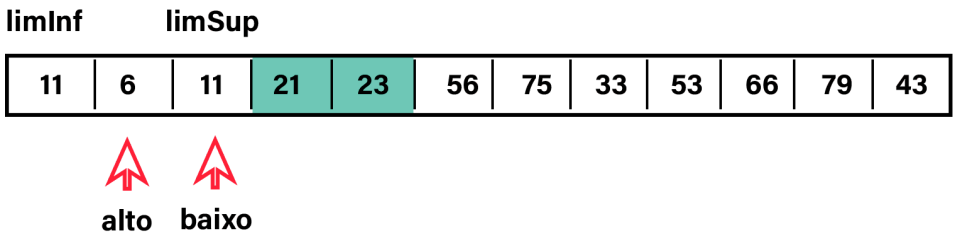
Autor: Gabriele Passeri

Após a troca dos valores, vamos incrementar baixo até que um valor maior ou igual ao pivô 11 seja encontrado. Então, encontramos o valor 11:



Autor: Gabriele Passeri

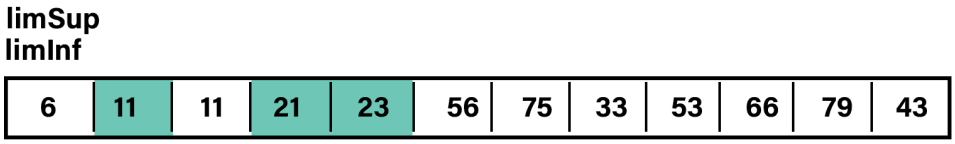
Agora, vamos decrementar alto até que um valor menor que o pivô 11 seja encontrado. Então, encontramos o valor 6. Entretanto, veja que atingimos a condição de parada do algoritmo, pois alto 6 é menor que baixo 11:



Autor: Gabriele Passeri

Em seguida, vamos trocar o pivô a[limInf]= 11 com a[alto] = 6. Assim, a divisão do vetor está concluída.

Agora, devemos particionar o subvetor esquerdo e reorganizar os elementos para esse subvetor da seguinte forma, onde limSup = alto – 1 = 1. Entretanto, como LimSup > LimInf é falso, então, devemos parar:



Autor: Gabriele Passeri

O mesmo procedimento deve ser executado para ordenar a segunda metade do vetor.

IMPLEMENTAÇÃO

Veja, a seguir, o algoritmo que implementa o quick sort, com o desenvolvimento do procedimento partição:

```
#include <stdio.h>

void lerVet( int *p, int t ){

    int i;

    for ( i=0; i<t; i++ ){

        printf("\tElemento da posicao %d? ",i);
        scanf("%d",p);
        p++;

    }

}

void mostrarVet( int *p, int t ){

    int i;

    for ( i=0; i < t; i++ ){

        printf("\tPosicao %d: %d\n",i,*p);
        p++;

    }

}
```

```
void trocar (int *pv, int x, int y ){

        int aux = pv[x];
        pv[x] =pv[y];
        pv[y] = aux;

}

int divide( int *v, int inf, int sup ) {

    int pivo, esq, dir;

    pivo = v[inf];
    esq = inf;
    dir = sup;

    while ( esq < dir ) {

        while ( v[esq] <= pivo  && esq < sup )
            esq++;

        while ( v[dir] > pivo )
            dir--;

        if ( esq < dir )
            trocar(v,esq,dir);

    }

    v[inf] = v[dir];
    v[dir] = pivo ;
    return dir;
}

void quickSort( int *p, int inf, int sup ) {

    int posPivo; // posição do pivô

    if ( inf >= sup )
        return;

    posPivo= divide(p,inf,sup);
    quickSort(p,inf,posPivo-1);
    quickSort(p,posPivo+1,sup);

}


void main(){
```

```
    int *p, tam;

    printf("Quantidade de elementos do vetor?");
    scanf("%d",&tam);

    p = (int*) malloc(tam * sizeof(int));

    printf("\nDigite o conteudo do vetor:\n");
    lerVet(p, tam);

    printf("\nConteudo digitado para o vetor:\n");
    mostrarVet(p, tam);

    printf("\nOrdenando o vetor...\n");
    quickSort(p, 0, tam-1);

    printf("\nConteudo do vetor ja ordenado:\n");
    mostrarVet(p, tam);

    free(p);

}
```

ANÁLISE DE COMPLEXIDADE

A ideia principal do algoritmo de ordenação *quick sort* é fazer o processo de particionar o vetor em duas partes, realizado pelo procedimento partição.

Nesse procedimento, o vetor é particionado na posição **j**, de forma que todos os elementos do lado esquerdo de **j** são menores ou iguais ao elemento denominado pivô, e todos os elementos do lado direito são maiores que o pivô.

Além disso, nesse procedimento, o tempo de execução é limitado pelo tamanho do vetor – no caso **n**. Isso acontece porque, para realizar esse particionamento, o algoritmo compara os elementos de posição **i**, cujo valor inicia na primeira posição e vai aumentando, e os elementos da posição **j**, cujo valor inicia com a última posição e decresce, com o valor pivô.

COMENTÁRIO

Em outros termos, o algoritmo compara todos os elementos do vetor com o pivô, enquanto os índices atenderem a condição i<j. Logo, o procedimento partição realizará O(n) comparações.

O pior caso ocorre quando o particionamento produz uma região com n-1 elementos e outra com somente um elemento.

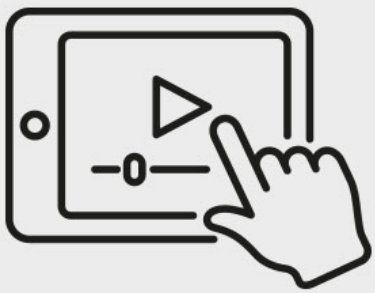
Dessa forma, o tempo de execução no pior caso é $\Theta(n^2)$ para n>= 1.

O melhor caso ocorre quando o particionamento produz duas regiões de tamanho n/2. Nesse caso, o tempo de execução é Θ(n.log n).

A seguir, será apresentado, com exemplos práticos, o algoritmo de ordenação *quick sort*, demonstrando a análise de complexidade.

ALGORITMO DE ORDENAÇÃO *QUICK SORT*

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. AS ESTRATÉGIAS DE DIVISÃO E CONQUISTA SÃO UTILIZADAS PELO SEGUINTE ALGORITMO DE ORDENAÇÃO:

A) *Selection sort*

B) *Insertion sort*

C) *Bubble sort*

D) *Half sort*

E) *Quick sort*

2. (CESGRANRIO - PETROBRAS - ANALISTA DE SISTEMAS JÚNIOR - ENGENHARIA DE *SOFTWARE* - 2011)

A ORDENAÇÃO É UM PROBLEMA MUITO IMPORTANTE PARA OS DESENVOLVEDORES DE *SOFTWARE*. PARA IMPLEMENTÁ-LA, EXISTEM VÁRIOS ALGORITMOS QUE JÁ FORAM AMPLAMENTE ESTUDADOS, COMO O *BUBBLE SORT*, O *QUICK SORT* E O *MERGE SORT*. UMA DAS CARACTERÍSTICAS ESTUDADAS DESSES ALGORITMOS É O TEMPO DE EXECUÇÃO, QUE, USUALMENTE, É MEDIDO PELA NOTAÇÃO O (BIG-OH).

I. O TEMPO DE PIOR CASO DO ALGORITMO *QUICK SORT* É DE ORDEM MENOR QUE O TEMPO MÉDIO DO ALGORITMO *BUBBLE SORT*.

II. O TEMPO MÉDIO DO *QUICK SORT* É O(NLOG2 N), POIS ELE USA COMO ESTRUTURA BÁSICA UMA ÁRVORE DE PRIORIDADES.

III. O TEMPO MÉDIO DO *QUICK SORT* É DE ORDEM IGUAL AO TEMPO MÉDIO DO *MERGE SORT*.

ESTÁ CORRETO APENAS O QUE SE AFIRMA EM:

A) I

B) II

C) III

D) I e III

E) II e III

GABARITO

1. As estratégias de divisão e conquista são utilizadas pelo seguinte algoritmo de ordenação:

A alternativa **"E "** está correta.

O *quick sort* é um algoritmo que utiliza a estratégia de divisão e conquista para ordenar um vetor, ou seja, ele particiona o vetor e vai realizando a ordenação de cada partição. Quando cada partição estiver ordenada, o vetor estará ordenado.

2. (CESGRANRIO - Petrobras - Analista de Sistemas Júnior - Engenharia de *Software* - 2011)

A ordenação é um problema muito importante para os desenvolvedores de *software*. Para implementá-la, existem vários algoritmos que já foram amplamente estudados, como o *bubble sort*, o *quick sort* e o *merge sort*. Uma das características estudadas desses algoritmos é o tempo de execução, que, usualmente, é medido pela notação O (Big-Oh).

I. O tempo de pior caso do algoritmo *quick sort* é de ordem menor que o tempo médio do algoritmo *bubble sort*.

II. O tempo médio do *quick sort* é O(nlog2 n), pois ele usa como estrutura básica uma árvore de prioridades.

III. O tempo médio do *quick sort* é de ordem igual ao tempo médio do *merge sort*.

Está correto apenas o que se afirma em:

A alternativa **"C "** está correta.

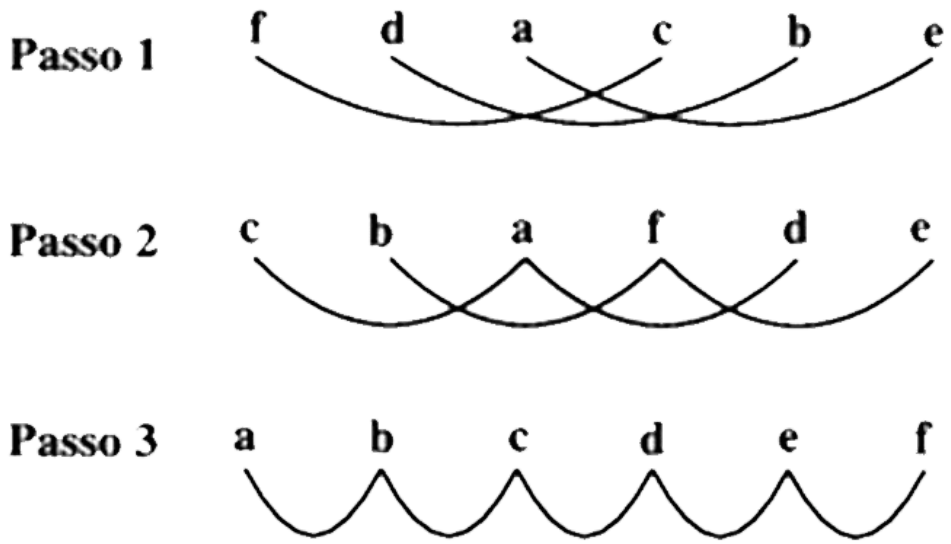
A complexidade de pior caso do *quick sort* é $O(n^2)$, e o tempo médio do *bubble sort* é $O(n^2)$. O tempo médio do *quick sort* é $O(n\log_2 n)$, mas sua estrutura básica é uma lista ou um vetor. O tempo médio do *quick sort* é $O(n \log_2 n)$, e o tempo médio do *merge sort* também.

MÓDULO 4

⊙ **Descrever o funcionamento do algoritmo de ordenação *shell sort***

ALGORITMO DE ORDENAÇÃO *SHELL SORT*

A *ordenação shell* é assim chamada devido a seu inventor: o cientista da computação americano Donald Shell. Mas, provavelmente, o nome vingou, porque seu método de operação é descrito, com frequência, como conchas do mar empilhadas umas sobre as outras.



Autor: Edgar Gurgel

Ordenação shell.

CARACTERÍSTICAS

O método geral é derivado da ordenação por inserção e é baseado na diminuição dos incrementos. Veja a figura Ordenação *shell*.

Primeiro, todos os elementos que estão três posições afastados uns dos outros são ordenados.

Em seguida, todos os elementos que estão duas posições afastados são ordenados.

Finalmente, todos os elementos adjacentes são ordenados.

Assim, temos:

Resultado: a b c d e f

Não é fácil perceber que esse método conduz a bons resultados ou mesmo que ordene o vetor, mas ele executa ambas as funções.

Cada passo da ordenação envolve relativamente poucos elementos ou elementos que já estão razoavelmente em ordem.

Logo, a ordenação shell é eficiente, e cada passo aumenta a ordenação dos dados.

A sequência exata para os incrementos pode mudar. A única regra é que o último incremento deve ser 1.

Por exemplo, a sequência 9 5 3 2 1 funciona bem e é usada na ordenação *shell* mostrada.

> ### ATENÇÃO
>
> Evite sequências que são potências de 2, pois, por razões matemáticas complexas, elas reduzem a eficiência do algoritmo de ordenação, embora a ordenação ainda funcione.

IMPLEMENTAÇÃO

O algoritmo que implementa a ordenação *shell sort* é o seguinte:

```
// algoritmo de ordenação Shell Sort
void shell(char *item, int count) {
        register int i, j, gap, k;
        char x, a[5];
        a[0] = 9; a[1] = 5; a[2] = 3; a[3] = 2; a[4] = 1;

        for( k = 0; k < 5; k++) {
                gap = a[ k ];
                for ( i = gap; i < count; i++) {
                        x = item[ i ];
                                for ( j = i - gap; x < item[ j ] && j >= 0; j = j  - gap)
                                        item [ j + gap ] = item [ j ] ;
                                item [ j + gap] = x;
                }
        }
}
```

Observe que o laço **for** mais interno tem duas condições de teste.

A comparação **x < item[j]** é obviamente necessária para o processo de ordenação.

O teste **j >= 0** evita que os limites da matriz **item** sejam ultrapassados.

Essas verificações extras diminuirão até certo ponto o desempenho da ordenação *shell*.

Versões um pouco diferentes desse tipo de ordenação empregam elementos especiais de matriz, chamados de **sentinelas**, que não fazem parte realmente da matriz ordenada.

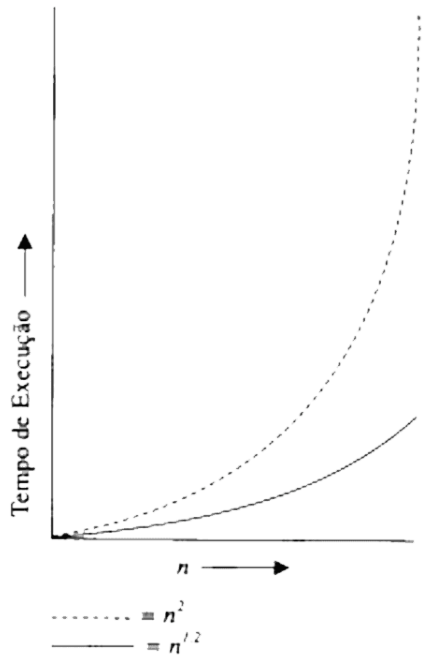
Sentinelas guardam valores especiais de terminação, que indicam o menor e o maior elemento possível. Dessa forma, as verificações dos limites são desnecessárias. No entanto, usar sentinelas requer um conhecimento específico dos dados, o que limita a generalização da função de ordenação.

ANÁLISE DE COMPLEXIDADE

A análise de ordenação do *shell* apresenta alguns problemas matemáticos.

O tempo de execução é proporcional a $\mathbf{n^{1.2}}$ para se ordenar n elementos. Esta é uma redução significativa com relação às ordenações **n-quadrado**.

Para entender o quanto essa ordenação é melhor, veja, a seguir, os gráficos das ordenações n^2 e $n^{1.2}$:



Autor: Gabriele Passeri

📷 Curvas n^2 e $n^{1.2}$.

COMPARAÇÃO ENTRE ALGORITMOS

Agora, vamos analisar, de forma comparativa, a complexidade dos algoritmos apresentados.

A eficiência do algoritmo *bubble sort* diminui drasticamente, à medida que o número de elementos no *array* aumenta.

Em outras palavras, esse algoritmo não é recomendado para aplicações que envolvam grandes quantidades de dados ou que precisem de velocidade.

Considerando um *array* com N elementos, o tempo de execução do *bubble sort* é:

O(n) – Melhor caso – Os elementos já estão ordenados.

$O(n^2)$ – Pior caso – Os elementos estão ordenados na ordem inversa.

$O(n^2)$ – Caso médio.

Assim como o *bubble sort*, o algoritmo *selection sort* não é eficiente.

Sua eficiência diminui drasticamente, à medida que o número de elementos no *array* aumenta. Por isso, esse algoritmo também não é recomendado para aplicações que envolvam grandes quantidades de dados ou que precisem de velocidade.

Considerando um *array* com N elementos, o tempo de execução do *selection sort* é sempre da ordem de $O(n^2)$. Como podemos notar, sua eficiência não depende da ordem inicial dos elementos.

Diante de um *array* com N elementos, o tempo de execução do *insertion sort* é:

O(n) – Melhor caso – Os elementos já estão ordenados.

$O(n^2)$ – Pior caso – Os elementos estão ordenados na ordem inversa.

$O(n^2)$ – Caso médio.

Agora, levando em conta um *array* de N elementos, o tempo de execução do *merge sort* é sempre de ordem **O(N log N)**.

Como podemos observar, a eficiência do *merge sort* não depende da ordem inicial dos elementos.

Embora a eficiência do *merge sort* seja a mesma, independentemente da ordem dos elementos, esse algoritmo possui um gasto extra de espaço de memória em relação aos demais métodos de ordenação. Isso ocorre porque ele cria uma cópia do *array* para cada chamada recursiva.

Em outra abordagem, é possível utilizar um único *array* auxiliar ao longo de toda a execução do *merge sort*.

No pior caso, o *merge sort* realiza cerca de 39% menos comparações do que o *quick sort* em seu caso médio.

Já no melhor caso, o *merge sort* realiza cerca de metade do número de iterações de seu pior caso.

O tempo de execução do *quick sort* depende de o particionamento ser ou não balanceado, o que, por sua vez, depende de quais elementos são utilizados para o particionamento. Se este é balanceado, o algoritmo executa tão rapidamente quanto o *merge sort*. Mas, se o particionamento não é balanceado, o algoritmo *quick sort* é tão lento quanto o *insertion sort*.

COMENTÁRIO

O *shell sort* possui a vantagem de ser uma ótima opção para arquivos de tamanho moderado. Sua implementação é simples e requer uma quantidade de código pequena. Entretanto, como desvantagem, o tempo de execução do algoritmo é sensível à ordem inicial do arquivo, e o método não é estável.

A tabela a seguir apresenta, resumidamente, a comparação entre os algoritmos de ordenação:

Algoritmo	Comparações			Movimentações			Espaço	Estável
	Melhor	Médio	Pior	Melhor	Médio	Pior		
Bubble	$O(n^2)$			$O(n^2)$			O(1)	Sim
Selection	$O(n^2)$			O(n)			O(1)	Não*
Insertion	O(n)		$O(n^2)$	O(n)		$O(n^2)$	O(1)	Sim
Merge	O(n log n)			-			O(n)	Sim

Algoritmo	Comparações			Movimentações			Espaço	Estável
	Melhor	Médio	Pior	Melhor	Médio	Pior		
Quick	O(n log n)		$O(n^2)$	-			O(n)	Não*
Shell	$O(n^{1.25})$ ou $O(n\ (\ln n)^2)$			-			O(1)	Não

Resumo comparativo dos algoritmos de ordenação.

A seguir, será apresentado com exemplos práticos, o algoritmo de ordenação *shell sort*, demonstrando a análise de complexidade.

ALGORITMO DE ORDENAÇÃO SHELL SORT

VERIFICANDO O APRENDIZADO

1. ENTRE OS MÉTODOS DE ORDENAÇÃO A SEGUIR, QUAL DELES É DERIVADO DA ORDENAÇÃO POR INSERÇÃO E BASEADO NA DIMINUIÇÃO DOS INCREMENTOS?

A) *Quick sort*

B) *Bubble sort*

C) *Selection sort*

D) *Merge sort*

E) *Shell sort*

2. UMA ÓTIMA OPÇÃO PARA ARQUIVOS DE TAMANHO MODERADO, CUJA IMPLEMENTAÇÃO É SIMPLES, É O ALGORITMO DE ORDENAÇÃO:

A) *Shell sort*

B) *Bubble sort*

C) *Selection sort*

D) *Merge sort*

E) *Quick sort*

GABARITO

1. Entre os métodos de ordenação a seguir, qual deles é derivado da ordenação por inserção e baseado na diminuição dos incrementos?

A alternativa **"E "** está correta.

O *shell sort* é uma adaptação do algoritmo *insertion sort* e é baseado na diminuição do número de incrementos para a comparação dos elementos do vetor.

2. Uma ótima opção para arquivos de tamanho moderado, cuja implementação é simples, é o algoritmo de ordenação:

A alternativa **"A "** está correta.

O *shell sort* é bom para ordenar um número moderado de elementos, pois requer uma quantidade de código pequena.

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Neste tema, percorremos os diversos algoritmos de ordenação que podem ser empregados para resolução de problemas. A correta escolha do algoritmo impactará significativamente no desempenho do programa que será desenvolvido.

Inicialmente, abordamos os algoritmos de ordenação elementares (inserção, seleção e bolha) e realizamos a análise de complexidade de cada um desses métodos.

Em seguida, discutimos sobre os algoritmos de ordenação avançados (*merge sort, quick sort e shell sort*) e apresentamos as complexidades computacionais de cada um deles.

Para ouvir um *podcast* sobre o assunto, acesse a versão online deste conteúdo.

AVALIAÇÃO DO TEMA:

REFERÊNCIAS

CORMEN, T. H. *et al*. **Algoritmos**: teoria e prática. Rio de Janeiro: Elsevier, 2002.

PONTI JÚNIOR, M. **Ordenação em memória interna (parte 1)**. São Paulo: USP, 2010.

SAMPAIO NETO, N. C. **Análise de algoritmos** – algoritmos de ordenação. Macapá: UNIFAP, 2016.

SAUNDERS, D. **Algorithms**: recurrence relations – master theorem and muster theorem. Newark: University of Delaware, 2011.

SCHILDT, H. **C completo e total**. São Paulo: Makron Books, 1996.

EXPLORE+

Pesquise na internet sobre outros algoritmos de ordenação, como o *heap sort* (seleção em árvore), e analise a complexidade computacional de cada um.

CONTEUDISTA

Edgar Augusto Gonçalves Gurgel do Amaral

🔗 CURRÍCULO LATTES