PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS

Agatha Assis - 24021612

Guilherme Guedes - 24021514

Trabalho Ordenação

repositório GitHub:

https://github.com/GuiGuedes10/Trabalho_ordenacao

CAMPINAS

2024

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS

<CENTRO CIÊNCIAS EXATAS,

Introdução

O propósito deste projeto é estudar e comparar diferentes algoritmos de ordenação em termos de eficiência e desempenho prático. A ordenação de dados é uma operação fundamental em ciência da computação, frequentemente utilizada em uma ampla gama de aplicações, desde bancos de dados e sistemas de busca até gráficos e processamento de dados em grande escala. Ao longo do projeto, implementaremos seis algoritmos clássicos de ordenação: **Selection Sort**, **Insertion Sort**, **Bubble Sort**, **Merge Sort**, **Quick Sort** e **Heap Sort**, todos comumente estudados no contexto de algoritmos e estruturas de dados.

Objetivo

O principal objetivo deste projeto é implementar esses seis algoritmos de ordenação e analisar seu desempenho sob diferentes condições. Para isso, serão realizados testes de performance utilizando arrays de diferentes tamanhos, variando de 100 até 100.000 elementos. Através dessas implementações e testes, será possível:

1. **Comparar a complexidade teórica e prática** dos algoritmos: A análise de complexidade permitirá entender o comportamento de cada algoritmo em termos de tempo e espaço, em diferentes cenários (melhor caso, pior caso e caso médio).
2. **Avaliar o tempo de execução**: Medir e comparar o tempo de execução de cada algoritmo para diferentes tamanhos de entrada ajudará a observar como cada algoritmo se comporta à medida que o número de elementos aumenta, e qual deles é mais eficiente em situações específicas.
3. **Gerar uma análise comparativa**: Além da medição do tempo, será realizada uma análise comparativa entre os algoritmos, destacando em quais cenários cada um é mais eficiente ou vantajoso, levando em consideração não só o tempo de execução, mas também o uso de memória.

Através deste estudo, buscamos fornecer uma visão clara sobre as vantagens e desvantagens de cada algoritmo, além de compreender em quais contextos e para quais tipos de entrada cada um deles pode ser mais adequado, otimizando assim a escolha de algoritmos em projetos de software real.

1.Implementação

1.Bubble sort

Descrição:

O algoritmo de **Bubble Sort** é baseado na ideia de "afundar" ou "subir" os elementos para suas posições corretas ao longo do vetor. Ele percorre o vetor repetidamente, comparando elementos adjacentes e trocando-os se estiverem na ordem errada. Isso é repetido até que o vetor esteja totalmente ordenado. A cada passagem, o maior (ou menor, dependendo da direção da comparação) elemento "borbulha" para o final (ou início) do vetor.

Código:

```
#include <stdio.h>

void bubblesort(int *v, int tam) {  // Vetor e tamanho passados como parâmetro
    int i, j; //criação das variáveis de interação
    int aux; // criação da variável auxiliar que serve para receber os elementos
    int cont = 1; // só um contador para ver em quantos passos foi feito
    for (i = 0; i < tam - 1; i++) {           // Loop externo limita o tamanho de cada interação, já que em cada uma ele acha o maior e trava a última casa
        for (j = 1; j < tam - i; j++) {       // Loop interno
            if (v[j] < v[j - 1]) {          // Compara o atual e o anterior. Se a j(atual) for menor
                aux = v[j];                 // variavel auxiliar vai receber o elemento do j atual
                v[j] = v[j - 1];            // e o j atual receberá o j anterior
                v[j - 1] = aux;             // e o j anterior recebe o aux assim colocando os números na ordem
                for (int i = 0; i < tam; i++) { // percorre todo o vetor mostrando cada passo dado
                  printf("%d ",v[i]);
                }
                 printf(" Passo %d \n", cont);
                 cont ++;
            }
        }
    }
}

int main() {
    int v[] = {25 , 57 , 48 , 37 , 12 , 92 , 86 , 33};  // Vetor para ordenar
    int tam = sizeof(v) / sizeof(v[0]); // Calcula o tamanho do vetor

    bubblesort(v, tam);  // Chama a função de ordenação

    // Exibe o vetor ordenado
    for (int i = 0; i < tam; i++) {
```

```
        printf("%d ", v[i]);
    }
    printf("  Vetor ordenado\n");

    return 0;
}
```

Saída:

```
25 48 57 37 12 92 86 33  Passo 1
25 48 37 57 12 92 86 33  Passo 2
25 48 37 12 57 92 86 33  Passo 3
25 48 37 12 57 86 92 33  Passo 4
25 48 37 12 57 86 33 92  Passo 5
25 37 48 12 57 86 33 92  Passo 6
25 37 12 48 57 86 33 92  Passo 7
25 37 12 48 57 33 86 92  Passo 8
25 12 37 48 57 33 86 92  Passo 9
25 12 37 48 33 57 86 92  Passo 10
12 25 37 48 33 57 86 92  Passo 11
12 25 37 33 48 57 86 92  Passo 12
12 25 33 37 48 57 86 92  Passo 13
12 25 33 37 48 57 86 92  Vetor ordenado
```

2.Select sort (pode ser feito procurando o maior ou o menor)

Descrição:

O **Selection Sort** trabalha encontrando o menor (ou maior) valor em cada iteração e trocando-o com o primeiro elemento não ordenado. Em cada passagem, o algoritmo assume que a parte já ordenada está no início do vetor, e o algoritmo vai selecionando o próximo menor (ou maior) elemento da parte não ordenada.

Código:

Exemplo procurando o maior:

```
#include <stdio.h>
// vai procurar o maior e vai trocar a posição de onde tá o maior com a posição do último elemento, trava a última posição e repete isso
void selectsort(int *v, int tam) { // Função recebe os parâmetros
    int i, j, aux;
    int cont =1;

    // Começa do final e vai diminuindo o intervalo da parte não ordenada
    for (i = tam - 1; i > 0; i--) {
        int maior = 0;  // Inicializa o índice do maior elemento como o primeiro da parte não ordenada
```

```
        for (j = 1; j <= i; j++) { //Percorre todo o vetor
            if (v[j] > v[maior]) { //Verifica se o valor atual é maior q o valor maior
                maior = j;  // Se for atualiza o maior elemento
            }
        }

        // Troca o maior elemento encontrado com o último elemento da parte não ordenada
        aux = v[maior];
        v[maior] = v[i];
        v[i] = aux;


         printf("Passo %d: ", cont); // printa cada interação
         for (int i = 0; i < tam; i++) {
            printf("%d ", v[i]);
        }
        cont ++;
        printf("\n");
    }

}
int main() {
    int v[] = {25 , 57 , 48 , 37 , 12 , 92 , 86 , 33};     // Exemplo de vetor para ordenar
    int tam = sizeof(v) / sizeof(v[0]); // Calcula o tamanho do vetor

    selectsort(v, tam);                 // Chama a função de ordenação por seleção

    // Mostra o vetor ordenado
    printf("Vetor ordenado: ");
    for (int i = 0; i < tam; i++) {
        printf("%d ", v[i]);
    }
    printf("\n");

    return 0;
}
```

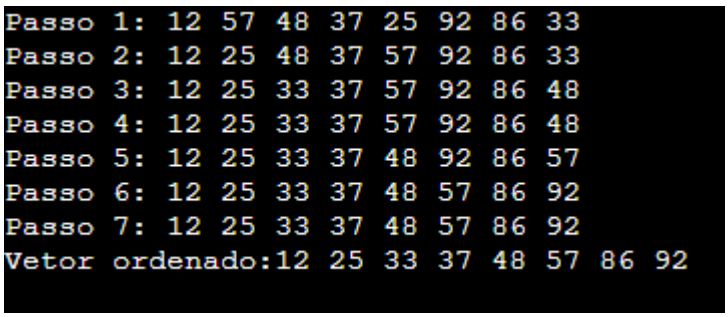
Saída:

```
Passo 1: 25 57 48 37 12 33 86 92
Passo 2: 25 57 48 37 12 33 86 92
Passo 3: 25 33 48 37 12 57 86 92
Passo 4: 25 33 12 37 48 57 86 92
Passo 5: 25 33 12 37 48 57 86 92
Passo 6: 25 12 33 37 48 57 86 92
Passo 7: 12 25 33 37 48 57 86 92
Vetor ordenado: 12 25 33 37 48 57 86 92
```

Exemplo procurando o menor:

```
#include <stdio.h>

void select(int *v, int tam) { //recebe os parâmetros
        int i, j, min;
        int aux;
   int cont =1;

        for (i = 0; i < tam - 1; i++) { // laço de interação externo
                min = i; // inicializa o primeiro como o menor
                for (j = i + 1; j < tam; j++) { // laço interno
                        if (v[j] < v[min]) {      // compara o j atual com o valor do menor
                                min = j; // se o valor menor for maior q o j atual ent o menor
passa a ser o j
                        }
                }
        // Faz a troca para para o menor ficar no começo
                aux = v[min];
                v[min] = v[i];
                v[i] = aux;
                 printf("Passo %d: ", cont); //printa cada interação
       for (int i = 0; i < tam; i++) {
          printf("%d ", v[i]);
       }
       cont ++;
       printf("\n");
          }
}

int main() {
        int v[] = {25 , 57 , 48 , 37 , 12 , 92 , 86 , 33};       // Exemplo de vetor
        int tam = sizeof(v) / sizeof(v[0]);     // Calcula o tamanho do vetor

        select(v, tam);                  //Chama a função

        // Mostra o vetor ordenado
        printf("Vetor ordenado:");
        for (int i = 0; i < tam; i++) {
                printf("%d ", v[i]);
        }
        printf("\n");

        return 0;
}
```

Saída:



3. Insertion sort:

Descrição:

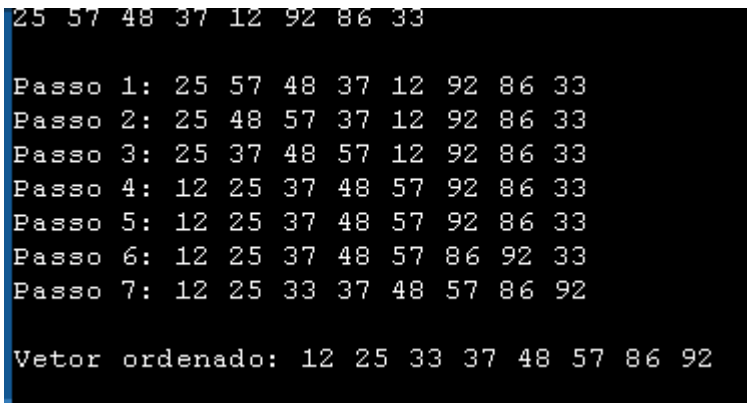
O **Insertion Sort** funciona como se estivesse organizando um conjunto de cartas de baralho, onde um elemento é inserido na posição correta de uma parte já ordenada do vetor. O algoritmo começa com o segundo elemento e o insere na posição correta, movendo todos os elementos maiores para a direita, repetindo esse processo para cada elemento subsequente.

Código:

```
#include <stdio.h>

// Função do Insertion Sort com contagem de passos
void insertionsort(int *v, int tam) {
    int i, j, aux;
    int cont = 1; // Contador de passos

    // Laço externo percorre os elementos de 1 até n-1
    for (i = 1; i < tam; i++) {
        aux = v[i];   // Guarda o valor atual para inserção
        j = i - 1;     // Começa comparando com o elemento anterior

        // Laço interno compara o elemento atual com os anteriores e desloca os maiores
        while (j >= 0 && v[j] > aux) {
            v[j + 1] = v[j];  // Move o elemento para a direita
            j--;              // Move para o próximo elemento da parte ordenada
        }
        v[j + 1] = aux;  // Insere o valor atual na posição correta

        // Imprime o vetor após cada passo (após cada inserção)
        printf("Passo %d: ", cont);
        for (int k = 0; k < tam; k++) {
            printf("%d ", v[k]);
```

```
        }
        printf("\n");
        cont++; // Incrementa o contador de passos
    }
}

int main() {
    int v[] = {25, 57, 48, 37, 12, 92, 86, 33};  // Vetor para ordenar
    int tam = sizeof(v) / sizeof(v[0]);          // Calcula o tamanho do vetor

    printf("Vetor original:\n");
    for (int i = 0; i < tam; i++) {
        printf("%d ", v[i]);
    }
    printf("\n\n");

    insertionsort(v, tam);  // Chama a função de ordenação por inserção

    // Mostra o vetor ordenado
    printf("\nVetor ordenado: ");
    for (int i = 0; i < tam; i++) {
        printf("%d ", v[i]);
    }
    printf("\n");

    return 0;
}
```

Saída:



4. Merge sort:

Descrição:

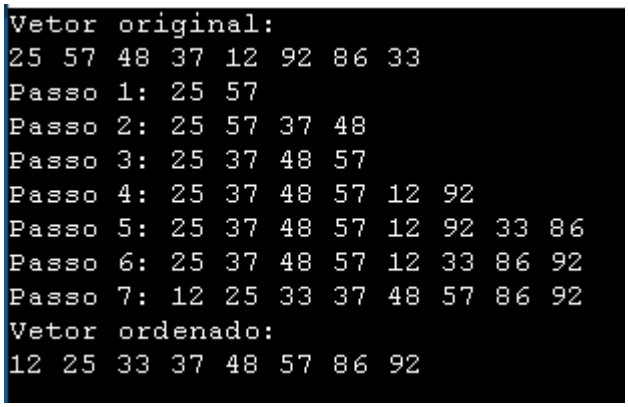
O **Merge Sort** é um algoritmo baseado na técnica de divisão e conquista. Ele divide o vetor em duas metades recursivamente até que cada subvetor tenha um único elemento. Em seguida, os subvetores são mesclados de forma ordenada. O processo de mesclagem garante que o vetor final estará ordenado.

Código:

```
#include <stdio.h>

// Função para mesclar dois subvetores de forma ordenada
void merge(int vetor[], int inicio, int meio, int fim, int* cont) {
    int tamanho = fim - inicio + 1; // Tamanho total do vetor a ser mesclado
    int temp[tamanho]; // Vetor temporário para armazenar a mesclagem
    int i = inicio, j = meio + 1, k = 0;

    // Copiar os elementos dos subvetores para o vetor temporário
    while (i <= meio && j <= fim) {
        if (vetor[i] < vetor[j]) {
            temp[k++] = vetor[i++];
        } else {
            temp[k++] = vetor[j++];
        }
    }

    // Se ainda houver elementos no subvetor esquerdo
    while (i <= meio) {
        temp[k++] = vetor[i++];
    }

    // Se ainda houver elementos no subvetor direito
    while (j <= fim) {
        temp[k++] = vetor[j++];
    }

    // Copiar os elementos de volta para o vetor original
    for (k = 0, i = inicio; i <= fim; i++, k++) {
        vetor[i] = temp[k];
    }

    // Imprimir o vetor a cada passo
    printf("Passo %d: ", (*cont)++);
    for (int i = 0; i <= fim; i++) {
        printf("%d ", vetor[i]);
    }
```

```
    printf("\n");
}

// Função recursiva para dividir o vetor e chamar a mesclagem
void mergeSort(int vetor[], int inicio, int fim, int* cont) {
    if (inicio < fim) {
        int meio = (inicio + fim) / 2;

        // Chamada recursiva para o subvetor esquerdo
        mergeSort(vetor, inicio, meio, cont);

        // Chamada recursiva para o subvetor direito
        mergeSort(vetor, meio + 1, fim, cont);

        // Mesclar os subvetores ordenados
        merge(vetor, inicio, meio, fim, cont);
    }
}

int main() {
    int vetor[] = {25, 57, 48, 37, 12, 92, 86, 33};  // Exemplo de vetor
    int tam = sizeof(vetor) / sizeof(vetor[0]);
    int cont = 1;  // Contador para marcar os passos

    printf("Vetor original:\n");
    for (int i = 0; i < tam; i++) {
        printf("%d ", vetor[i]);
    }
    printf("\n");

    // Chama o mergeSort para ordenar o vetor
    mergeSort(vetor, 0, tam - 1, &cont);

    // Exibe o vetor ordenado
    printf("Vetor ordenado:\n");
    for (int i = 0; i < tam; i++) {
        printf("%d ", vetor[i]);
    }
    printf("\n");

    return 0;
}
```

Saída:



5. Quick sort:

Lógica:

O **Quick Sort** também é baseado em **divisão e conquista**. Ele escolhe um elemento como "pivô" e particiona o vetor em dois subvetores: um com elementos menores que o pivô e outro com elementos maiores. Em seguida, o algoritmo é recursivamente aplicado a cada subvetor até que o vetor esteja completamente ordenado. O pivô é normalmente escolhido de maneira aleatória ou através de heurísticas como "pivô mediano".

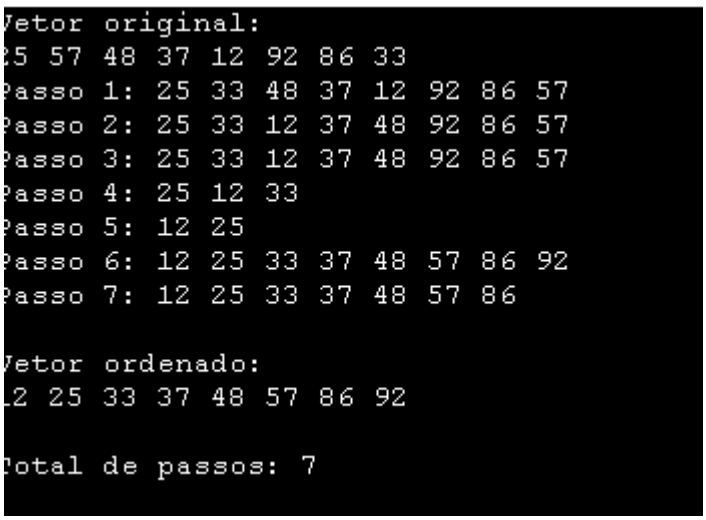
Código:

```
#include <stdio.h>

typedef struct {
    int Chave;
} Item;

int passos = 0;  // Contador de passos global

// Função de particionamento do Quick Sort
void Particao(int Esq, int Dir, int *i, int *j, Item *A) {
    Item x, aux;
    *i = Esq;
    *j = Dir;
    x = A[(*i + *j) / 2]; // Pivo
```

```
    do {
        while (x.Chave > A[*i].Chave) (*i)++;  // Move o índice da esquerda até encontrar um valor maior ou igual ao pivô
        while (x.Chave < A[*j].Chave) (*j)--;  // Move o índice da direita até encontrar um valor menor ou igual ao pivô

        if (*i <= *j) {
            // Troca os elementos A[*i] e A[*j]
            aux = A[*i];
            A[*i] = A[*j];
            A[*j] = aux;

            // Incrementa o contador de passos por cada troca realizada
            passos++;
            (*i)++;
            (*j)--;

            // Imprime o vetor após a troca (opcional, para visualizar os passos)
            printf("Passo %d: ", passos);
            for (int k = 0; k < Dir + 1; k++) {
                printf("%d ", A[k].Chave);
            }
            printf("\n");
        }
    } while (*i <= *j);  // Continua enquanto os índices não se cruzarem
}

// Função recursiva para ordenar as duas metades
void Ordena(int Esq, int Dir, Item *A) {
    int i, j;

    if (Esq < Dir) {
        // Chama a função de particionamento
        Particao(Esq, Dir, &i, &j, A);

        // Chama recursivamente para as duas metades
        Ordena(Esq, j, A);
        Ordena(i, Dir, A);
    }
}

// Função principal do QuickSort
void QuickSort(Item *A, int n) {
    Ordena(0, n - 1, A);  // Ordena o vetor inteiro
}

int main() {
    Item A[] = {{25}, {57}, {48}, {37}, {12}, {92}, {86}, {33}}; // Vetor de exemplo
```

```
    int n = sizeof(A) / sizeof(A[0]);

    printf("Vetor original:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", A[i].Chave);
    }
    printf("\n");

    // Chama o QuickSort e conta os passos
    QuickSort(A, n);

    // Exibe o vetor ordenado
    printf("\nVetor ordenado:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", A[i].Chave);
    }
    printf("\n");

    // Exibe o total de passos
    printf("\nTotal de passos: %d\n", passos);

    return 0;
}
```

Saída:



6. Heap sort:

Lógica:

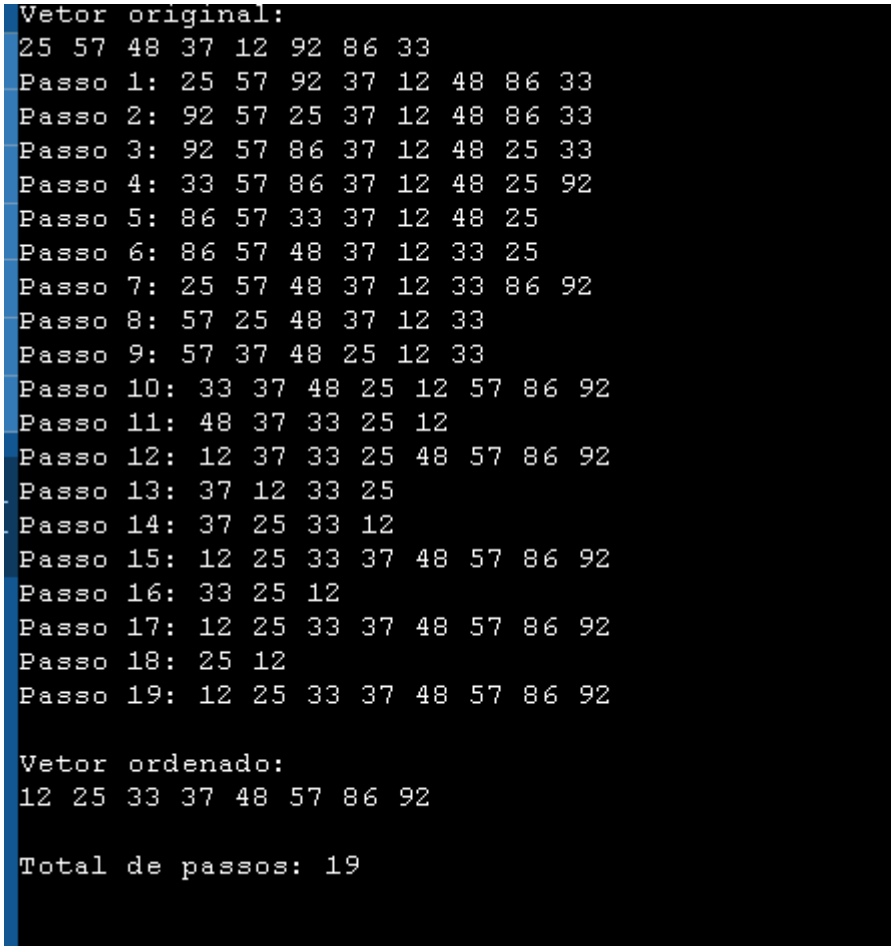
O **Heap Sort** utiliza uma estrutura de dados chamada **heap** (ou montículo), que é uma árvore binária onde o valor de um nó é maior ou menor que o valor dos seus filhos. O algoritmo constrói um heap máximo (ou mínimo), em que o maior (ou menor) valor está na raiz, e então move esse valor para a última posição do vetor e ajusta o heap para manter sua propriedade. Este processo é repetido até que o vetor esteja ordenado.

Código:

```
#include <stdio.h>

#define MAX 100

int passos = 0;  // Variável global para contar os passos

// Função para ajustar o heap
void AjustarHeap(int n, int V[], int i) {
    int maior = i;  // Inicializa o maior como a raiz
    int esq = 2 * i + 1;  // Filho à esquerda
    int dir = 2 * i + 2;  // Filho à direita

    // Verifica se o filho à esquerda existe e é maior que a raiz
    if (esq < n && V[esq] > V[maior]) {
        maior = esq;
    }

    // Verifica se o filho à direita existe e é maior que a raiz (ou filho à esquerda)
    if (dir < n && V[dir] > V[maior]) {
        maior = dir;
    }

    // Se o maior não for a raiz, troca e ajusta o heap
    if (maior != i) {
        int temp = V[i];
        V[i] = V[maior];
        V[maior] = temp;

        passos++;  // Contabiliza o passo (troca de elementos)

        printf("Passo %d: ", passos);
        for (int k = 0; k < n; k++) {
            printf("%d ", V[k]);
        }
        printf("\n");
```

```
        // Chama recursivamente para ajustar o heap
        AjustarHeap(n, V, maior);
    }
}

// Função para construir o heap a partir de um vetor
void FormarFilaPrioridades(int n, int V[]) {
    // Começa do último nó não folha e ajusta o heap
    for (int i = n / 2 - 1; i >= 0; i--) {
        AjustarHeap(n, V, i);
    }
}

// Função para ordenar usando o HeapSort
void HeapSort(int n, int V[]) {
    // Constrói o heap
    FormarFilaPrioridades(n, V);

    // Um por um, extrai os elementos do heap
    for (int i = n - 1; i > 0; i--) {
        // Troca o topo (maior elemento) com o último elemento
        int temp = V[0];
        V[0] = V[i];
        V[i] = temp;

        passos++;  // Contabiliza o passo (troca de elementos)

        printf("Passo %d: ", passos);
        for (int k = 0; k < n; k++) {
            printf("%d ", V[k]);
        }
        printf("\n");

        // Ajusta o heap para o vetor restante
        AjustarHeap(i, V, 0);
    }
}

// Função para imprimir o vetor
void ImprimirVetor(int V[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", V[i]);
    }
    printf("\n");
}

int main() {
    int V[] = {25, 57, 48, 37, 12, 92, 86, 33};
```

```
    int n = sizeof(V) / sizeof(V[0]);

    printf("Vetor original:\n");
    ImprimirVetor(V, n);

    // Chama o HeapSort
    HeapSort(n, V);

    printf("\nVetor ordenado:\n");
    ImprimirVetor(V, n);

    printf("\nTotal de passos: %d\n", passos);

    return 0;
}
```



2. Análise de Complexidade

1. Selection Sort

- ### Complexidade de Tempo no Melhor Caso:

- O **Selection Sort** sempre percorre todo o array para encontrar o menor elemento em cada iteração. Mesmo que o array já esteja ordenado, ele ainda fará todas as comparações, resultando em **O(n²)**. Portanto, o melhor caso é **O(n²)**.

- ### Complexidade de Tempo no Pior Caso:

- No pior caso, o comportamento do **Selection Sort** é idêntico ao do melhor caso. Ele sempre faz **n** comparações para encontrar o menor elemento, independentemente da ordem dos dados. A complexidade de tempo no pior caso é **O(n²)**.

- ### Complexidade de Tempo no Caso Médio:

- Assim como no melhor e no pior caso, o **Selection Sort** sempre faz o mesmo número de comparações, independentemente da ordem inicial dos elementos. Portanto, a complexidade no caso médio também é **O(n²)**.

- ### Complexidade de Espaço:

- **O(1)**, pois é um algoritmo **in-place**, ou seja, a ordenação é feita diretamente no array, sem a necessidade de estruturas de dados adicionais.

---

2. Insertion Sort

- ### Complexidade de Tempo no Melhor Caso:

- O melhor cenário ocorre quando o array já está ordenado. O **Insertion Sort** apenas percorre os elementos uma vez, sem realizar muitas trocas, resultando em uma complexidade de **O(n)**.

- ### Complexidade de Tempo no Pior Caso:

- O pior cenário ocorre quando o array está ordenado de forma inversa, o que exige que cada novo elemento inserido seja comparado com todos os anteriores. Nesse caso, a complexidade é **O(n²)**, pois o número de comparações e trocas aumenta quadraticamente.

- ### Complexidade de Tempo no Caso Médio:

- No caso médio, os elementos estão em uma ordem aleatória, o que faz com que o número de comparações e trocas varie, mas ainda seja proporcional ao quadrado do número de elementos. A complexidade no caso médio é **O(n²)**.

- ### Complexidade de Espaço:

- **O(1)**, já que o **Insertion Sort** é **in-place**, utilizando apenas uma variável temporária para manter o valor do elemento sendo inserido.

---

3. Bubble Sort

- ### Complexidade de Tempo no Melhor Caso:

- O melhor caso ocorre quando o array já está ordenado. Com uma pequena otimização (que detecta se houve trocas em uma passagem), o **Bubble Sort** pode parar após uma única passagem, resultando em **O(n)**.

- ### Complexidade de Tempo no Pior Caso:

- No pior cenário, onde os elementos estão em ordem inversa, o **Bubble Sort** precisa fazer todas as comparações e trocas possíveis, resultando em uma complexidade de **O(n²)**.

- ### Complexidade de Tempo no Caso Médio:

- Em média, o **Bubble Sort** também precisa fazer várias comparações e trocas, mas de forma menos intensa que no pior caso. No entanto, a complexidade no caso médio ainda é **O(n²)**.

- ### Complexidade de Espaço:

- **O(1)**, pois o algoritmo realiza a ordenação diretamente no array, sem a necessidade de espaço adicional significativo.

---

4. Merge Sort

- ### Complexidade de Tempo no Melhor Caso:

- O **Merge Sort** sempre divide o array ao meio e, em seguida, mescla os subarrays ordenados. Como esse processo é independente da ordem dos dados, a complexidade é sempre **O(n log n)**, mesmo no melhor caso.

- ### Complexidade de Tempo no Pior Caso:

- Assim como no melhor caso, a complexidade no pior caso também é **O(n log n)**, já que a estratégia de divisão e mesclagem permanece constante, independentemente da entrada.

- ### Complexidade de Tempo no Caso Médio:

- O caso médio do **Merge Sort** é o mesmo que o melhor e o pior caso, pois o número de divisões e o processo de mesclagem são sempre consistentes, resultando em **O(n log n)**.

- ### Complexidade de Espaço:

- **O(n)**, pois o **Merge Sort** necessita de arrays temporários para armazenar os subarrays durante o processo de mesclagem, resultando em um uso de memória proporcional ao tamanho do array original.

---

5. Quick Sort

- ### Complexidade de Tempo no Melhor Caso:

- O melhor cenário ocorre quando o pivô escolhido divide o array de maneira equilibrada em cada recursão, resultando em **O(n log n)**, pois, a cada divisão, o array é dividido pela metade, e para cada nível de recursão são feitas **n** comparações.

- ### Complexidade de Tempo no Pior Caso:

- O pior cenário ocorre quando o pivô escolhido é sempre o maior ou o menor elemento, resultando em uma divisão extremamente desigual (subarrays de tamanho 1 e n-1). Neste caso, a complexidade é **O(n^2)**.

- ### Complexidade de Tempo no Caso Médio:

- Em média, o pivô divide o array de forma relativamente equilibrada, levando a uma complexidade de **O(n log n)**, semelhante ao melhor caso.

- ### Complexidade de Espaço:

- **O(log n)** no melhor caso, devido à profundidade da recursão. No pior caso, se a recursão não for balanceada, a profundidade pode ser até **O(n)**, resultando em maior uso de memória para a pilha de chamadas recursivas.

---

6. Heap Sort

- ### Complexidade de Tempo no Melhor Caso:

- Assim como no pior e no caso médio, o **Heap Sort** tem uma complexidade de tempo **O(n log n)** em todos os cenários, já que construir o heap leva **O(n)** e reorganizar os elementos em torno do heap leva **O(log n)**.

- ### Complexidade de Tempo no Pior Caso:

- No pior caso, o algoritmo ainda tem a mesma complexidade de **O(n log n)**, pois o processo de reorganização do heap não depende da ordem inicial dos dados.

- ### Complexidade de Tempo no Caso Médio:

- No caso médio, a complexidade do **Heap Sort** continua sendo **O(n log n)**, pois o processo de construção do heap e de extração dos elementos segue o mesmo padrão.

- ### Complexidade de Espaço:

- **O(1)**, pois o **Heap Sort** é um algoritmo **in-place**, que usa o array original para organizar os elementos em forma de heap, sem a necessidade de espaço extra além de algumas variáveis auxiliares.

Resumo de Complexidades:

Algoritmo	Melhor Caso	Pior Caso	Caso Médio	Espaço
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	$O(n^2)$	O(n log n)	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)

Explicação de complexibilidade

- **O(n log n)**

  Este tipo de complexidade ocorre quando o tempo de execução de um algoritmo cresce em proporção ao tamanho da entrada (n), multiplicado pelo logaritmo de **n** (log n). O logaritmo aqui geralmente é base 2, porque muitos algoritmos que envolvem divisões sucessivas do problema pela metade (como o **Merge Sort** e o **Quick Sort** no melhor caso) têm essa característica.
  Algoritmos com O(n log n): **Merge Sort**, **Quick Sort** (caso médio e melhor), **Heap Sort**.

- **O(n)**

  A complexidade **O(n)** indica que o tempo de execução do algoritmo cresce linearmente com o tamanho da entrada. Se você dobrar a entrada, o tempo de execução também dobra. O algoritmo precisa fazer uma operação em cada elemento da entrada.
  Algoritmos com O(n): **Insertion Sort** no melhor caso (quando o array já está ordenado), **pesquisa linear**.

- **O(1)**

  A complexidade **O(n²)** ocorre quando o tempo de execução do algoritmo cresce **quadraticamente** em relação ao tamanho da entrada. Isso significa que, se você dobrar o tamanho da entrada, o tempo de execução aumenta quatro vezes (2^2), ou seja, cresce muito rapidamente à medida que o tamanho da entrada aumenta.
  Algoritmos com O(1): Algoritmos que realizam operações fixas ou que não dependem da entrada, como a atribuição de valores a variáveis ou acessos diretos em arrays.

- **O(n²)**

  A complexidade **O(n²)** ocorre quando o tempo de execução do algoritmo cresce **quadraticamente** em relação ao tamanho da entrada. Isso significa que, se você dobrar o tamanho da entrada, o tempo de execução aumenta quatro vezes (2^2), ou seja, cresce muito rapidamente à medida que o tamanho da entrada aumenta.
  Algoritmos com O(n²): **Bubble Sort**, **Selection Sort**, **Insertion Sort** (pior caso).

Comparação entre essas complexidades

- **O(1)** é a melhor complexidade, pois o tempo de execução é constante.
- **O(n)** é mais eficiente que **O(n²)**, pois cresce linearmente em vez de quadraticamente.
- **O(n log n)** é comum em algoritmos eficientes de divisão e conquista, como o **Merge Sort**.
- **O(n²)** é menos eficiente e ocorre em algoritmos que envolvem muitas comparações ou trocas, como **Bubble Sort** e **Selection Sort**.

3. Testes de Performance:

Nesta etapa faremos uma pequena alteração nos códigos apresentados acima, utilizaremos a biblioteca do c <time.h> para cronometrar a performance de cada método de ordenação com diferentes tamanhos de array (100, 1000, 10000, 50000, 100000). Os tempos mostrados são variáveis, pois cada vez que rodamos o código os números inseridos no vetor são randômicos, e por isto, nem sempre dão os mesmo valores, eles apenas nos dão uma noção básica da performance de cada método.

Bubble Sort:

Código:

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

void bubblesort(int *v, int tam) {
    int i, j, aux;
    for (i = 0; i < tam - 1; i++) {
        for (j = 1; j < tam - i; j++) {
            if (v[j] < v[j - 1]) {
                aux = v[j];
                v[j] = v[j - 1];
                v[j - 1] = aux;
            }
        }
    }
}

void generate_random_array(int *v, int tam) {
    for (int i = 0; i < tam; i++) {
        v[i] = rand() % 100000;
    }
}

int main() {
    srand(time(NULL));

    int sizes[] = {100, 1000, 10000, 50000, 100000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    for (int s = 0; s < num_sizes; s++) {
        int tam = sizes[s];
        int *v = (int *)malloc(tam * sizeof(int));
        if (v == NULL) {
            printf("Erro ao alocar memória para o tamanho %d\n", tam);
```

```
            return 1;
        }

        generate_random_array(v, tam);

        clock_t start = clock();
        bubblesort(v, tam);
        clock_t end = clock();

        double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("Tempo para ordenar %d elementos: %.4f segundos\n", tam, time_taken);

        free(v);
    }

    return 0;
}
```

Saída:

```
Tempo para ordenar 100 elementos: 0.0000 segundos
Tempo para ordenar 1000 elementos: 0.0022 segundos
Tempo para ordenar 10000 elementos: 0.2844 segundos
Tempo para ordenar 50000 elementos: 7.7273 segundos
Tempo para ordenar 100000 elementos: 31.0838 segundos
```

Select Sort:

Código:

select sort, procurando o maior

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

// Função de ordenação por seleção (Selection Sort)
void selectsort(int *v, int tam) {
    int i, j, aux;
    int cont = 1;

    // Ordena o vetor pela técnica de Selection Sort
    for (i = tam - 1; i > 0; i--) {
        int maior = 0;  // Inicializa o índice do maior elemento como o primeiro da parte não
ordenada
        for (j = 1; j <= i; j++) { // Percorre todo o vetor
            if (v[j] > v[maior]) { // Verifica se o valor atual é maior que o valor do maior
                maior = j;  // Se for, atualiza o maior índice
```

```
            }
        }

        // Troca o maior elemento encontrado com o último elemento da parte não ordenada
        aux = v[maior];
        v[maior] = v[i];
        v[i] = aux;

        // Exibe o vetor após cada passo (opcional para visualizar a ordenação)
        // printf("Passo %d: ", cont); // Imprime a interação
        // for (int k = 0; k < tam; k++) { // Corrigido para não usar a mesma variável i
        //     printf("%d ", v[k]);
        // }
        // printf("\n");
        cont++;
    }
}

// Função para imprimir o vetor
void print_array(int *v, int tam) {
    for (int i = 0; i < tam; i++) {
        printf("%d ", v[i]);
    }
    printf("\n");
}

int main() {
    // Diferentes tamanhos de array
    int sizes[] = {100, 1000, 10000, 50000, 100000};
    int n_tests = sizeof(sizes) / sizeof(sizes[0]);

    // Realiza o teste para cada tamanho de array
    for (int t = 0; t < n_tests; t++) {
        int tam = sizes[t];
        int *v = (int *)malloc(tam * sizeof(int));  // Aloca memória para o array de tamanho `tam`

        // Preenche o array com números aleatórios
        srand(time(NULL));
        for (int i = 0; i < tam; i++) {
            v[i] = rand() % 100000;  // Preenche com números aleatórios de 0 a 99999
        }

        // Exibe o tamanho do vetor
        printf("\nTestando com %d elementos:\n", tam);

        // Mede o tempo de execução do Selection Sort
        clock_t start_time = clock();
        selectsort(v, tam);  // Chama a função de ordenação
```

```
        clock_t end_time = clock();

        double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
        printf("Tempo de execução para ordenar %d elementos: %f segundos\n", tam,
time_taken);

        // Descomente para visualizar o vetor ordenado
        // printf("Vetor ordenado:\n");
        // print_array(v, tam);

        // Libera a memória alocada
        free(v);
    }

    return 0;
}
```

Saída:

```
Tempo de execução para 100 elementos: 0.0000 segundos
Tempo de execução para 1000 elementos: 0.0017 segundos
Tempo de execução para 10000 elementos: 0.1084 segundos
Tempo de execução para 50000 elementos: 2.6461 segundos
Tempo de execução para 100000 elementos: 9.5440 segundos
```

Select sort, procurando menor:

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

// Função de ordenação por seleção (Selection Sort)
void select_sort(int *v, int tam) {
    int i, j, min;
    int aux;
    int cont = 1;

    // Ordena o vetor pela técnica de Selection Sort
    for (i = 0; i < tam - 1; i++) { // Laço de interação externo
        min = i; // Inicializa o primeiro como o menor
        for (j = i + 1; j < tam; j++) { // Laço interno
            if (v[j] < v[min]) { // Compara o j atual com o valor do menor
                min = j; // Se o valor menor for maior que o j atual, o menor passa a ser o j
            }
```

```
        }
        // Faz a troca para o menor ficar no começo
        aux = v[min];
        v[min] = v[i];
        v[i] = aux;

        // Exibe o vetor após cada passo (opcional para visualizar a ordenação)
        // printf("Passo %d: ", cont); // Imprime a interação
        // for (int k = 0; k < tam; k++) {
        //     printf("%d ", v[k]);
        // }
        // printf("\n");
        cont++;
    }
}

// Função para imprimir o vetor
void print_array(int *v, int tam) {
    for (int i = 0; i < tam; i++) {
        printf("%d ", v[i]);
    }
    printf("\n");
}

int main() {
    // Diferentes tamanhos de array
    int sizes[] = {100, 1000, 10000, 50000, 100000};
    int n_tests = sizeof(sizes) / sizeof(sizes[0]);

    // Realiza o teste para cada tamanho de array
    for (int t = 0; t < n_tests; t++) {
        int tam = sizes[t];
        int *v = (int *)malloc(tam * sizeof(int));  // Aloca memória para o array de tamanho `tam`

        // Preenche o array com números aleatórios
        srand(time(NULL));
        for (int i = 0; i < tam; i++) {
            v[i] = rand() % 100000;  // Preenche com números aleatórios de 0 a 99999
        }

        // Exibe o tamanho do vetor
        printf("\nTestando com %d elementos:\n", tam);

        // Mede o tempo de execução do Selection Sort
        clock_t start_time = clock();
        select_sort(v, tam);  // Chama a função de ordenação
        clock_t end_time = clock();
```

```
        double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
        printf("Tempo de execução para ordenar %d elementos: %f segundos\n", tam,
time_taken);

        // Descomente para visualizar o vetor ordenado
        // printf("Vetor ordenado:\n");
        // print_array(v, tam);

        // Libera a memória alocada
        free(v);
    }

    return 0;
}
```

Saída:

```
Testando com 100 elementos:
  Tempo de execução para ordenar 100 elementos: 0.000018 segundos

Testando com 1000 elementos:
  Tempo de execução para ordenar 1000 elementos: 0.001394 segundos

Testando com 10000 elementos:
  Tempo de execução para ordenar 10000 elementos: 0.126500 segundos

Testando com 50000 elementos:
  Tempo de execução para ordenar 50000 elementos: 3.168856 segundos

Testando com 100000 elementos:
  Tempo de execução para ordenar 100000 elementos: 12.638273 segundos
```

Insertion Sort:

código:

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

void insertionsort(int *v, int tam) {
    int i, j, aux;
    for (i = 1; i < tam; i++) {
        aux = v[i];
        j = i - 1;
        while (j >= 0 && v[j] > aux) {
            v[j + 1] = v[j];
            j--;
        }
        v[j + 1] = aux;
    }
```

```
}

void gerar_dados(int *v, int tam) {
    for (int i = 0; i < tam; i++) {
        v[i] = rand() % 100000;
    }
}

int main() {
    srand(time(NULL));
    int tamanhos[] = {100, 1000, 10000, 50000, 100000};
    int n_tamanhos = sizeof(tamanhos) / sizeof(tamanhos[0]);

    for (int i = 0; i < n_tamanhos; i++) {
        int tam = tamanhos[i];
        int *v = malloc(tam * sizeof(int));

        gerar_dados(v, tam);
        clock_t inicio = clock();
        insertionsort(v, tam);
        clock_t fim = clock();

        double tempo_execucao = (double)(fim - inicio) / CLOCKS_PER_SEC;
        printf("    Tempo de execução para %d elementos: %.4f segundos\n", tam,
tempo_execucao);

        free(v);
    } return 0;
}
```

Saída:

```
Tempo de execução para 100 elementos: 0.0000 segundos
Tempo de execução para 1000 elementos: 0.0010 segundos
Tempo de execução para 10000 elementos: 0.0700 segundos
Tempo de execução para 50000 elementos: 1.7655 segundos
Tempo de execução para 100000 elementos: 6.8817 segundos
```

Marge Sort:

Código:

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

void merge(int vetor[], int inicio, int meio, int fim) {
    int tamanho = fim - inicio + 1;
    int temp[tamanho];
    int i = inicio, j = meio + 1, k = 0;

    while (i <= meio && j <= fim) {
        if (vetor[i] < vetor[j]) {
            temp[k++] = vetor[i++];
        } else {
            temp[k++] = vetor[j++];
        }
    }

    while (i <= meio) {
        temp[k++] = vetor[i++];
    }

    while (j <= fim) {
        temp[k++] = vetor[j++];
    }

    for (k = 0, i = inicio; i <= fim; i++, k++) {
        vetor[i] = temp[k];
    }
}

void mergeSort(int vetor[], int inicio, int fim) {
    if (inicio < fim) {
        int meio = (inicio + fim) / 2;
        mergeSort(vetor, inicio, meio);
        mergeSort(vetor, meio + 1, fim);
        merge(vetor, inicio, meio, fim);
    }
}

void gerar_dados(int *v, int tam) {
    for (int i = 0; i < tam; i++) {
        v[i] = rand() % 100000;
    }
}

int main() {
    srand(time(NULL));
    int tamanhos[] = {100, 1000, 10000, 50000, 100000};
```

```
    int n_tamanhos = sizeof(tamanhos) / sizeof(tamanhos[0]);

    for (int i = 0; i < n_tamanhos; i++) {
        int tam = tamanhos[i];
        int *v = malloc(tam * sizeof(int));

        gerar_dados(v, tam);
        clock_t inicio = clock();
        mergeSort(v, 0, tam - 1);
        clock_t fim = clock();

        double tempo_execucao = (double)(fim - inicio) / CLOCKS_PER_SEC;
        printf("    Tempo de execução para %d elementos: %.4f segundos\n", tam,
tempo_execucao);

        free(v);
    }

    return 0;
}
```

Saída:

```
Tempo de execução para 100 elementos: 0.0000 segundos
Tempo de execução para 1000 elementos: 0.0001 segundos
Tempo de execução para 10000 elementos: 0.0014 segundos
Tempo de execução para 50000 elementos: 0.0082 segundos
Tempo de execução para 100000 elementos: 0.0166 segundos
```

Quick Sort:

Código:

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

```
#include <stdlib.h>
#include <time.h>

typedef struct {
    int Chave;
} Item;

void Particao(int Esq, int Dir, int *i, int *j, Item *A) {
    Item x, aux;
    *i = Esq;
    *j = Dir;
    x = A[(*i + *j) / 2];
    do {
        while (x.Chave > A[*i].Chave) (*i)++;
        while (x.Chave < A[*j].Chave) (*j)--;

        if (*i <= *j) {

            aux = A[*i];
            A[*i] = A[*j];
            A[*j] = aux;

            (*i)++;
            (*j)--;

        }
    } while (*i <= *j);
}

void Ordena(int Esq, int Dir, Item *A) {
    int i, j;

    if (Esq < Dir) {
        Particao(Esq, Dir, &i, &j, A);
        Ordena(Esq, j, A);
        Ordena(i, Dir, A);
    }
}
void QuickSort(Item *A, int n) {
    Ordena(0, n - 1, A);
}
void GerarVetorAleatorio(Item *A, int n) {
    for (int i = 0; i < n; i++) {
        A[i].Chave = rand() % 1000000;
    }
}

int main() {
```

```
    int tamanhos[] = {100, 1000, 10000, 50000, 100000};
    int num_testes = sizeof(tamanhos) / sizeof(tamanhos[0]);

    for (int t = 0; t < num_testes; t++) {
        int n = tamanhos[t];
        Item *A = (Item *)malloc(n * sizeof(Item));

        GerarVetorAleatorio(A, n);

        clock_t inicio = clock();

        QuickSort(A, n);

        clock_t fim = clock();
        double tempo_execucao = ((double)(fim - inicio)) / CLOCKS_PER_SEC;

        printf("    Tempo de execução para %d elementos: %.6f segundos\n", n,
tempo_execucao);

        free(A);
    }

    return 0;}
```

Saída:

```
Tempo de execução para 100 elementos: 0.000010 segundos
Tempo de execução para 1000 elementos: 0.000139 segundos
Tempo de execução para 10000 elementos: 0.001482 segundos
Tempo de execução para 50000 elementos: 0.006768 segundos
Tempo de execução para 100000 elementos: 0.015165 segundos
```

Heap Sort:

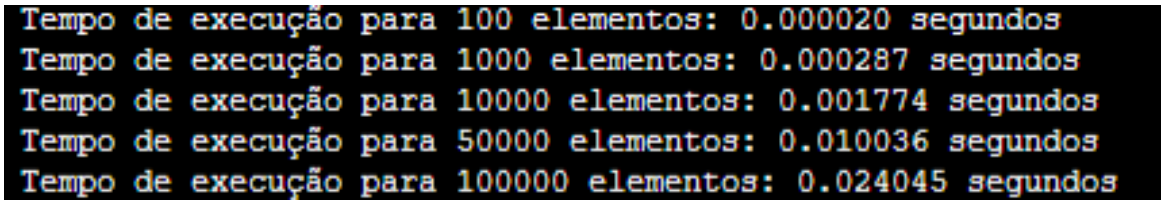
Código:

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

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

void AjustarHeap(int n, int V[], int i) {
    int maior = i;
    int esq = 2 * i + 1;
    int dir = 2 * i + 2;

    if (esq < n && V[esq] > V[maior]) {
        maior = esq;
    }

    if (dir < n && V[dir] > V[maior]) {
        maior = dir;
    }

    if (maior != i) {
        int temp = V[i];
        V[i] = V[maior];
        V[maior] = temp;

        AjustarHeap(n, V, maior);
    }
}

void FormarFilaPrioridades(int n, int V[]) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        AjustarHeap(n, V, i);
    }
}

void HeapSort(int n, int V[]) {
    FormarFilaPrioridades(n, V);

    for (int i = n - 1; i > 0; i--) {
        int temp = V[0];
        V[0] = V[i];
        V[i] = temp;

        AjustarHeap(i, V, 0);
    }
}

void GerarVetorAleatorio(int V[], int n) {
    for (int i = 0; i < n; i++) {
        V[i] = rand() % 1000000;
    }
}
```

```
void ImprimirVetor(int V[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", V[i]);
    }
    printf("\n");
}

int main() {
    int tamanhos[] = {100, 1000, 10000, 50000, 100000};
    int num_testes = sizeof(tamanhos) / sizeof(tamanhos[0]);

    for (int t = 0; t < num_testes; t++) {
        int n = tamanhos[t];
        int *V = (int *)malloc(n * sizeof(int));

        GerarVetorAleatorio(V, n);

        clock_t inicio = clock();

        HeapSort(n, V);

        clock_t fim = clock();
        double tempo_execucao = ((double)(fim - inicio)) / CLOCKS_PER_SEC;

        printf("    Tempo de execução para %d elementos: %.6f segundos\n", n,
tempo_execucao);

        free(V);
    }

    return 0;
}
```

Saida:



4.Análise Comparativa:

1. Desempenho dos Algoritmos para Diferentes Tamanhos
    - **Bubble Sort, Selection Sort e Insertion Sort** são algoritmos com complexidade **O(n²)** no pior caso, o que significa que, à medida que o tamanho da entrada aumenta, o tempo de execução cresce de forma significativa. Para pequenas entradas (menores que 100 elementos, por exemplo), o impacto é menos perceptível, mas para grandes entradas, como **1000, 10.000 ou 100.000 elementos**, esses algoritmos tornam-se extremamente lentos.
    - **Merge Sort, Quick Sort e Heap Sort**, por outro lado, possuem **complexidade O(n log n)** no pior caso (em média, para o **Quick Sort**, ou garantido no **Merge Sort** e **Heap Sort**), o que implica um crescimento mais suave do tempo de execução à medida que o número de elementos aumenta. Eles conseguem lidar com entradas muito maiores de forma mais eficiente, já que o tempo de execução não cresce tão rapidamente quanto os algoritmos O(n²).

2. Análise do Crescimento do Tempo de Execução
    - **Bubble Sort**: Para **entradas pequenas**, o **Bubble Sort** pode ter um desempenho razoável, especialmente se os dados estiverem quase ordenados. Contudo, **com entradas maiores**, o tempo de execução cresce drasticamente, devido ao seu **O(n²)**. Um vetor de 1000 elementos pode ser ordenado em segundos, mas um vetor de 100.000 elementos pode levar minutos.
    - **Selection Sort**: Como o **Bubble Sort**, o **Selection Sort** também tem **complexidade O(n²)**, e o tempo de execução cresce de maneira semelhante com o aumento do número de elementos. Sua principal vantagem sobre o Bubble Sort é que ele sempre realiza o mesmo número de trocas, o que pode ser uma vantagem em sistemas onde as trocas são caras.
    - **Insertion Sort**: O **Insertion Sort** tem um desempenho melhor que os dois anteriores quando a entrada já está parcialmente ordenada, pois seu tempo de execução pode se aproximar de **O(n)**. No entanto, para entradas **desordenadas**, a complexidade de **O(n²)** ainda é predominante, resultando em tempos de execução longos quando o número de elementos cresce para 10.000 ou mais.
    - **Merge Sort**: O **Merge Sort** tem **complexidade O(n log n)** em todos os casos. Mesmo para entradas muito grandes, o tempo de execução não aumenta tão rapidamente quanto os algoritmos quadráticos. Por exemplo, para **entradas de 100.000 elementos**, o tempo de execução do Merge Sort pode ser muito mais rápido que o do Bubble Sort, apesar de ter a desvantagem de exigir memória adicional.
    - **Quick Sort**: Embora o **Quick Sort** também tenha complexidade **O(n log n)** na maioria dos casos, seu desempenho prático é geralmente melhor que o do Merge Sort devido a menores sobrecargas de memória. Em **entradas grandes**, o **Quick Sort** pode ser significativamente mais rápido, desde que o pivô seja escolhido de forma eficiente. No entanto, ele pode atingir **O(n²)** no

pior caso, se o pivô escolhido for ruim (como em casos de vetores já ordenados ou quase ordenados).

- **Heap Sort**: Como o **Merge Sort**, o **Heap Sort** garante **O(n log n)** no pior caso, mas geralmente é mais lento na prática devido ao overhead da operação de heapify, embora não precise de memória adicional, ao contrário do **Merge Sort**.

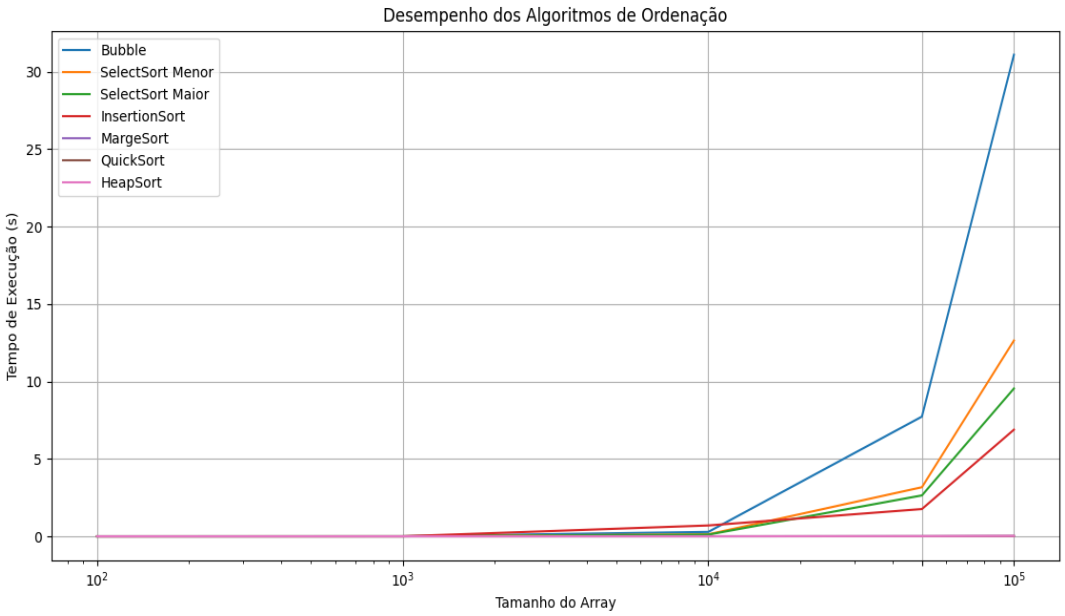
3. Situações em que Cada Algoritmo se SAI Melhor
    - **Pequenos Vetores ou Dados Quase Ordenados**: **Insertion Sort** ou **Bubble Sort** são opções viáveis. Se o vetor já está quase ordenado, o **Insertion Sort** pode ser especialmente eficiente, com desempenho de **O(n)**.
    - **Grandes Vetores**: **Merge Sort** ou **Quick Sort** são as melhores opções, com **Quick Sort** frequentemente oferecendo melhores resultados na prática, apesar do risco de seu pior caso. Se a estabilidade for necessária (quando a ordem de elementos iguais deve ser mantida), o **Merge Sort** é a escolha ideal.
    - **Vetor com Muitos Elementos e Memória Limitada**: **Heap Sort** ou **Quick Sort** são preferíveis, já que ambos podem operar sem o uso de memória adicional significativa. O **Heap Sort** é garantido para ter desempenho **O(n log n)** no pior caso, mas o **Quick Sort** é, em geral, mais rápido se bem implementado.

4. Diferença entre Desempenho Teórico e Prático Observado
    - **Overhead de constante** (operações extras que um algoritmo pode realizar independentemente de sua complexidade assintótica).
    - **Localidade de memória** (algoritmos como o **Quick Sort** tendem a ser mais rápidos na prática devido ao melhor uso da cache).
    - **Estrutura de dados subjacente** (alguns algoritmos podem ser mais eficientes dependendo da implementação interna e das otimizações do compilador).

    Por exemplo, o **Quick Sort** pode ser muito mais rápido que o **Merge Sort** na prática, apesar de ambos terem complexidade **O(n log n)**, devido ao menor uso de memória e ao fato de o Quick Sort frequentemente realizar menos trocas. Por outro lado, o **Heap Sort**, embora tenha a mesma complexidade assintótica que o **Quick Sort** no pior caso, pode ser mais lento devido ao overhead das operações de heapify, especialmente em implementações ineficientes.

5.Visualização de dados:

Para uma melhor visualização dos resultados gerados criamos alguns gráficos para uma melhor compreensão dos métodos



Como podemos analisar neste gráfico, os métodos de ordenação mais complexos tendem a demorarem menos à medida que o tamanho da array cresce, ao ponto que o HeapSort não chega a passar de 1 segundo mesmo tendo que ordenar uma array com 10000 elementos.

6.Conclusão:

Ao estudar e comparar os algoritmos de ordenação, fica claro que não existe um único "melhor" algoritmo, pois a escolha do algoritmo mais adequado depende de vários fatores, como o tamanho do vetor, a estrutura dos dados e as restrições de espaço e tempo. Abaixo, são destacadas as conclusões sobre quando usar cada algoritmo:

1. **Bubble Sort**:
    - **Melhor situação**: Quando o vetor é pequeno ou já está parcialmente ordenado. Embora o **Bubble Sort** tenha complexidade **O(n²)** no pior e no caso médio, ele pode ser eficiente quando o número de inversões é pequeno, já que em casos quase ordenados ele pode operar em **O(n)**.
    - **Desvantagens**: É ineficiente para grandes vetores e pode ser substituído por algoritmos mais rápidos em quase todos os cenários.
2. **Selection Sort**:
    - **Melhor situação**: Também para vetores pequenos e quando a simplicidade do algoritmo é uma prioridade. Embora tenha complexidade **O(n²)** em todos os casos, ele não exige memória extra e sempre faz o mesmo número de trocas (o que pode ser vantajoso em cenários onde as trocas são caras).
    - **Desvantagens**: A mesma complexidade **O(n²)** o torna inadequado para grandes volumes de dados.
3. **Insertion Sort**:
    - **Melhor situação**: Para listas pequenas ou quase ordenadas. O **Insertion Sort** é mais eficiente que o **Bubble Sort** e o **Selection Sort** quando o vetor já está parcialmente ordenado, já que pode operar em **O(n)** nesse caso.
    - **Desvantagens**: Seu desempenho se degrade para **O(n²)** à medida que o vetor cresce, tornando-o impraticável para grandes entradas de dados.
4. **Merge Sort**:
    - **Melhor situação**: Quando a estabilidade é necessária ou quando lidamos com grandes volumes de dados. O **Merge Sort** tem complexidade **O(n log n)** em todos os casos, garantindo um desempenho consistente, além de ser estável (não altera a ordem de elementos iguais).
    - **Desvantagens**: Requer **O(n)** de memória adicional, o que pode ser um problema quando o espaço de memória é limitado.
5. **Quick Sort**:
    - **Melhor situação**: Quando a performance no caso médio é crucial. O **Quick Sort** é muito eficiente na prática, com complexidade **O(n log n)** na maioria dos casos, sendo mais rápido que o **Merge Sort** em muitos cenários devido a menores sobrecargas de memória e operações de comparação.
    - **Desvantagens**: Seu pior caso é **O(n²)**, embora isso possa ser mitigado com boas estratégias de escolha de pivô (como o pivô aleatório ou mediano). Além disso, o algoritmo não é estável.

6. **Heap Sort**:
    - **Melhor situação**: Quando não se pode usar memória extra (como no caso do **Merge Sort**) e quando se precisa garantir uma complexidade **O(n log n)** no pior caso. O **Heap Sort** é uma boa escolha quando a eficiência no pior caso é uma prioridade.
    - **Desvantagens**: Ele pode ser mais lento que o **Quick Sort** na prática devido ao overhead das operações de heapify, embora seu desempenho seja garantido em termos de complexidade.

Resumo das melhores situações:

- **Para listas pequenas ou quase ordenadas**, **Insertion Sort** é o mais eficiente na prática, seguido de **Bubble Sort**.
- **Para listas grandes** com **memória suficiente**, **Quick Sort** geralmente oferece a melhor performance, seguido de **Merge Sort**.
- **Quando a estabilidade é essencial**, o **Merge Sort** é a escolha adequada.
- **Quando a memória é uma preocupação**, **Quick Sort** ou **Heap Sort** são mais recomendados.

7.Referências:

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009.

SEDGEWICK, Robert; WAYNE, Kevin. *Algorithms*. 4th ed. Addison-Wesley, 2011.

WEISS, Mark Allen. *Data Structures and Algorithm Analysis in C*. 3rd ed. Addison-Wesley, 2006.