# Algoritmos de Ordenação

**EINF4_T2** **GRUPO A** **TRAB 2**

20210429- Ivandro Neto - Heap Sort
20210663- Siomara dos Santos - Shell Sort
20210765- Eugelice Yuye - Selection Sort
20210571 - André Yanga - Insertion Sort
20211354 - Jéssica Correia - Quick Sort
20211624 - Manuela Oliveira - Merge Sort

**DOCENTE :** Sílvia António **ISPTEC 2023**

# INTRODUÇÃO

Algoritmo é uma sequência de instruções bem definidas, normalmente usadas para resolver problemas de matemática específicos, executar tarefas, ou para realizar cálculos e equações.

Um algoritmo de ordenação é usado para organizar uma determinada matriz ou lista de elementos de acordo com um operador de comparação nos elementos.

**Tipos de algoritmos de ordenação**

Heap Sort
Shell Sort
Selection Sort
Insertion Sort
Quick Sort
Merge Sort

# Heapsort

O algoritmo organiza os elementos de maior para menor (MaxHeap), de seguida troca o último elemento do array com o primeiro elemento e elimina o último elemento da nossa árvore, e procede desta forma até restar apenas um elemento na nossa árvore.

**Árvore Binária** é uma forma de organizar os dados com base em uma estrutura de dados que tem formato de uma árvore genealógica que consiste em ter 3 elementos, sendo:

- **Data** – A informação na qual cada nó da árvore deverá armazenar;
- **Left** – Um ponteiro que irá apontar para uma sub-árvore ou folha do lado esquerdo;
- **Right** – Um ponteiro que irá apontar para uma sub-árvore ou folha do lado direito;

## Estabilidade

É um algoritmo de classificação instável.

## Complexidade

**Fórmula Geral**
**O(n*logn)**

## Funcionalidade

Converta a matriz em estrutura de dados heap usando heapify. Um por um, exclua o nó raiz do Max-heap e substitua-o pelo último nó no heap e, em seguida, heapify a raiz do heap. Repita esse processo até que o tamanho do heap seja maior que 1.

Crie um heap a partir da matriz de entrada fornecida e repita as etapas a seguir até que a pilha contenha apenas um elemento:

a. Troque o elemento raiz do heap pelo último elemento do heap.
b. Remova o último elemento da pilha
c. Empilhe os elementos restantes da pilha.

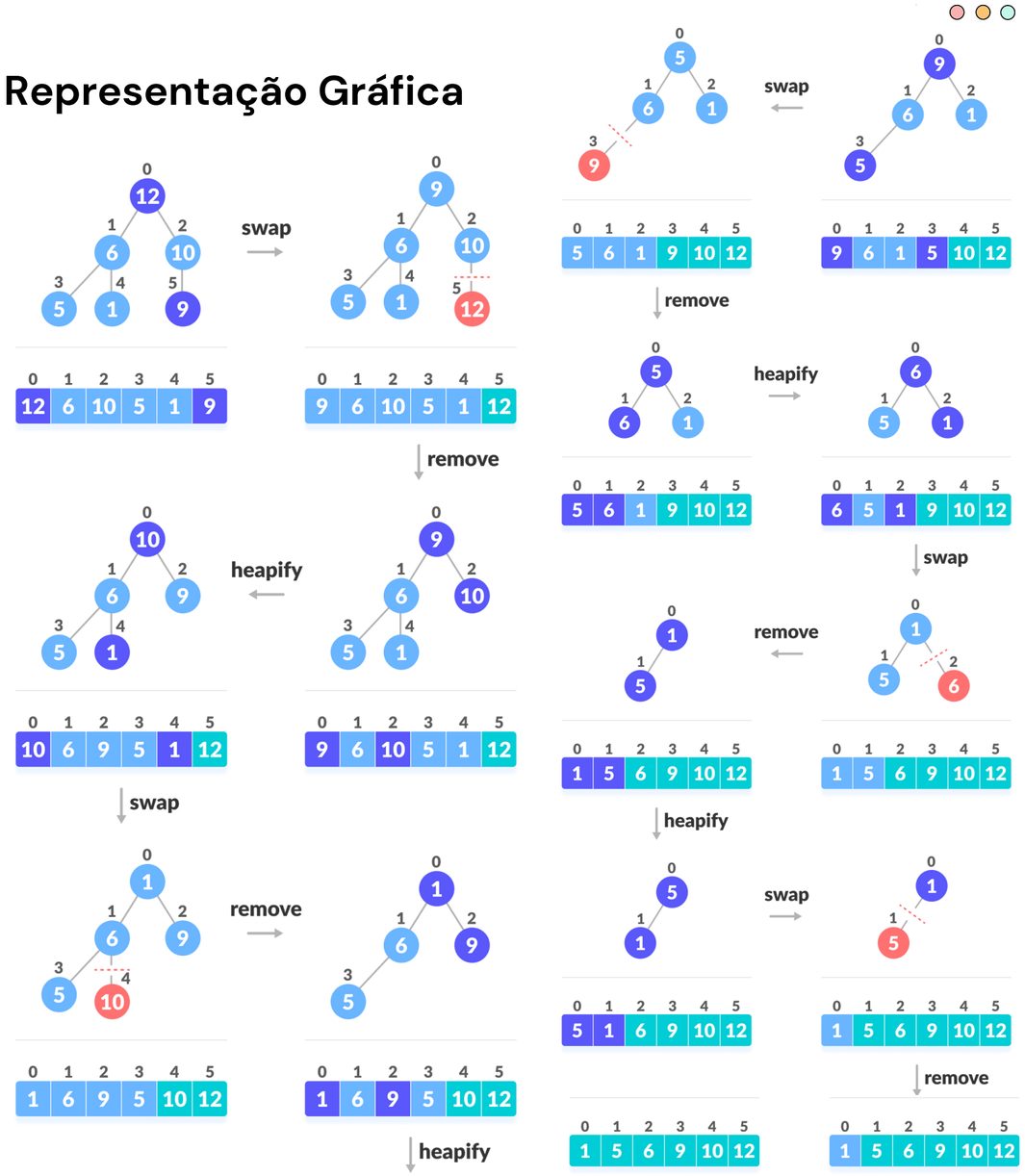
A matriz classificada é obtida invertendo a ordem dos elementos na matriz de entrada.

## Nota

**Heapify** é uma função recursiva que consiste em comparar e trocar os elementos presentes no array organizando-os em formato de uma árvore binária.

**Max-Heap** é a forma na qual os elementos do array serão organizados, colocando os menores valores no final e os maiores no ínicio, enquanto o **min-heap** é o seu inverso.

## Representação Gráfica



# Heapsort – implementação em C

```c
#include <stdio.h>
#include <stdlib.h>

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n\n");
}
void swap(int *x, int *y)
{
    int aux = *x;
    *x = *y;
    *y = aux;
}
void min_heapify(int arr[], int size, int current)
{
    int smallest = current;
    int left = 2 * current + 1;
    int right = 2 * current + 2;
    // Encontra o menor elemento entre o nó atual e seus filhos esquerdo e direito
    if (left < size && arr[left] < arr[smallest])
        smallest = left;
    if (right < size && arr[right] < arr[smallest])
        smallest = right;
    // Se o nó atual não for o menor, troca ele com o menor e chama heapify recursivamente para o subárvore afetada
    if (smallest != current)
    {
        swap(&arr[current], &arr[smallest]);
        min_heapify(arr, size, smallest);
    }
}

void heapify(int arr[], int size, int current)
{
    // primeiramente, localiza-se os elementos da esquerda e direita
    int largest = current;       // Inicialmente o maior valor será sempre o elemento raiz
    int left = 2 * current + 1;  // fórmula para achar o elemento esquerdo da árvore
    int right = 2 * current + 2; // fórmula para achar o elemento direito da árvore

    // Verifica-se se o índice do elemento esquerdo está dentro do array e se o seu valor é maior que o maior valor
    if (left < size && arr[left] > arr[largest])
        largest = left;

    // Verifica-se se o índice do elemento direito está dentro do array e se o seu valor é maior que o maior valor
    if (right < size && arr[right] > arr[largest])
        largest = right;

    // Verifica-se se o elemento selecionado se é o maior também
    if (current != largest)
    {
        // Caso não for os troque
        swap(&arr[largest], &arr[current]);
        // Depois pega o maior como o selecionado da próxima verificação
        printArray(arr, size);
        heapify(arr, size, largest);
    }
}
void MaxHeap(int arr[], int size)
{
    // Começando pelo último nó não-folha, chama heapify para cada nó em ordem reversa de nível para garantir um max heap válido
    for (int i = size / 2 - 1; i >= 0; i--)
        heapify(arr, size, i);
}
void MinHeap(int arr[], int size)
{
    // Começando pelo último nó não-folha, chama heapify para cada nó em ordem reversa de nível para garantir um min heap válido
    for (int i = size / 2 - 1; i >= 0; i--)
        min_heapify(arr, size, i);
}
void HeapSort(int arr[], int size)
{
    // Gera um max heap apartir do array dado.
    MaxHeap(arr, size);
    /* Para min heap comenta a linha à cima e descomenta a linha abaixo */
    // MinHeap(arr, size);
    // Heap Sort
    for (int i = size - 1; i >= 0; i--)
    {
        // Troca a posição do primeiro elemento com o último do array
        swap(&arr[0], &arr[i]);
        // Elimina o último elemento e volta a reorganiza-los até sobrar um único elemento
        heapify(arr, i, 0);
        /*  Para min heap comenta a linha à cima e descomenta a linha abaixo    */
        // min_heapify(arr, i, 0);
    }
}

int main(int argc, char const *argv[])
{
    int arr[] = {1, 12, 9, 5, 6, 10, 20};
    int size = sizeof(arr) / sizeof(arr[0]); // Obtendo o tamanho do array
    printArray(arr, size);
    HeapSort(arr, size);
    printArray(arr, size);
    // OUTPUT::1 5 6 9 10 12 20
    return 0;
}
```

# Shellsort

Criado por Donald Shell em 1959, publicado pela universidade de Cincinnati. É considerado um refinamento do método Insertion Sort. O algoritmo difere do Insertion Sort pelo facto de ao invés de considerar o vetor a ser ordenado como um único segmento, ele considera vários segmentos.

## Estabilidade

É um algoritmo de classificação  não estável .

## Complexidade

**Melhor  Caso**
**O(n^2)**

**Pior Caso**
**O(nlog n)**

**Caso Médio**
**O(nlog n)**

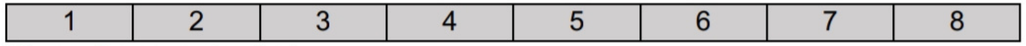
## Representação Gráfica

Começamos a organizar os elementos dividindo pela metade, como temos 8 elementos, o resultado é 4.


O 3 é maior que 6? Se a resposta for verdadeira trocamos o elemento de lugar, se não deixamos na mesma posição.


Dividimos os 8 elementos pela metade e o resultado foi 4, agora pegamos esse valor 4 e dividimos pela metade, o resultado é 2.


Pegamos o resultado anterior que foi 2 e dividimos pela metade, o resultado é 1.




## Funcionalidade

A ordenação é realizada usando uma sequência de valores <h1, h2, h3...hN> onde começando por hN selecionamos apenas os valores que estão hN elementos distantes um do outro, então ordenamos esses elementos com algum algoritmo de ordenação simples.

**Definir valores para h**

- Escolher aleatoriamente **(baixa eficiência)**
- Proposta de Donald Shell foi **h = n/2 (baixa eficiência)**
- Proposta de Donald Knuth **h= 3*h+1,** onde para decrementar pode se utilizar a fórmula inversa **h=(h-1)/ 3**

# Shellsort – implementação em C

```
void shellsort(struct item *v, int n) {
   int i, j, h;
   struct item aux;

   for(h = 1; h < n; h = 3*h+1); /* calcula o h inicial. */

   while(h > 0) {
      h = (h-1)/3; /* atualiza o valor de h. */
      for(i = h; i < n; i++) {
         aux = v[i];
         j = i;
         /* efetua comparações entre elementos com distância h: */
         while(v[j - h].chave > aux.chave) {
            v[j] = v[j - h];
            j -= h;
            if(j < h) break;
         }
         v[j] = aux;
      }
   }
}
```

# Selectionsort

A ideia de ordenação por selecção é procurar o menor elemento do vector (ou maior) e movimentá-lo para a primeira (última) posição do vector.

## Estabilidade

A implementação padrão não é estável. No entanto, qualquer algoritmo de ordenação pode se tornar estável.

## Complexidade

É um algoritmo simples de executar por apresentar uma das menores quantidades de movimentos entre os elementos;

A sua eficiência diminui a medida que a quantidade de dados aumenta, torna-se mais lento;

O tempo de execução é na ordem de O(n^2).

## Funcionalidade

Selecciona-se o maior ou menor elemento do conjunto e troca-se pelo primeiro elemento. Para os elementos restantes faz-se (n-1). Onde 'n' é o número de elementos.

Selecciona-se o maior ou menor elemento e troca-se pelo elemento da segunda posição e assim sucessivamente até que os elementos estejam ordenados.

## Nota

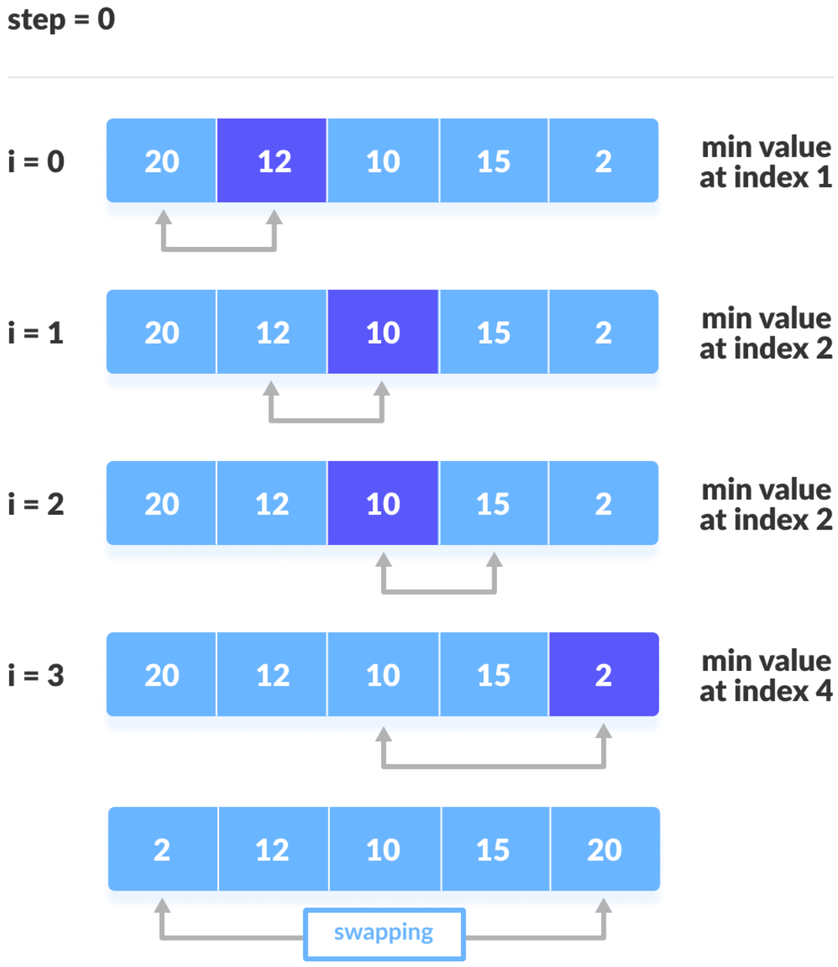
O "for" mais interno deste algoritmo é responsável por encontrar o menor elemento do vector.

As linhas de código mais abaixo do segundo for são responsáveis por colocar o menor elemento identificado pelo for mais interno em sua posição correta.

O "for" mais externo é responsável por percorrer todo o vector, permitindo que todos os elementos sejam colocados no lugar certo.

O valor da variável i representa a posição atual onde o menor elemento deverá ser colocado.

## Representação Gráfica



# Selectionsort – implementação em C

```
1
2  Void selection_sort(int *v, int n)
3  {
4    Int i, j, menor, aux;
5    for(i=0; i<n-1;i++)
6    {
7      menor=i;
8      for(j=i+1;j<n;j++)
9      {
10       if(v[i]<v[menor])
11         menor=j;
12     }
13     If(i != menor)
14     {
15       aux=v[i];
16       v[i]=v[menor];
17       v[menor]=aux;
18     }
19   }
20 }
```

# Insertionsort

É um algoritmo de classificação simples, onde a matriz é virtualmente dividida em uma parte classificada e uma parte não classificada. Os valores da parte não classificada são escolhidos e colocados na posição correta da parte classificada.

## Funcionalidade

- O procedimento usa um único argumento, 'A' , que é uma lista de itens classificáveis.
- A variável 'n' é atribuída ao comprimento da matriz A.
- O loop for externo começa no índice '1' e é executado por iterações 'n-1' , onde 'n' é o comprimento da matriz.
- O loop while interno começa no índice atual i do loop for externo e compara cada elemento com seu vizinho esquerdo. Se um elemento for menor que seu vizinho esquerdo, os elementos serão trocados.
- O loop while interno continua a mover um elemento para a esquerda, desde que seja menor que o elemento à sua esquerda.
- Depois que o loop while interno é concluído, o elemento no índice atual está em sua posição correta na parte classificada da matriz.
- O loop for externo continua iterando pela matriz até que todos os elementos estejam em suas posições corretas e a matriz esteja totalmente classificada.

## Estabilidade

É um algoritmo de classificação estável .

## Complexidade

**Fórmula Geral**
**O(n^2)**

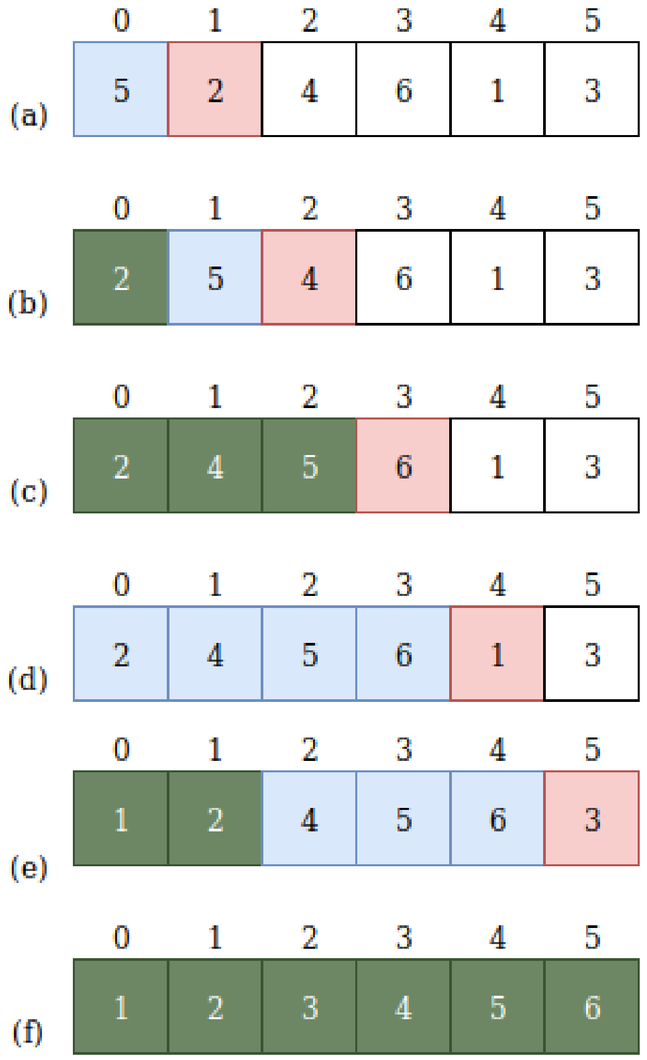
**Pior Caso**
Quando os elementos forem classificados em ordem inversa.

**Melhor Caso**
Quando os elementos já estão classificados **O(n)**

## Pseudo-Código

```
procedimento insertSort(A: lista de itens classificáveis)
  n = comprimento(A)
  para i = 1 até n − 1 faça
    j = eu
    enquanto j > 0 e A[j-1] > A[j] faça
      troca(A[j], A[j-1])
      j = j − 1
    terminar enquanto
  fim para
procedimento final
```

## Representação Gráfica



# Insertionsort – implementação em C

```c
void insertionSort(int *lista, int tamanho)
{
  int i, j, aux;
  for ( i = 0; i < tamanho - 1; i++)
  {
    if (lista[i] > lista[i+1])
    {
      aux = lista[i+1];
      lista[i+1] = lista[i];
      lista[i] = aux;
      j = i-1;
      while (j>=0)
      {
        if (aux < lista[j])
        {
         lista[j+1] = lista[j];
         lista[j] = aux;
         }
         else
         {
         break;
         }
         j=j-1;
      }
    }
  }
}
```

# Quicksort

É um algoritmo que utilizando o paradigma de Dividir para Conquistar, particiona o vector recursivamente até que todas as subsequências estejam ordenadas, para uni-las novamente.

## Estabilidade

A implementação padrão não é estável. No entanto, qualquer algoritmo de ordenação pode se tornar estável.

## Complexidade

**Fórmula Geral**
**T(n) = T(k) + T(nk-1) + O(n)**

**Pior Caso**
Quando o processo de partição sempre escolhe o maior ou o menor elemento como pivô. **O(n^2)**

**Melhor Caso**
Quando o processo de partição sempre escolhe o elemento do meio como pivô. **O(nLogn)**

**Caso Médio**
**(theta)(nLogn)**

## Funcionalidade

O algoritmo receber um array preenchido, e escolhe um pivô, que pode ser selecionado aleatoriamente.

1

Duas variáveis j e i, são posicionadas nos extremos do array. A variável j começa a percorrer da esquerda a direita comparando os valores aos do pivô. Se o valor de j for maior a constante para, e a variável i começa a percorrer da direita a esquerda. Se o valor de i for menor que o pivô, a constante para e é feita uma troca com a posição em que a variável j parou.

2

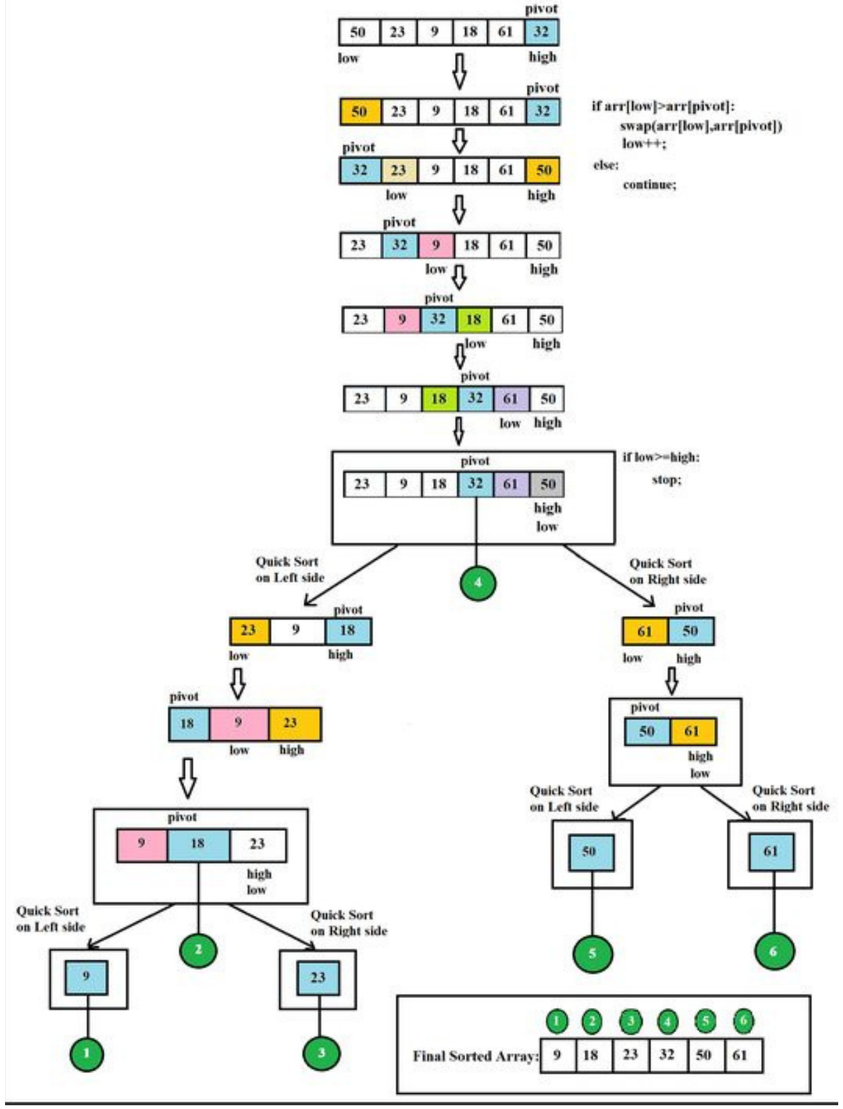
As duas variáveis andam uma casa, e o processo repete-se até o momento em que elas encontraram-se durante o percurso. Quando isso acontece o array é dividido, e é feita uma chamada recursiva para cada nova repartição

3

O processo repete-se até o momento em que só houver um único elemento em cada repartição gerada. E por fim todos os valores são unidos novamente.

4

## Representação Gráfica



# Quicksort – implementação em C

```c
#include <stdlib.h>
#include <stdio.h>

// função que realiza a troca entre dois elementos
void troca(int vet[], int i, int j)
{
        int aux = vet[i];
        vet[i] = vet[j];
        vet[j] = aux;
}

// particiona e retorna o indice do pivô
int particiona(int vet[], int inicio, int fim)
{
        int pivo, pivo_indice, i;

        pivo = vet[fim]; // o pivô é sempre o último elemento
        pivo_indice = inicio;

        for(i = inicio; i < fim; i++)
        {
                // verifica se o elemento é <= ao pivô
                if(vet[i] <= pivo)
                {
                        // realiza a troca
                        troca(vet, i, pivo_indice);
                        // incrementa o pivo_indice
                        pivo_indice++;
                }
        }
        // troca o pivô
        troca(vet, pivo_indice, fim);

        // retorna o indice do pivô
        return pivo_indice;
}

// escolhe um pivô aleatorio para evitar o pior caso do quicksort
int particiona_random(int vet[], int inicio, int fim)
{
        // seleciona um número entre fim (inclusive) e inicio (inclusive)
        int pivo_indice = (rand() % (fim - inicio + 1)) + inicio;

        // faz a troca para colocar o pivô no fim
        troca(vet, pivo_indice, fim);
        // chama a particiona
        return particiona(vet, inicio, fim);
}

void quick_sort(int vet[], int inicio, int fim)
{
        if(inicio < fim)
        {
                // função particionar retorna o indice do pivô
                int pivo_indice = particiona_random(vet, inicio, fim);

                // chamadas recursivas quick_sort
                quick_sort(vet, inicio, pivo_indice - 1);
                quick_sort(vet, pivo_indice + 1, fim);
        }
}

int main()
{
        // vetor que será ordenado
        int vet[] = {25,40,55,20,44,35,38,99,10,65,50};
        int tam_vet = sizeof(vet) / sizeof(int);
        int i;

        // inicializa random seed
        srand(time(NULL));

        // chamada do quicksort
        quick_sort(vet, 0, tam_vet - 1);

        // mostra o vetor ordenado
        for(i = 0; i < tam_vet; i++)
                printf("%d ", vet[i]);

        return 0;
}
```

# Mergesort

É um algoritmo de ordenação que funciona dividindo um array em subarrays menores, ordenando cada subarray e, em seguida, junta ou mescla os subarrays ordenados novamente para formar o array ordenado final.

## Estabilidade

É um algoritmo de classificação estável.

## Complexidade

A sua complexidade não depende da sequência de entrada, e é medida pelo número de divisões e combinações e é de fácil implementação

**Caso Geral**
**O(n*logn)**

## Funcionalidade

1. Pensemos nisso como um algoritmo recursivo que divide continuamente o vector ao meio até que ele não possa mais ser dividido.

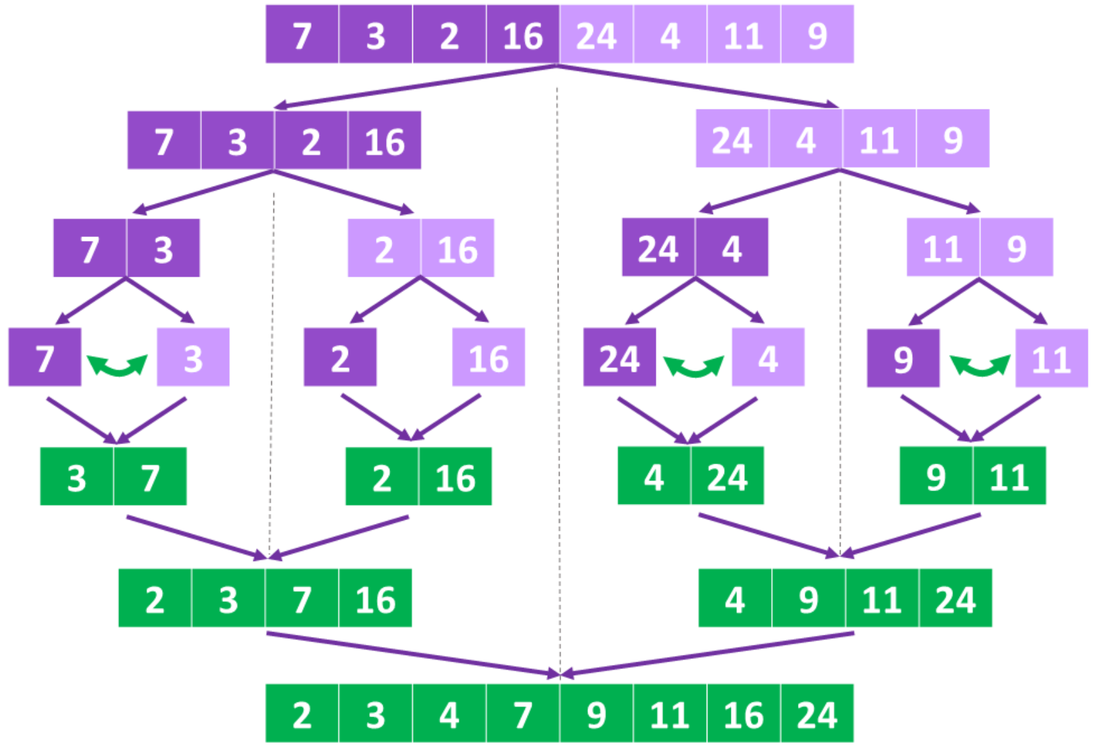
2. Isso significa que se o vector ficar vazio ou tiver apenas um elemento restante, a divisão será interrompida, ou seja, é o caso básico para interromper a recursão.

3. Finalmente, quando ambas as metades forem ordenadas, a operação de mesclagem ou junção é aplicada.

## Nota

requer um vector auxiliar com as mesmas dimensões do que estiver a ser ordenado, o que aumenta o consumo de memória e tempo de execução.

## Representação Gráfica



# Mergesort – implementação em C

```c
// Merge sort em C

#include <stdio.h>

void merge(int arr[], int p, int q, int r) {

  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];


  int i, j, k;
  i = 0;
  j = 0;
  k = p;


  while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    arr[k] = M[j];
    j++;
    k++;
  }
}

void mergeSort(int arr[], int l, int r) {
  if (l < r) {

    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    merge(arr, l, m, r);
  }
}

void printArray(int arr[], int size) {
  for (int i = 0; i < size; i++)
    printf("%d ", arr[i]);
  printf("\n");
}

//MAIN
int main() {
  int arr[] = {6, 5, 12, 10, 9, 1};
  int size = sizeof(arr) / sizeof(arr[0]);

  mergeSort(arr, 0, size - 1);

  printf("Array ordenado: \n");
  printArray(arr, size);
}
```

# CONCLUSÃO

- O Quicksort é o mais rápido na prática, porque seu loop interno pode ser implementado com eficiência na maioria das arquiteturas e na maioria dados do mundo real. E há experimentos que demonstram que o Quicksort em seu melhor caso e caso médio é por volta de 3x mais eficiente que o Mergesort, porque ele contém constantes menores.

- O Mergesort, tem a eficiência n∗lognp para todos os casos. Ou seja, isso nos dá uma garantia de que, independente da disposição dos dados em um array, a ordenação será eficiente. Ele é mais eficiente que o quicksort em listas vinculada,devido à diferença na alocação de memória de arrays, onde o acesso pode ser aleatório, e listas vinculadas, onde o  acesso dos dados é sequencial.

- O Heapsort em comparação com o quicksort e mergesort, perde em termos de eficiência. No entanto ele ajuda a resolver outros problemas de forma eficiente, por ex: a procura do maior e menor elemento ou a implementação de listas de prioridades.

- O Shellsort é uma boa solução para arquivos de tamanho moderado (na ordem de centenas).

- O Insertionsort é um bom método para arquivos quase ordenados em contrapartida para grandes quantidades de dados a ordenação torna-se lenta tal como o Selectionsort.

# BIBLIOGRAFIA

## HeapSort


- (s.d.). Obtido em 15 de Março de 2023, de Binary Heap - GeeksforGeeks: http://www.geeksforgeeks.org/
- (s.d.). Obtido em 15 de Março de 2023, de Heap Sort - GeeksforGeeks: http://www.geeksforgeeks.org/
- (s.d.). Obtido em 16 de Março de 2023, de Heap Sort (With Code in Python, C++, Java and C): http://www.programiz.com/
- (s.d.). Obtido em 16 de Março de 2023, de Introduction to Binary Tree - Data Structure and Algorithm Tutorials - GeeksforGeeks: http://www.geeksforgeeks.org/
- (s.d.). Obtido em 16 de Março de 2023, de Binary Tree: http://www.programiz.com/
- (s.d.). Obtido em 18 de Março de 2023, de (935) Estrutura de Dados - Aula 15 - Árvores - Conceitos básicos - YouTube: http://www.youtube.com/
- Learn Heap sort in C program | Simplilearn. (s.d.). Obtido em 18 de Março de 2023, de http://www.simplilearn.com/


## ShellSort


- TreinaWeb. (s.d.). Obtido em 24 de Março de 2023, de https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao
- CadernoGeek. (s.d.). Obtido em 24 de Março de 2023, de https://cadernogeek.wordpress.com/tag/shell-sort/
- (s.d.). Obtido em 26 de Março de 2023, de https://youtu.be/9TGB6A4svMk
- Slideshare. (s.d.). Obtido em 26 de Março de 2023, de https://pt.slideshare.net/jackocap/aa-algoritmo-shell-sort
- (s.d.). Obtido em 26 de Março de 2023, de http://tiagodemelo.info/wp-content/uploads/2019/08/algoritmos-ordenacao-1.pdf


## SelectionSort


- AlgoRythmics. (s.d.). Obtido em 25 de 03 de 2023, de https://youtu.be/Ns4TPTC8whw
- Coelho, H., & Félix, N. (s.d.). Obtido em 15 de Março de 2023, de https://ww2.inf.ufg.br/~hebert/disc/aed1/AED1_04_ordenacao1.pdf
- Sambol, M. (s.d.). Obtido em 15 de 03 de 2023, de https://youtu.be/g-PGLbMth_g
- Viana, D. (s.d.). TREINAWEB. Obtido em 25 de 03 de 2023, de https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao#:~:text=Insertion%20Sort,-Insertion%20Sort%20ou&text=Possui%20complexidade%20C(n)%20%3D,caso%20m%C3%A9dio%20e%20pior%20caso.


## InsertionSort


- (s.d.). Obtido em 27 de Março de 2023, de https://www.geeksforgeeks.org/insetion-sort/


## QuickSort


- (s.d.). Obtido em 15 de Março de 2023, de https://www.youtube.com/watch?v=wU7Q8Z51MUI
- (s.d.). Obtido em 15 de Março de 2023, de https://www.geeksforgeeks.org/quick-sort/
- (s.d.). Obtido em 22 de Março de 2023, de https://www.geeksforgeeks.org/introduction-to-divide-and-conquer-algorithm-data-str
- (s.d.). Obtido em 15 de Março de 2023, de https://gist.github.com/marcoscastro/1dd65900cc7b188e1ab9
- (s.d.). Obtido em 22 de Março de 2023, de https://www.programiz.com/dsa/quick-sort
- (s.d.). Obtido em 15 de Março de 2023, de https://joaoarthurbm.github.io/eda/posts/quick-sort/


## MergeSort


- (s.d.). Obtido em 15 de Março de 2023, de https://www.geeksforgeeks.org/merge-sort/