SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE VIÇOSA · UFV
CAMPUS FLORESTAL

Trabalho 3 - AEDS 1

UNO: Ordenação de Cartas Desordenadas

Rafaella Ferreira [05363]
Luana Tavares[05364]
Aline Cristina [05791]

Florestal- MG
2023

SUMÁRIO

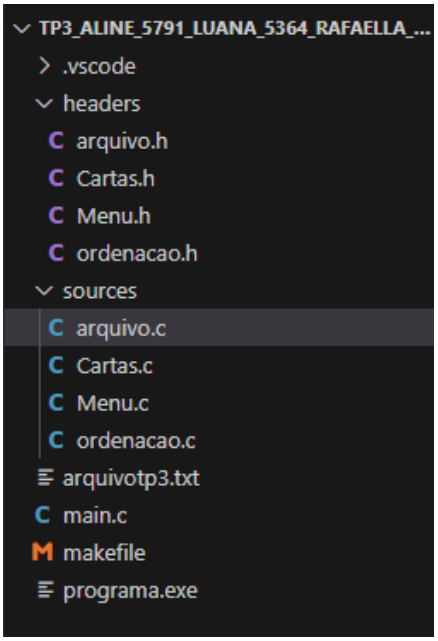
1. Introdução

O trabalho prático tem como objetivo a implementação e a análise de desempenho de seis algoritmos de ordenação clássicos, aplicados à ordenação de cartas do jogo UNO. O problema consiste em ordenar uma mão de jogador representada por uma lista de cartas embaralhadas, levando em consideração cores e valores.

O trabalho prático conta com a implementação dos algoritmos de ordenação, seguida da análise de desempenho com métricas como o número de comparações de chaves, número de movimentações de itens e o tempo total gasto para cada ordenação.

2. Organização

Fizemos a organização do código através de pastas. Separamos os arquivos .c na pasta sources e os arquivos .h na pasta headers. Optamos por fazer a divisão em pastas para obtermos melhor organização e controle do código que estamos criando. Ademais, a divisão do código em pastas é uma prática fundamental em qualquer desenvolvimento de software. Na imagem abaixo, mostramos como está a divisão. O arquivo main.c, makefile, programa.exe e arquivotp3.txt, foram deixados fora de ambas as pastas.



(Figura 1: Divisão das pastas do código).

3. Desenvolvimento

3.1 Algoritmos de ordenação

Foram implementados 6 algoritmos de ordenação no trabalho prático. Dentre eles temos: BubbleSort, SelectSort, InsertSort, ShellSort, QuickSort e HeapSort. A seguir, falaremos sobre como foram as suas implementações.

1. **BubbleSort -** É um algoritmo de ordenação que pode ser aplicado em Arrays e Listas dinâmicas. A posição atual é comparada com a próxima posição e, se a posição atual for maior que a posição posterior, é realizada a troca dos valores nessa posição. Caso contrário, não é realizada a troca, apenas passa-se para o próximo par de comparações. No código do trabalho prático foram usados os parâmetros: **Item*v** que representa o vetor de itens a ser ordenado; **int n** que é o número de elementos do vetor; **int*comparacoes** e **int*movimentacoes** que respectivamente servem para guardar o número de comparações e movimentações que a ordenação BubbleSort realiza. Dois loops aninhados percorrem o vetor de itens. O **loop externo(i)** controla o número de iterações necessárias para posicionar o maior elemento na posição correta, e o **loop interno(j)** compara elementos adjacentes e os troca quando estiverem fora da ordem. Ademais, foi criada uma função para auxiliar na chamada da função **BubbleSort** , ela é denominada **"ordenarCartasBubbleSort"** que encapsula a lógica específica de ordenação do jogo UNO usando o algoritmo BubbleSort.

```
void BubbleSort(Item* v, int n, int* comparacoes, int* movimentacoes) {
    int i, j;
    Item aux;

    for (i = 0; i < n - 1; i++) {
        for (j = 1; j < n - i; j++) {
            // Incrementa o contador de comparações
            (*comparacoes)++;

            if (v[j].Dado.cor < v[j - 1].Dado.cor ||
               (v[j].Dado.cor == v[j - 1].Dado.cor && v[j].Dado.valor < v[j - 1].Dado.valor)) {
                aux = v[j];
                v[j] = v[j - 1];
                v[j - 1] = aux;

                // Incrementa o contador de movimentações
                (*movimentacoes)++;
            }
        }
    }
}
```

(Figura 2: Função BubbleSort).

```c
void ordenarCartasBubbleSort(Carta vetorRetiradas[NUMERO_DE_CARTAS]) {
    int comparacoes = 0;
    int movimentacoes = 0;

    // Criar um vetor de itens para facilitar a ordenação
    Item vetorItens[NUMERO_DE_CARTAS];

    // Preencher o vetor de itens com as cartas a serem ordenadas
    for (int i = 0; i < NUMERO_DE_CARTAS; i++) {
        vetorItens[i].Dado = vetorRetiradas[i];
    }

    // Ordenar o vetor de itens usando o método de ordenação bolha
    BubbleSort(vetorItens, NUMERO_DE_CARTAS, &comparacoes, &movimentacoes);

    // Atualizar o vetor de cartas ordenado
    for (int i = 0; i < NUMERO_DE_CARTAS; i++) {
        vetorRetiradas[i] = vetorItens[i].Dado;
    }
    mostrarVetorCartas(vetorRetiradas);
    // Mostrar o número de comparações e movimentações
    printf("\nNúmero de comparações: %d\n", comparacoes);
    printf("Número de movimentações: %d\n", movimentacoes);
}
```

(Figura 3: Função ordenarCartasBubbleSort).

2. **SelectSort -** O algoritmo de ordenação select sort é baseado em selecionar o menor elemento da sequência e colocar na primeira posição do array. A ideia é fazer esses dois passos até ordenar o array na ordem desejada. Ideal para pequenos arquivos e também é muito utilizado para arquivos com registros muito grandes. No código do trabalho prático, iniciamos a função "**ordenarCartasSelectSort**", que foi utilizada como uma auxiliar na "**SelectSort**", criando dois contadores (comparações e movimentações), para contar as movimentações feitas pela função. Após isso, criamos o vetor de itens para facilitar a ordenação e logo em seguida preenchemos o vetor de itens com as cartas de UNO que devemos ordenar. Em seguida, começamos o laço de condição (IF), onde começamos de fato a ordenação de select, analisando as cores e os valores de carta carta de acordo com o seu nível de importância. Trocamos a lógica do select depois para podermos incrementar o contador nas movimentações feitas pelo método. Depois, atualizamos o vetor de cartas ordenado e mostramos ele no terminal de acordo com o princípio deste algoritmo de ordenação.

```c
void SelectSort(Item* v, int n, int* comparacoes, int* movimentacoes) {
    for (int i = 0; i < n - 1; i++) {
        int indiceMin = i;
        for (int j = i + 1; j < n; j++) {
            (*comparacoes)++;
            if (v[j].Dado.cor < v[indiceMin].Dado.cor ||
                (v[j].Dado.cor == v[indiceMin].Dado.cor &&
                 v[j].Dado.valor < v[indiceMin].Dado.valor)) {
                indiceMin = j;
            }
        }
        if (indiceMin != i) {
            // Troca diretamente na lógica do Selection Sort
            Item temp = v[i];
            v[i] = v[indiceMin];
            v[indiceMin] = temp;
            // Incrementa o contador de movimentações
            (*movimentacoes)++;
        }
    }
}
```

(Figura 4: Função SelectSort).

```c
void ordenarCartasSelectSort(Carta vetorRetiradas[NUMERO_DE_CARTAS]) {
    int comparacoes = 0;
    int movimentacoes = 0;
    // Criar um vetor de itens para facilitar a ordenação
    Item vetorItens[NUMERO_DE_CARTAS];
    // Preencher o vetor de itens com as cartas a serem ordenadas
    for (int i = 0; i < NUMERO_DE_CARTAS; i++) {
        vetorItens[i].Dado = vetorRetiradas[i];
    }

    // Ordenar o vetor de itens usando o método de ordenação Selection Sort
    SelectSort(vetorItens, NUMERO_DE_CARTAS, &comparacoes, &movimentacoes);

    // Atualizar o vetor de cartas ordenado
    for (int i = 0; i < NUMERO_DE_CARTAS; i++) {
        vetorRetiradas[i] = vetorItens[i].Dado;
    }

    // Mostrar o vetor ordenado
    mostrarVetorCartas(vetorRetiradas);

    // Mostrar o número de comparações e movimentações
    printf("\nNúmero de comparações: %d\n", comparacoes);
    printf("Número de movimentações: %d\n", movimentacoes);
}
```

(Figura 5: Função ordenarCartasSelectSort).

3. **InsertSort-** Este código implementa o algoritmo de ordenação conhecido como Insertion Sort (Ordenação por Inserção). O algoritmo percorre o vetor a partir do segundo elemento (i = 1) até o último (i = n-1). Em cada iteração, o elemento atual (aux = v[i]) é comparado com os elementos à sua esquerda. O loop interno (`while`) compara a cor e o valor do elemento atual com os elementos à sua esquerda, deslocando os elementos à direita até encontrar a posição correta para inserir o elemento atual. Durante esse processo, os contadores de comparações e movimentações são incrementados conforme as comparações e movimentações de elementos são realizadas. O objetivo é contar o número de operações fundamentais para avaliar o desempenho do algoritmo. Ao final de cada iteração do loop externo, o elemento "aux" é inserido na posição correta no subvetor ordenado. O algoritmo continua esse processo até que todos os elementos estejam ordenados no vetor. O resultado é um vetor ordenado em ordem crescente com base nas propriedades definidas da ordenação. Além disso, uma função para auxiliar na chamada da função **Insercao** foi implementada, ela é denominada **"ordenarCartasInsercao"** que encapsula a lógica específica de ordenação do jogo UNO usando o algoritmo de Inserção.

```c
void Insercao(Item* v, int n, int* comparacoes, int* movimentacoes) {
    int i, j;
    Item aux;

    for (i = 1; i < n; i++) {
        aux = v[i];
        j = i - 1;
        (*comparacoes)++;  // Incrementa o contador de comparações

        while ((j >= 0) && (aux.Dado.cor < v[j].Dado.cor || (aux.Dado.cor == v[j].Dado.cor && aux.Dado.valor < v[j].Dado.valor))) {
            v[j + 1] = v[j];
            j--;
            (*movimentacoes)++;  // Incrementa o contador de movimentações
            if (j >= 0) {
                (*comparacoes)++;  // Incrementa o contador de comparações
            }
        }

        v[j + 1] = aux;
        (*movimentacoes)++;  // Incrementa o contador de movimentações
    }
}
```

(Figura 6: Função Insercao).

```c
void ordenarCartasInsercao(Carta vetorRetiradas[NUMERO_DE_CARTAS]) {
    // Criar um vetor de itens para facilitar a ordenação
    Item vetorItens[NUMERO_DE_CARTAS];

    // Preencher o vetor de itens com as cartas a serem ordenadas
    for (int i = 0; i < NUMERO_DE_CARTAS; i++) {
        vetorItens[i].Dado = vetorRetiradas[i];
    }

    // Inicializar os contadores
    int comparacoesInsercao = 0;
    int movimentacoesInsercao = 0;

    // Ordenar o vetor de itens usando o método de ordenação por inserção
    Insercao(vetorItens, NUMERO_DE_CARTAS, &comparacoesInsercao, &movimentacoesInsercao);

    // Atualizar o vetor de cartas ordenado
    for (int i = 0; i < NUMERO_DE_CARTAS; i++) {
        vetorRetiradas[i] = vetorItens[i].Dado;
    }

    mostrarVetorCartas(vetorRetiradas);

    // Mostrar o número de comparações e movimentações
    printf("\nNúmero de comparações: %d\n", comparacoesInsercao);
    printf("Número de movimentações: %d\n", movimentacoesInsercao);
}
```

(Figura 7: Função ordenarCartasInsercao).

4. **ShellSort -** O algoritmo funciona passando várias vezes pela lista dividindo o grupo maior em menores. Nos grupos menores é aplicado o método da ordenação por inserção. No código do trabalho prático, a função calcula o valor inicial de **'h'** de acordo com a sequência de ShellSort. O algoritmo então realiza uma série de ordenações, onde em cada passo, os elementos são comparados e trocados em intervalos de 'h'. Os contadores **'comparacoes'** e **'movimentacao'** são incrementados conforme ambas são feitas. O processo se repete até que **'h'** seja reduzido para 1, momento em que é aplicado um último passo do algoritmo de inserção. Uma função denominada "**ordenarCartasShellSort**" foi usada como auxiliar na função "ShellSort" . Seu papel é coordenar o processo de ordenação das cartas do jogo UNO usando o algoritmo de ordenação ShellSort.

```c
void Shellsort(Item* A, int n, int* comparacoes, int* movimentacoes) {
    int i, j;
    int h = 1;
    Item aux;

    do h = h * 3 + 1; while (h < n);

    do {
        h = h / 3;

        for (i = h; i < n; i++) {
            aux = A[i];
            j = i;

            (*comparacoes)++;  // Incrementa o contador de comparações

            while (A[j - h].Dado.cor > aux.Dado.cor ||
                   (A[j - h].Dado.cor == aux.Dado.cor && A[j - h].Dado.valor > aux.Dado.valor)) {
                A[j] = A[j - h];
                j -= h;

                (*movimentacoes)++;  // Incrementa o contador de movimentações

                if (j < h) break;
            }

            A[j] = aux;
        }
    } while (h != 1);
}
```

(Figura 8: Função ShellSort)

```c
void ordenarCartasShellSort(Carta vetorRetiradas[NUMERO_DE_CARTAS]) {
    // Criar um vetor de itens para facilitar a ordenação
    Item vetorItens[NUMERO_DE_CARTAS];

    // Preencher o vetor de itens com as cartas a serem ordenadas
    for (int i = 0; i < NUMERO_DE_CARTAS; i++) {
        vetorItens[i].Dado = vetorRetiradas[i];
    }

    // Inicializar os contadores
    int comparacoesShell = 0;
    int movimentacoesShell = 0;

    // Ordenar o vetor de itens usando o método de ordenação shell sort
    Shellsort(vetorItens, NUMERO_DE_CARTAS, &comparacoesShell, &movimentacoesShell);

    // Atualizar o vetor de cartas ordenado
    for (int i = 0; i < NUMERO_DE_CARTAS; i++) {
        vetorRetiradas[i] = vetorItens[i].Dado;
    }

    mostrarVetorCartas(vetorRetiradas);

    // Mostrar o número de comparações e movimentações
    printf("\nNúmero de comparações: %d\n", comparacoesShell);
    printf("Número de movimentações: %d\n", movimentacoesShell);
}
```

(Figura 9: Função ordenarCartasShellSort).

5. **QuickSort -** O método de ordenação QuickSort é um algoritmo de ordenação eficiente que é caracterizado pela divisão e pela conquista. O mesmo é mais eficiente que seus `'irmãos", o Heap e o Merge Sort, pois suas constantes são menores. No código do trabalho prático, utilizamos este método para ordenar um conjunto de cartas do baralho UNO. Começamos com a função "**particionar**", que seleciona o último elemento como o pivô e rearranja os elementos de forma que os menores fiquem à esquerda e os maiores à direita. A função "**QuickSort**", em si, utiliza da recursividade para ordenar sub listas tanto na direita quanto na esquerda do pivô. A função "**ordenarCartasQuickSort**", que faz o papel da função "**QuickSort**", inicializou os contadores, criou um vetor de itens para facilitar a ordenação e preencheu de cartas de UNO, e então chamamos o método de ordenação. Os contadores de comparações e de movimentações são exibidos , e o vetor ordenado é mostrado usando a função "**mostrarVetorCartas**".

```
int particionar(Item* A, int baixo, int alto, int* comparacoes, int* movimentacoes) {
    Item pivo = A[alto];
    int i = baixo - 1;

    for (int j = baixo; j < alto; j++) {
        (*comparacoes)++;
        if (A[j].Dado.cor < pivo.Dado.cor || (A[j].Dado.cor == pivo.Dado.cor && A[j].Dado.valor < pivo.Dado.valc
            i++;

            // Troca diretamente na lógica de particionar
            Item temp = A[i];
            A[i] = A[j];
            A[j] = temp;

            // Incrementa o contador de movimentações
            (*movimentacoes)++;
        }
    }
```

(Figura 10: Função particionar).

```
void quicksort(Item* A, int baixo, int alto, int* comparacoes, int* movimentacoes) {
    if (baixo < alto) {
        int indicePivo = particionar(A, baixo, alto, comparacoes, movimentacoes);
        quicksort(A, baixo, indicePivo - 1, comparacoes, movimentacoes);
        quicksort(A, indicePivo + 1, alto, comparacoes, movimentacoes);
    }
}
```

(Figura 11: Função quicksort).

```c
void ordenarCartasQuickSort(Carta vetorRetiradas[NUMERO_DE_CARTAS]) {
    int comparacoes = 0;
    int movimentacoes = 0;
    // Criar um vetor de itens para facilitar a ordenação
    Item vetorItens[NUMERO_DE_CARTAS];

    // Preencher o vetor de itens com as cartas a serem ordenadas
    for (int i = 0; i < NUMERO_DE_CARTAS; i++) {
        vetorItens[i].Dado = vetorRetiradas[i];
    }

    // Ordenar o vetor de itens usando o método de ordenação Quicksort
    quicksort(vetorItens, 0, NUMERO_DE_CARTAS - 1, &comparacoes, &movimentacoes);

    // Atualizar o vetor de cartas ordenado
    for (int i = 0; i < NUMERO_DE_CARTAS; i++) {
        vetorRetiradas[i] = vetorItens[i].Dado;
    }

    mostrarVetorCartas(vetorRetiradas);

    // Mostrar o número de comparações e movimentações
    printf("\nNúmero de comparações: %d\n", comparacoes);
    printf("Número de movimentações: %d\n", movimentacoes);
}
```

(Figura 12: Função ordenarCartasQuickSort).

6. **HeapSort -** Este código implementa o algoritmo de ordenação conhecido como Heapsort. O Heapsort possui duas funções principais: “heapify” e “heapSort”. A função heapify é responsável por transformar um array “ A”em uma heap. Uma heap é uma árvore binária especial em que o valor de cada nó é maior (ou menor, dependendo do tipo de heap) do que os valores de seus filhos. A função heapify é chamada recursivamente para ajustar a estrutura do heap. Os parâmetros comparações e movimentações são contadores que registram o número de comparações e movimentações realizadas durante o processo. A função heapSort utiliza a função heapify para criar uma heap máxima (no caso de Heapsort, onde o maior elemento é colocado na raiz) a partir do array de entrada A. Em seguida, o algoritmo extrai elementos da heap um por vez, colocando o maior elemento no final do array ordenado. O processo é repetido até que todos os elementos estejam no lugar correto. Adicionalmente, foi desenvolvida uma função denominada "ordenarCartasHeapSort" para facilitar a utilização do algoritmo de HeapSort. Esta função encapsula a lógica específica de ordenação do jogo UNO, simplificando a chamada e utilização do algoritmo no contexto de ordenação.

```c
void heapify(Item* A, int n, int i, int* comparacoes, int* movimentacoes) {
    int maior = i;
    int esquerda = 2 * i + 1;
    int direita = 2 * i + 2;

    // Incrementa o contador de comparações
    (*comparacoes)++;

    if (esquerda < n && (A[esquerda].Dado.cor > A[maior].Dado.cor ||
                         (A[esquerda].Dado.cor == A[maior].Dado.cor && A[esquerda].Dado.valor > A[maior].Dado.valor))) {
        maior = esquerda;
    }

    // Incrementa o contador de comparações
    (*comparacoes)++;

    if (direita < n && (A[direita].Dado.cor > A[maior].Dado.cor ||
                        (A[direita].Dado.cor == A[maior].Dado.cor && A[direita].Dado.valor > A[maior].Dado.valor))) {
        maior = direita;
    }

    // Incrementa o contador de comparações
    (*comparacoes)++;

    if (maior != i) {
        // Troca diretamente na lógica do heapify
        Item temp = A[i];
        A[i] = A[maior];
        A[maior] = temp;

        // Incrementa o contador de movimentações
        (*movimentacoes)++;
```

(Figura 13: Função heapify).

```c
void heapSort(Item* A, int n, int* comparacoes, int* movimentacoes) {
    // Constrói o heap (reorganiza o array)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(A, n, i, comparacoes, movimentacoes);
    }

    // Extrai um elemento por vez do heap
    for (int i = n - 1; i > 0; i--) {
        // Move a raiz atual para o final
        Item temp = A[0];
        A[0] = A[i];
        A[i] = temp;

        // Incrementa o contador de movimentações
        (*movimentacoes)++;

        // Chama a função heapify para o heap reduzido
        heapify(A, i, 0, comparacoes, movimentacoes);
    }
}
```

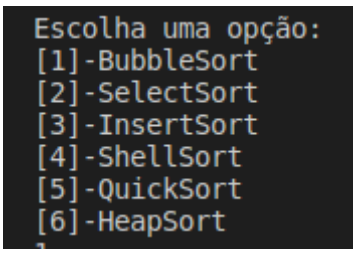
(Figura 14: Função heapSort).

```c
void ordenarCartasHeapSort(Carta vetorRetiradas[NUMERO_DE_CARTAS]) {
    int comparacoes = 0;
    int movimentacoes = 0;

    // Cria um vetor de itens para facilitar a ordenação
    Item vetorItens[NUMERO_DE_CARTAS];

    // Preenche o vetor de itens com as cartas a serem ordenadas
    for (int i = 0; i < NUMERO_DE_CARTAS; i++) {
        vetorItens[i].Dado = vetorRetiradas[i];
    }

    // Ordena o vetor de itens usando o método de ordenação Heap Sort
    heapSort(vetorItens, NUMERO_DE_CARTAS, &comparacoes, &movimentacoes);

    // Atualiza o vetor de cartas ordenado
    for (int i = 0; i < NUMERO_DE_CARTAS; i++) {
        vetorRetiradas[i] = vetorItens[i].Dado;
    }

    // Mostra o vetor ordenado
    mostrarVetorCartas(vetorRetiradas);

    // Mostra o número de comparações e movimentações
    printf("\nNúmero de comparações: %d\n", comparacoes);
    printf("Número de movimentações: %d\n", movimentacoes);
}
```

(Figura 15: Função ordenarCartasHeapSort).

3.2. Dificuldades

Encontramos algumas dificuldades ao longo do trabalho prático que gostaríamos de deixar documentadas. No modo interativo criamos 7 opções de visualização para o usuário, as 6 primeiras funcionam perfeitamente, geram os números de movimentação e comparação certos. Encontramos problema na opção 7, no qual o número de movimentações é sempre igual a 0 nas ordenações **ShellSort** e **SelectSort** quando a opção 7 é digitada. Não conseguimos resolver esse problema, e por esse motivo, decidimos colocar print das saídas uma a uma na seção 5.1.



(Figura 16: Opções de entrada).

Encontramos dificuldades e problemas relacionados ao modo arquivo, mas em específico um problema que não conseguimos resolver. Ao executar a função QuickSort suas movimentações e comparações permanecem 0.

4. Compilação e Execução

Para compilar e executar o nosso código, usamos o Makefile, pois ele nos permite executar todos os nossos arquivos de código mesmo não estando no mesmo diretório. Assim, criamos o Makefile no mesmo diretório onde estão seus arquivos de código-fonte.

- Para compilar o código, basta digitar **make all,** e em seguida digitar o nome que escolhemos para chamar os comandos(**./programa**).

```
all: main.c sources/Cartas.c sources/Menu.c sources/ordenacao.c sources/arquivo.c
    gcc -o programa main.c sources/Cartas.c sources/Menu.c sources/ordenacao.c  sources/arquivo.c
```

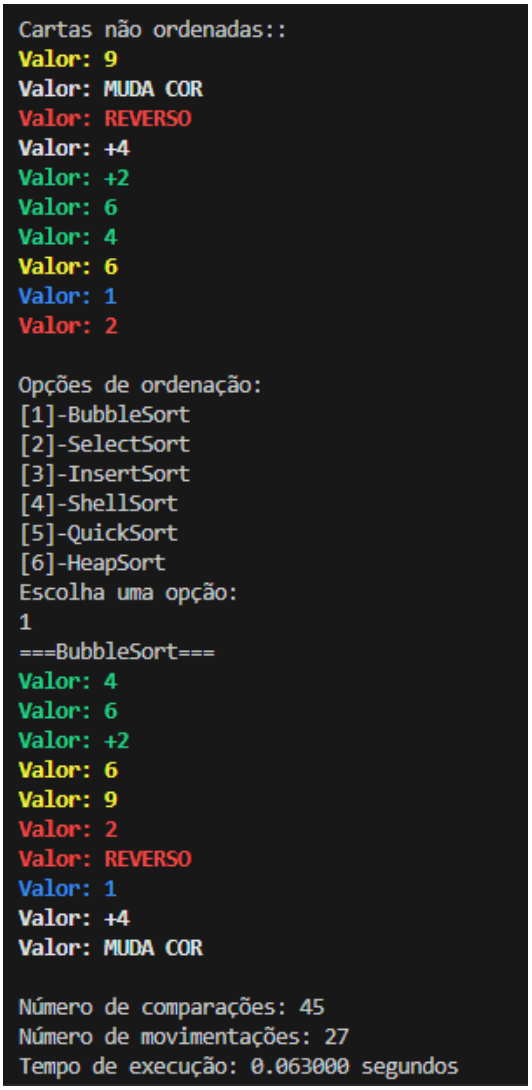
(Figura 17: Arquivo makefile; Comandos)

5. Resultados

Nesta etapa, serão exibidos os resultados obtidos ao executar o programa nos modos interativo e no arquivo. Para executar as saídas, usamos o notebook Lenovo Ideapad s145, equipado com o processador intel core i5, 4GB de memória RAM , um SSD de 1TB. Optamos por colocar saída por saída do terminal. Embora nosso código possua a opção de exibir todas as ordenações, estamos enfrentando problemas com o número de movimentação da ordenação Select Sort e Shell Sort. Ambas estão apontando esse número como 0.Por esse motivo, estamos exibindo as saídas uma a uma.

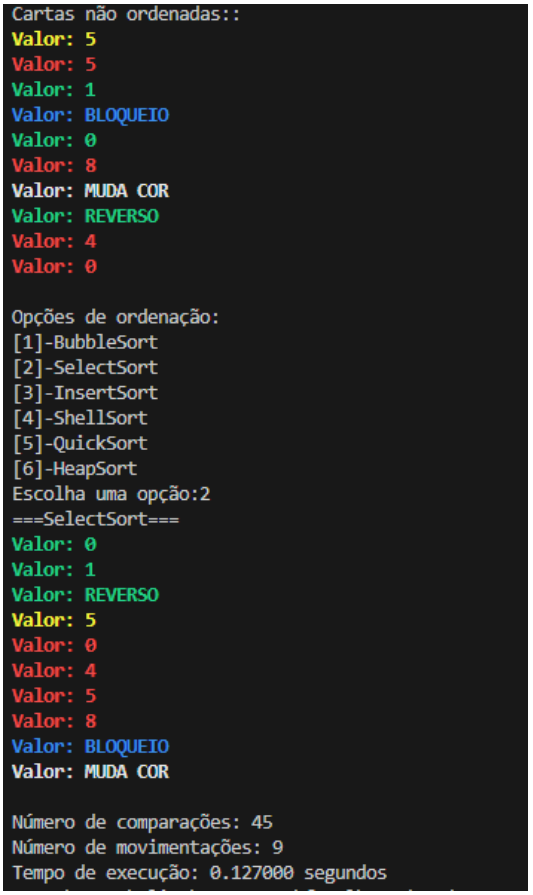
5.1. Print Saídas Modo Interativo

BubbleSort



(Figura 18: Saída BubbleSort)

SelectSort



(Figura 19: Saída SelectSort)

InsertSort

```
Cartas não ordenadas::
Valor: 9
Valor: MUDA COR
Valor: 0
Valor: BLOQUEIO
Valor: BLOQUEIO
Valor: 8
Valor: 2
Valor: 6
Valor: 2
Valor: 8

Opções de ordenação:
[1]-BubbleSort
[2]-SelectSort
[3]-InsertSort
[4]-ShellSort
[5]-QuickSort
[6]-HeapSort
Escolha uma opção:3
===InsertSort===
Valor: 2
Valor: 6
Valor: 9
Valor: BLOQUEIO
Valor: BLOQUEIO
Valor: 0
Valor: 2
Valor: 8
Valor: 8
Valor: MUDA COR

Número de comparações: 30
Número de movimentações: 31
Tempo de execução: 0.069000 segundos
```

(Figura 20: Saída InsertSort)

ShellSort

```
Cartas não ordenadas::
Valor: 3
Valor: +4
Valor: 8
Valor: 5
Valor: 2
Valor: 8
Valor: REVERSO
Valor: 1
Valor: 6
Valor: 5

Opções de ordenação:
[1]-BubbleSort
[2]-SelectSort
[3]-InsertSort
[4]-ShellSort
[5]-QuickSort
[6]-HeapSort
Escolha uma opção:4
===ShellSort===
Valor: 5
Valor: 5
Valor: 3
Valor: 6
Valor: 8
Valor: REVERSO
Valor: 1
Valor: 2
Valor: 8
Valor: +4

Número de comparações: 15
Número de movimentações: 11
Tempo de execução: 0.092000 segundos
```

(Figura 21: Saída ShellSort)

QuickSort

```
Cartas não ordenadas::
Valor: 0
Valor: +2
Valor: 9
Valor: 1
Valor: 0
Valor: 4
Valor: 5
Valor: 6
Valor: 5
Valor: 3

Opções de ordenação:
[1]-BubbleSort
[2]-SelectSort
[3]-InsertSort
[4]-ShellSort
[5]-QuickSort
[6]-HeapSort
Escolha uma opção:5
===QuickSort===
Valor: 1
Valor: 3
Valor: 0
Valor: 5
Valor: 9
Valor: +2
Valor: 0
Valor: 4
Valor: 5
Valor: 6

Número de comparações: 22
Número de movimentações: 7
Tempo de execução: 0.160000 segundos
```

(Figura 22: Saída QuickSort)

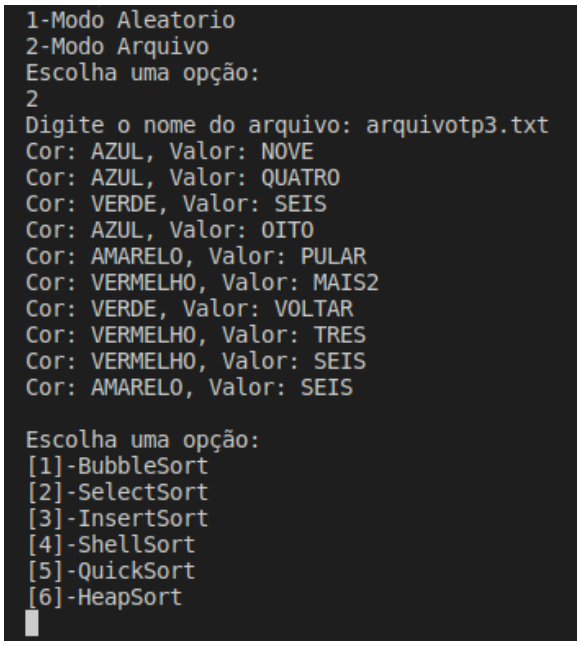
HeapSort

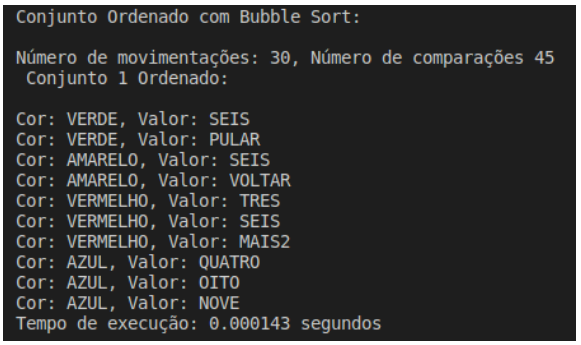
```
Cartas não ordenadas::
Valor: 7
Valor: REVERSO
Valor: 4
Valor: 2
Valor: 0
Valor: 7
Valor: 6
Valor: 4
Valor: 1
Valor: 9

Opções de ordenação:
[1]-BubbleSort
[2]-SelectSort
[3]-InsertSort
[4]-ShellSort
[5]-QuickSort
[6]-HeapSort
Escolha uma opção:6
===HeapSort===
Valor: 6
Valor: 7
Valor: 1
Valor: 4
Valor: 9
Valor: REVERSO
Valor: 0
Valor: 2
Valor: 4
Valor: 7

Número de comparações: 99
Número de movimentações: 28
Tempo de execução: 0.033000 segundos
```

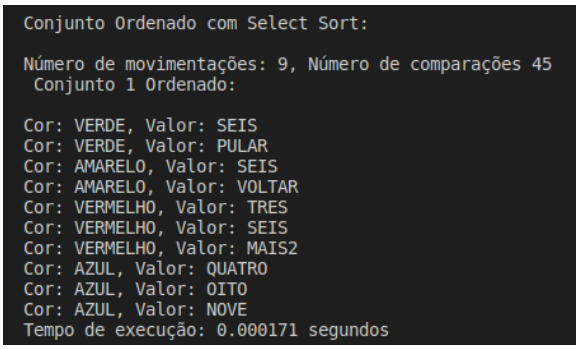
(Figura 23: Saída HeapSort)

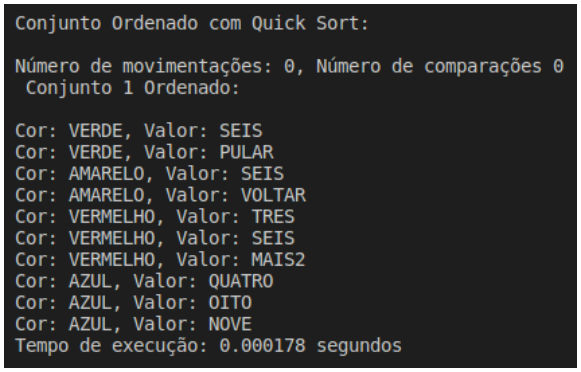
5.2. Print Saídas Modo Arquivo

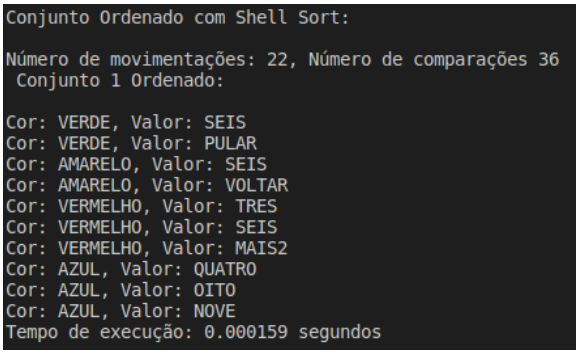
- **Conjunto 1**

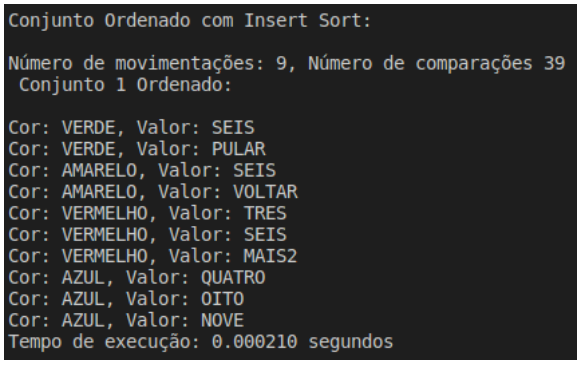


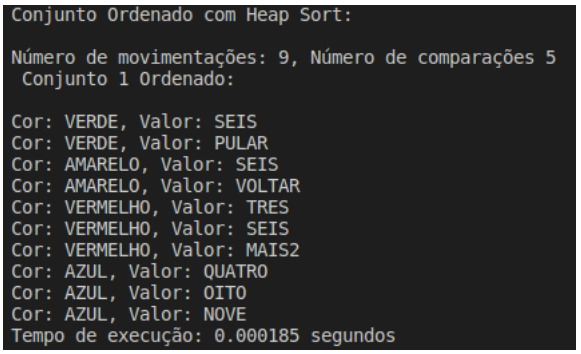






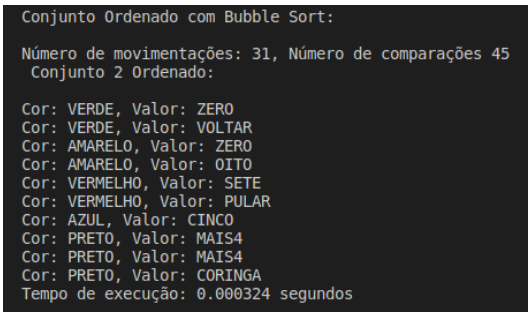


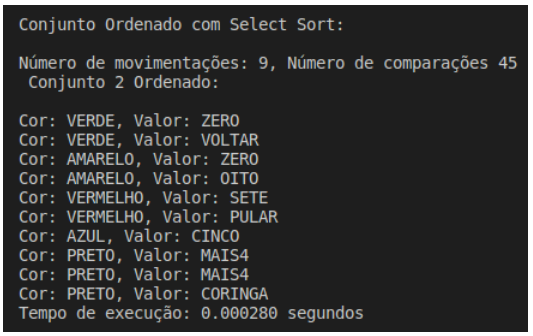


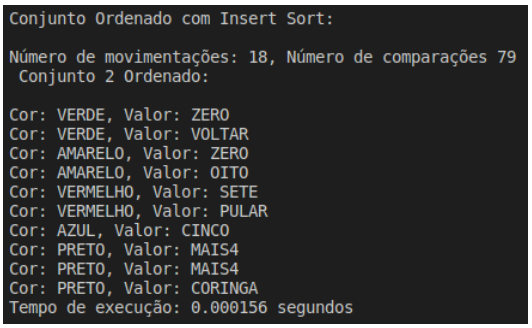


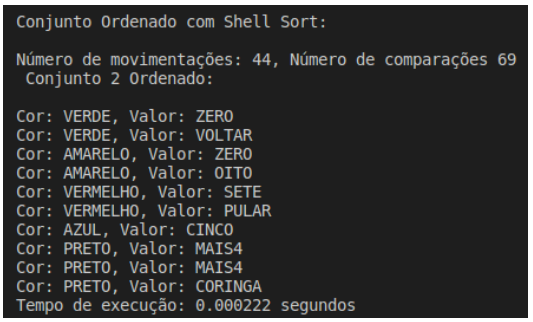
(Figura 24,25,26, 27, 28, 29, 30: Ordenações do conjunto 1)

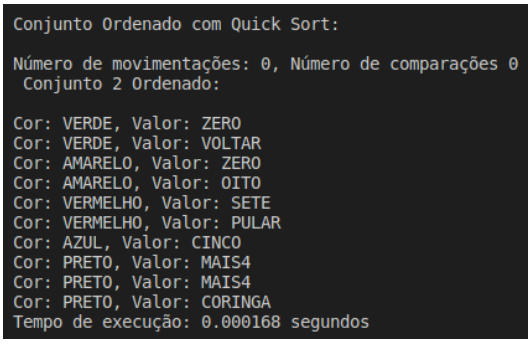
- **Conjunto 2**

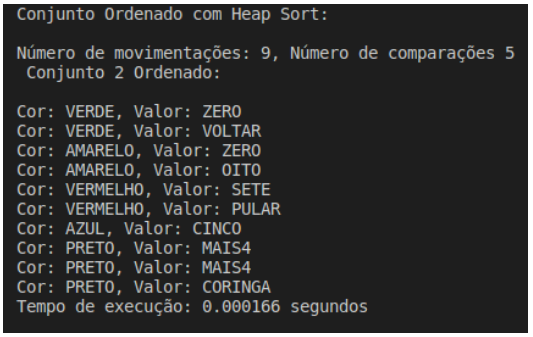






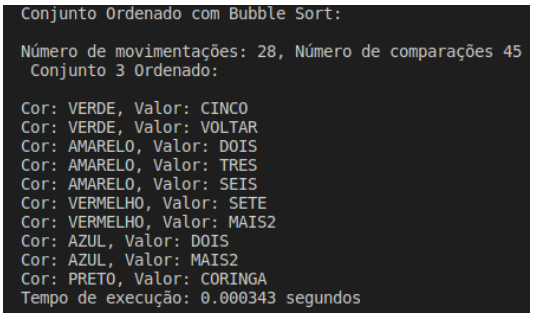


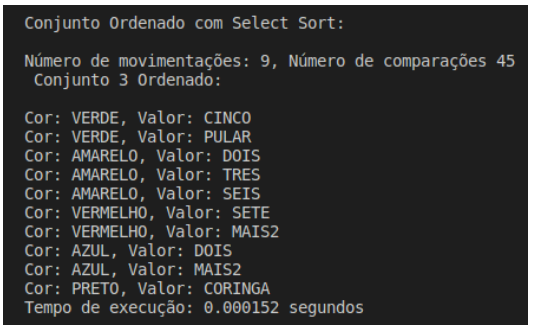


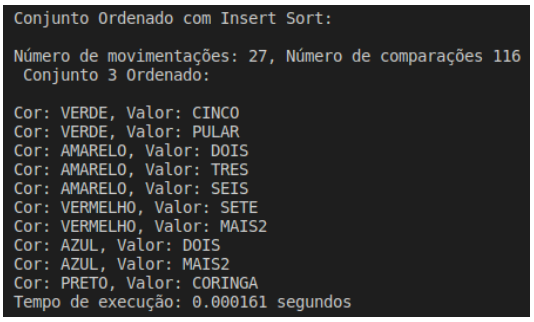


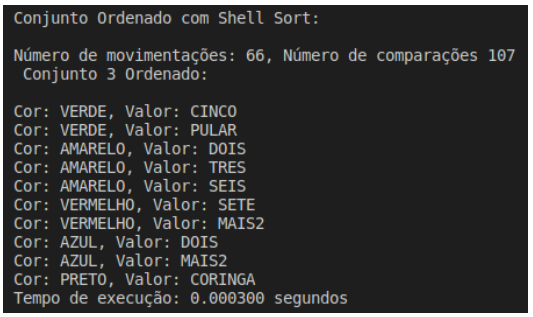
(Figura 30, 31,32,33,34,35,36: Ordenações do conjunto 2)

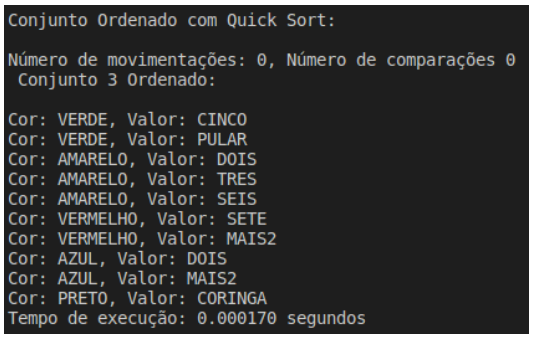
- **Conjunto 3**

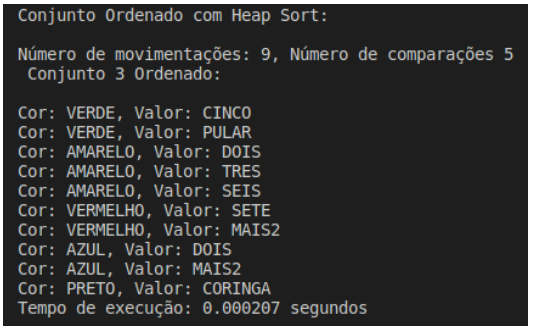












(Figura 37,38,39,40,41,42: Ordenações do conjunto 3)

6. Comparações entre algoritmos

Número de Comparações:

- **Bubble Sort**: Tende a fazer um grande número de comparações, mesmo em conjuntos de dados pequenos. É menos eficiente em comparação com outros algoritmos.
- **Selection Sort**: Realiza um número fixo de comparações para cada elemento, independentemente do estado do conjunto de dados. Assim como o Bubble Sort, não é eficiente para conjuntos grandes.
- **Insertion Sort**: Tende a fazer um número de comparações e movimentações que é proporcional ao número de inversões no conjunto de dados. Desempenho médio em conjuntos pequenos.
- **Shell Sort:** O número de comparações varia dependendo da sequência de lacunas escolhida. Pode ser eficiente em alguns casos, mas ainda pode ser superado por algoritmos mais avançados.
- **Quick Sort:** Eficiente na média dos casos, com um número menor de comparações em comparação com os métodos anteriores. No entanto, o pior caso pode ser desvantajoso.
- **Heap Sort:** Geralmente tem menos comparações em comparação com os métodos anteriores, tornando-o mais eficiente em termos de comparações.

Número de Movimentações:

- **Bubble Sort:** Tende a realizar um grande número de movimentações, especialmente em conjuntos inversamente ordenados.
- **Selection Sort**: Realiza um número fixo de movimentações para cada elemento, independentemente do estado do conjunto de dados.
- **Insertion Sort:** Tende a ter um número de movimentações proporcional ao número de inversões no conjunto de dados.
- **Shell Sort:** O número de movimentações pode variar dependendo da sequência de lacunas escolhida. Em geral, menos movimentações do que o Bubble Sort e Selection Sort.
- **Quick Sort**: Realiza um número significativo de trocas, especialmente no pior caso. No entanto, essas trocas são feitas de maneira mais eficiente do que em métodos simples.
- **Heap Sort**: Realiza um número moderado de movimentações, tornando-o eficiente nesse aspecto.

Tempo de Execução:

- **Bubble Sort, Selection Sort, Insertion Sort**: Geralmente mais lentos, especialmente em conjuntos grandes, devido às suas complexidades quadráticas.

- **Shell Sort**: Melhor desempenho do que os métodos anteriores, mas ainda pode ser superado por métodos mais avançados.
- **Quick Sort e Heap Sort:** Têm um desempenho mais eficiente em conjuntos de dados grandes devido à sua complexidade de tempo médio/quase linear.

Vantagens e Desvantagens:

- **Bubble Sort**, Selection Sort, Insertion Sort: São simples de implementar, mas geralmente ineficientes em conjuntos grandes.
- **Shell Sort**: Melhora a eficiência do Bubble Sort, mas a escolha da sequência de lacunas pode impactar o desempenho.
- **Quick Sort:** Rápido em média, mas pode ser lento no pior caso. Boa escolha para conjuntos grandes e em aplicações gerais.
- **Heap Sort**: Eficiente em termos de comparações e movimentações, mas pode ter um desempenho um pouco mais lento em comparação com Quick Sort em alguns casos.

7. Conclusão

Chegamos a conclusão que os algoritmos de classificação dependem de vários fatores, como o tamanho da entrada, a distribuição dos dados e os requisitos específicos da aplicação. Mas, usando como base o código que criamos no trabalho prático, podemos fazer as seguintes observações:

1. **Bubble Sort:** É um algoritmo simples e fácil de entender, mas não é eficiente para grandes conjuntos de dados. Tem uma complexidade de tempo de O(n^2) nos casos pior e médio e requer um grande número de comparações e trocas. Portanto, não é recomendado para classificação de conjuntos grandes.
2. **Select Sort** : Também é um algoritmo simples, mas tem desempenho melhor do que a classificação por bolha em termos de número de trocas. No entanto, ainda possui uma complexidade de tempo de O(n^2) em todos os casos, tornando-o ineficiente para grandes conjuntos de dados.
3. **Insert Sort**: É eficiente para pequenos conjuntos de dados.Possui uma complexidade de tempo de O(n^2) nos casos pior e médio, mas funciona bem quando a entrada já está parcialmente ordenada. Requer menos comparações e trocas em comparação com Bubble Sort e Selection Sort.
4. **Shell Sort:** É uma melhoria em relação ao Insertion Sort. Ele tem uma complexidade de tempo de O (n ^ 2) no pior caso, mas tem desempenho melhor do que a classificação por inserção para conjuntos de dados maiores. Shell Sort usa uma técnica chamada "sequência de intervalo" para classificar elementos distantes antes de classificar todo o array. Ele tem melhor desempenho do que classificação por bolha, classificação por seleção e classificação por inserção para conjuntos de dados maiores.

5. **Quick Sort**: É um algoritmo de classificação amplamente utilizado, conhecido por sua eficiência. Tem uma complexidade de tempo média de O(n log n) e uma complexidade de tempo de pior caso de O(n^2). No entanto, na prática, o Quick Sort tem um bom desempenho e geralmente é mais rápido que outros algoritmos de classificação. Ele usa uma abordagem de dividir e conquistar e tem bom desempenho de cache.
6. **Heap Sort**: É um algoritmo de classificação eficiente com uma complexidade de tempo de O (n log n) em todos os casos. Ele usa uma estrutura de dados heap binária para classificar os elementos e tem bom desempenho de cache. Embora tenha um fator constante um pouco mais alto em comparação com o Quick Sort, ainda é uma escolha popular para classificar grandes conjuntos de dados.

Com base nas nossas observações, pensamos que a melhor conclusão seria que a escolha do algoritmo de ordenação depende dos requisitos específicos da aplicação e das características dos dados de entrada. Para conjuntos de dados pequenos ou conjuntos parcialmente ordenados, insertSort ou shellSort podem ser adequados. Para conjuntos de dados maiores, Quick Sort ou Heap Sort seriam mais eficientes. E como nosso trabalho se trata de ordenar pequenos conjuntos , os algoritmos de ordenação simples (bubble,insert e select) atenderam facilmente a demanda do nosso código.