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

Trabalho 3 - AEDS 1

Algoritmos de ordenação para cartas de uno.

Marcus Eduardo [EF05779]
Yuri Roberto [EF05380]
Matheus Junio [EF05382]

Florestal - MG
2023

Sumário

1. Introdução

A proposta deste trabalho foi organizar um baralho de uno por meio dos seis algoritmos de ordenação apresentados ao longo do curso, além disso, o código deve ser capaz de ler e ordenar um arquivo contendo cartas do jogo.

O código ainda conta com a interação do usuário para selecionar os modos de ordenação que o mesmo deseja, e exibe a ordenação realizada por cada algoritmo separadamente, além do numero de comparações e movimentações de cada algoritmo, e exibindo ao fim seu tempo de execução.

2. Organização

.

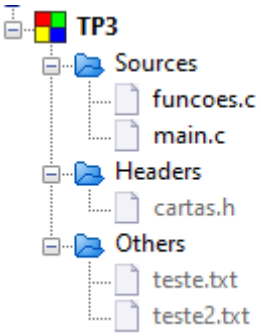

Figura 1 - Repositório do projeto.

O código foi separado em três partes, no TAD cartas, o mesmo inclui as funções de ler uma carta e de criar uma carta aleatória, além de conter a estrtura de uma carta.

No arquivo das funcoes.c o código armazena todas as funções necessárias para o funcionamento do código no main.

O main é responsável por atender as escolhas do usuário e de exibir a interface do código quando executado, e também pela interação.

Por fim os arquivos de teste servem para testar o modo arquivo contido no código, por meio de diferentes entradas para o primeiro e segundo teste.

3. Desenvolvimento

Como dito acima, o código foi divido em partes, e cada parte com sua importância no funcionamento do mesmo, isto será explicado de forma simplificada e breve a fim de possuir um bom entendimento de forma geral sobre todo o código.

3.1 Tad cartas.h

```
#ifndef CARTAS_H_INCLUDED
#define CARTAS_H_INCLUDED


typedef struct {
    int cor;
    int numero;
} Carta;

const char* nomeCor(int cor);
Carta gerarCartaAleatoria();
int lerCartaEspecial(const char* specialCard);


#endif // CARTAS_H_INCLUDED
```

Figura 2-Cartas.h.

Nesta parte o código cria uma estrutura do tipo carta para armazenar os valores e as funções relacionadas a ela, além de criar uma carta aleatória e uma função para ler carta do tipo especial, vale ressaltar que esta parte do código é extremamente importante em seu funcionamento pois sem o mesma o código não funciona.

3.2 Funcoes.c

Nesta parte o código armazena todas as funções contidas e necessárias para o funcionamento do main e de header, logo uma representação do mesmo de forma geral se torna inviável, devido ao seu enorme tamanho, portanto o mesmo terá cada uma de suas funções explicadas de forma separada.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <windows.h>
#include "cartas.h"

#define VERDE 0
#define AMARELO 1
#define VERMELHO 2
#define AZUL 3
#define PRETO 4

#define PULAR 10
#define VOLTAR 11
#define MAIS_DOIS 12
#define CORINGA 13
#define MAIS_QUATRO 14

long long comparacoes = 0;
long long movimentacoes = 0;
```

Figura 3-Includes necessários.

Os includes nesta parte servem para a definição de cada tipo e cor de carta, o que posteriormente será utilizado na comparação de cada algoritmo, que é feita por inteiros, isto explica o uso dos defines, além disso, algumas bibliotecas são incluídas para que o código funcione corretamente.

```c
const char* nomeCor(int cor) {
    switch (cor) {
        case VERDE:
            return "Verde";
        case AMARELO:
            return "Amarelo";
        case VERMELHO:
            return "Vermelho";
        case AZUL:
            return "Azul";
        case PRETO:
            return "Preto";
        default:
            return "Cor Inválida";
    }
}
```

Figura 4-nomeCor.

A função nomeCor é responsável por ler a cor e retornar um valor em inteiro que foi definido nos defines acima, possibilitando assim uma comparação.

```c
void PrintarCartas(Carta cards[], int n) {
    for (int i = 0; i < n; i++) {
        printf("[%d]%s ", i + 1, nomeCor(cards[i].cor));

        if (cards[i].numero >= 0 && cards[i].numero <= 9) {
            printf("%d", cards[i].numero);
        } else {
            switch (cards[i].numero) {
                case 10:
                    printf("Voltar");
                    break;
                case 11:
                    printf("Pular");
                    break;
                case 12:
                    printf("+2");
                    break;
                case 13:
                    printf("Coringa");
                    break;
                case 14:
                    printf("+4");
                    break;
                default:
                    printf("Erro: Numero invalido");
```

Figura 5-PrintarCarta.

Esta função é responsável por exibir os conjuntos de cartas na tela em forma de coluna, além disso, ela converte os inteiros para seus respectivos nomes antes de realizar o print.

```c
void embaralhar(Carta cards[], int n) {
    srand(time(NULL));
    for (int i = n - 1; i > 0; i--) {
        int j = rand() % (i + 1);
        // Troca as cartas[i] e cartas[j]
        Carta temp = cards[i];
        cards[i] = cards[j];
        cards[j] = temp;
        movimentacoes += 3;  // 3 movimentações foram realizadas
    }
}
```

Figura 6-Embaralhar.

A função embaralhar é responsável por embaralhar os conjuntos de cartas criadas no modo interativo do código.

```c
Carta gerarCartaAleatoria() {
    Carta carta;
    carta.cor = rand() % 4;     // Ge
    carta.numero = rand() % 12 + 1;
    return carta;
}
```

Figura 7-GerarCartaAleatoria.

Esta função é capaz de criar uma carta de qualquer cor e tipo, servindo assim para criar sempre conjuntos diferentes de cartas no modo aleatório do código.

```
void quickSort(Carta cards[], int low, int high) {
    if (low < high) {
        int pi = particao(cards, low, high);

        quickSort(cards, low, pi - 1);
        quickSort(cards, pi + 1, high);
    }
}
```

Figura 8-QuickSort.

Esta função funciona para aplicar as regras de ordenação do Quick Sort, porém esta parte da função mostra uma recursividade, enquanto a função partição é quem ordena os elementos de acordo com a implementação do algoritmo, segue abaixo a representação meramente ilustrativa da função partição:

```
int particao(Carta cards[], int low, int high) {
    Carta pivot = cards[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        comparacoes++;  // Para cada comparação
        // Compare primeiro as cores, em caso de igualdade compare os números
        if (cards[j].cor < pivot.cor || (cards[j].cor == pivot.cor && cards[j].numero < pivot.numero)) {
            i++;

            // Troca as cartas[i] e cartas[j]
            Carta temp = cards[i];
            cards[i] = cards[j];
            cards[j] = temp;
            movimentacoes += 3;  // 3 movimentações foram realizadas
        }
    }

    // Troca as cartas[i + 1] e cartas[high]
    Carta temp = cards[i + 1];
    cards[i + 1] = cards[high];
    cards[high] = temp;
    movimentacoes += 3;  // 3 movimentações foram realizadas

    return i + 1;
}
```

Figura 9-Partição.

```
void bubbleSort(Carta cards[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            comparacoes++;  // Para cada comparação

            // Compare primeiro as cores, em caso de igualdade compare os números
            if (cards[j].cor > cards[j + 1].cor ||
                (cards[j].cor == cards[j + 1].cor && cards[j].numero > cards[j + 1].numero)) {
                // Troca as cartas[j] e cartas[j + 1]
                Carta temp = cards[j];
                cards[j] = cards[j + 1];
                cards[j + 1] = temp;
                movimentacoes += 3;  // 3 movimentações foram realizadas
            }
        }
    }
}
```

Figura 10-BubbleSort.

A função bubbleSort, contem a forma de funcionamento do algoritmo em questão, a mesma compara primeiro as cores e depois seus números e/ou tipos da carta, fazendo assim sua ordenação.

```
void heapSort(Carta cards[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(cards, n, i);
    }

    for (int i = n - 1; i > 0; i--) {
        // Troca as cartas[0] e cartas[i]
        Carta temp = cards[0];
        cards[0] = cards[i];
        cards[i] = temp;
        movimentacoes += 3;  // 3 movimentações foram realizadas

        heapify(cards, i, 0);
    }
}
```

Figura 11-HeapSort.

Tal qual no Quick Sort, a função do heapsort utiliza uma função auxiliar para verificar suas condições e implementar seus métodos, sendo a função do heapsort responsável por trocar as cartas de lugar e de chamar a função heapify para uma nova comparação caso seja necessário, segue abaixo a imagem da função heapify:

```
void heapify(Carta cards[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n) {
        comparacoes++;  // Para cada comparação
        // Compare primeiro as cores, em caso de igualdade compare os números
        if (cards[left].cor > cards[largest].cor ||
            (cards[left].cor == cards[largest].cor && cards[left].numero > cards[largest].numero)) {
            largest = left;
        }
    }

    if (right < n) {
        comparacoes++;  // Para cada comparação
        // Compare primeiro as cores, em caso de igualdade compare os números
        if (cards[right].cor > cards[largest].cor ||
            (cards[right].cor == cards[largest].cor && cards[right].numero > cards[largest].numero)) {
            largest = right;
        }
    }

    if (largest != i) {
        // Troca as cartas[i] e cartas[largest]
        Carta temp = cards[i];
        cards[i] = cards[largest];
```

Figura 12-Heapify.

```
void selectionSort(Carta cards[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            comparacoes++;  // Para cada comparação
            // Compare primeiro as cores, em caso de igualdade compare os números
            if (cards[j].cor < cards[minIndex].cor ||
                (cards[j].cor == cards[minIndex].cor && cards[j].numero < cards[minIndex].numero)) {
                minIndex = j;
            }
        }
        // Troca as cartas[i] e cartas[minIndex]
        Carta temp = cards[i];
        cards[i] = cards[minIndex];
        cards[minIndex] = temp;
        movimentacoes += 3;  // 3 movimentações foram realizadas
    }
}
```

Figura 13-selectionSort.

A função selectionSort, implementa todo o funcionamento do algoritmo de seleção e independe de outra função, já realizando as trocas e contando as movimentações e comparações na própria função.

```
void insertionSort(Carta cards[], int n) {
    for (int i = 1; i < n; i++) {
        Carta key = cards[i];
        int j = i - 1;

        while (j >= 0 && (cards[j].cor > key.cor ||
                          (cards[j].cor == key.cor && cards[j].numero > key.numero))) {
            comparacoes++;  // Para cada comparação
            // Move as cartas[j] para a posição correta
            cards[j + 1] = cards[j];
            movimentacoes++;
            j--;
        }

        cards[j + 1] = key;
        movimentacoes++;  // 1 movimentação foi realizada
    }
}
```

Figura 14-insertionSort.

A função insertionSort, independe de uma função auxiliar e implementa a ordenação por inserção além de já contabilizar todas as movimentações efetuadas durante o código.

```
void shellSort(Carta cards[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            Carta temp = cards[i];
            int j;

            for (j = i; j >= gap && (temp.cor < cards[j - gap].cor ||
                    (temp.cor == cards[j - gap].cor && temp.numero < cards[j - gap].numero)); j -= gap) {
                comparacoes++;
                cards[j] = cards[j - gap];
                movimentacoes++;
            }

            cards[j] = temp;
            movimentacoes++;
        }
    }
```

Figura 15-ShellSort.

A função shellSort, independe de uma função auxiliar, e assim como as demais funções realiza suas comparações por meio de inteiros, entregando ao fim um conjunto de inteiros ordenado, sendo convertido no print.

```
int lerCartaEspecial(const char* specialCard) {
    if (strcmp(specialCard, "Voltar") == 0) {
        return VOLTAR;
    } else if (strcmp(specialCard, "Pular") == 0) {
        return PULAR;
    } else if (strcmp(specialCard, "+2") == 0) {
        return MAIS_DOIS;
    } else {
        // Retorne -1 ou outro valor para indicar um erro
        return -1;
    }
}
```

Figura 16-LerCartaEspecial.

Esta função serve apenas para ler os valores não numéricos e fazer a conversão para as variáveis que possuem seus valores inteiros correspondentes.

Vale ressaltar que o código possui uma função para ler um arquivo, cuja a entrada são cartas do tipo Cartas, e realizar sua ordenação posteriormente pelos algoritmos, o código também tem uma função para o modo interativo que possibilita que o usuário escolha qual algoritmo irá usar em sua ordenação, e por fim uma função que aplica este algoritmo a escolha do usuário. As representações destas funções estão disponíveis no link do github nas referencias.

3.3 main.c.

O main ficou responsável por gerenciar a interface e as escolhas do usuário, realizando as chamadas corretas das funções contidas no arquivo das funções, sendo assim o main ficou organizado e limpo, para promover melhor entendimento e leitura do código, segue abaixo um trecho do main.

```
Carta cards[100]; // Ajuste o tamanho conforme necessário
int n=10, opcaoAlgoritmo; // n = 10 é o tamanho do conjunto

printf("Escolha o modo:\n");
printf("1 - (Ponto extra) Modo Interativo: escolher algoritmo individualmente\n");
printf("2 - (Ponto extra) Modo Arquivo: Todos conjuntos ordenados juntos\n");
printf("3 - Modo Arquivo: Cada conjunto ordenado separadamente aplicando todas ordenacoes\n");
printf("4 - Modo Interativo: Aplicando todas ordenacoes no conjunto criado\n");

int modo;
scanf("%d", &modo);
```

Figura 17 – Ilustração do main.

4. Compilação e Execução

Para uma execução correta e sem erros, o usuário deverá verificar se seu repositório está completo, contendo tudo que foi apresentado na figura 1, deste documento, caso contrario o código estará sujeito a erros.

5. Resultados

Ao executar o código o mesmo irá perguntar ao usuário quais são os modos de jogo:

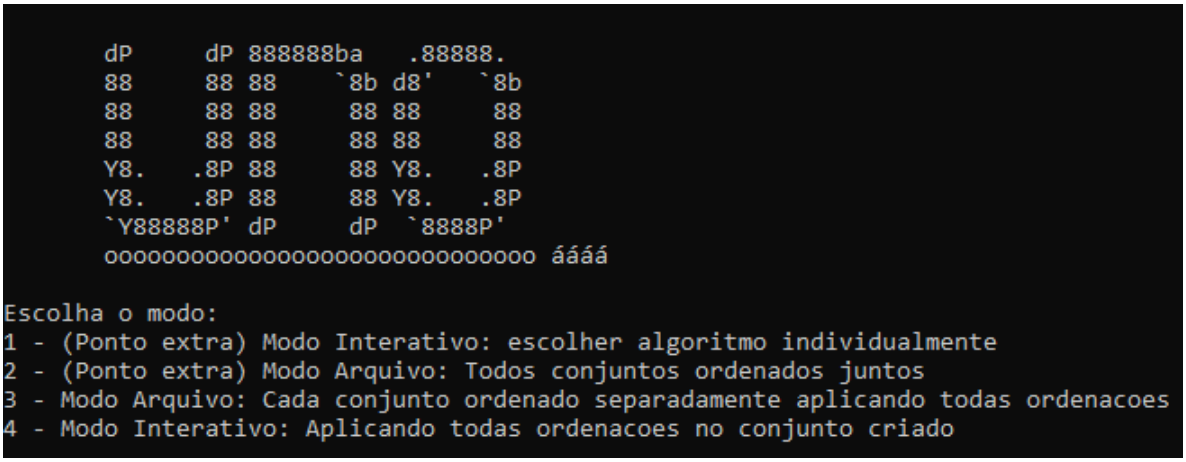

Figura 18- Interface.

O código gera quatro opções ao usuário, sendo duas delas extras, isso foi feito para que fosse realizado o teste de cada algoritmo separadamente, e achamos interessante manter no código final tais opções.

→Escolha 1:

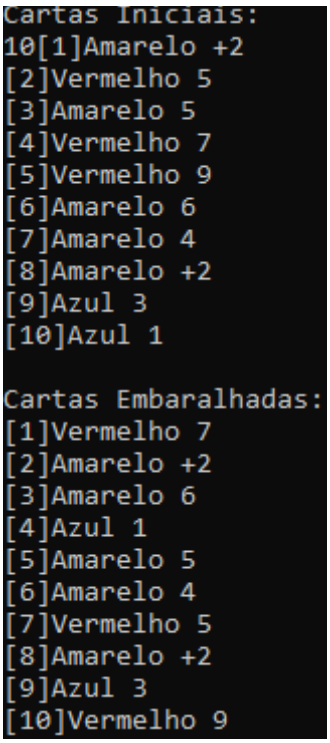

Figura 19 – embaralhar.

Ao escolher a opção um, o código cria um conjunto de 10 cartas, e as embaralha, pedindo ao jogador posteriormente que escolha um algoritmo de ordenação para ordenar o conjunto.

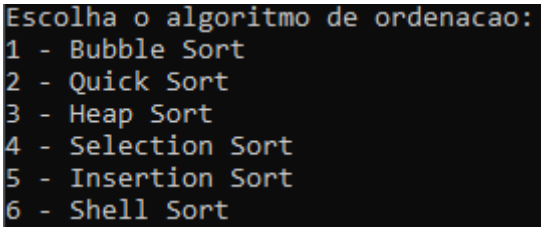

Figura 20 – Seleção.

Para critérios ilustrativos escolhemos a opção seis, e seu resultado será exibido na próxima imagem, o mesmo equivale para os demais algoritmos de ordenação.

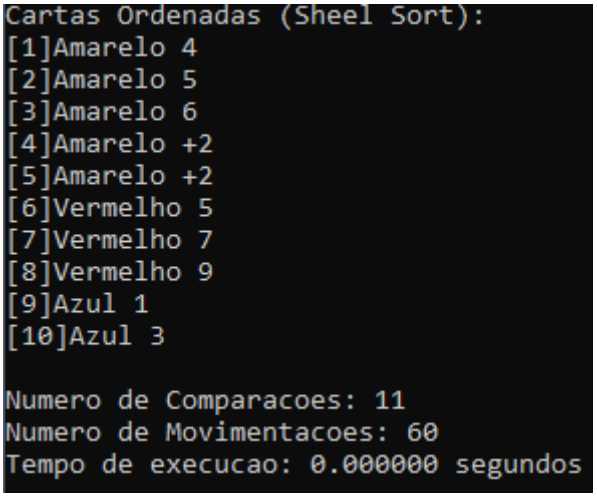

Figura 21 – Ordenados.

→Escolha 2:

Caso a opção escolhida seja a de numero 2, o código irá ler um arquivo e embaralhar as cartas, e posteriormente solicitar ao usuário para que escolha um algoritmo para sua ordenação, casos análogos aos representados nas Figuras 20 e 21.

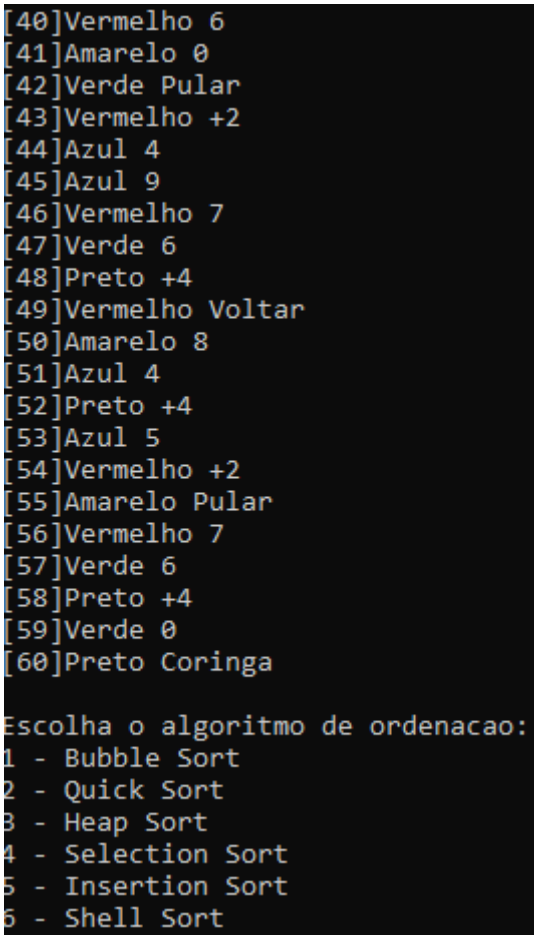

Figura 22 – Opção 2.

OBS: O conjunto contido no arquivo possui 60 cartas, por isso a figura 22 foi cortada.

→Escolha 3:

Ao escolher a opção 3, o código irá ler um arquivo de teste, e separar pelo numero de conjuntos que o arquivo deseja, e a partir dai realizar a ordenação de cada conjunto separadamente e por cada um dos seis algoritmos, exibindo tudo na tela ao final do processo.

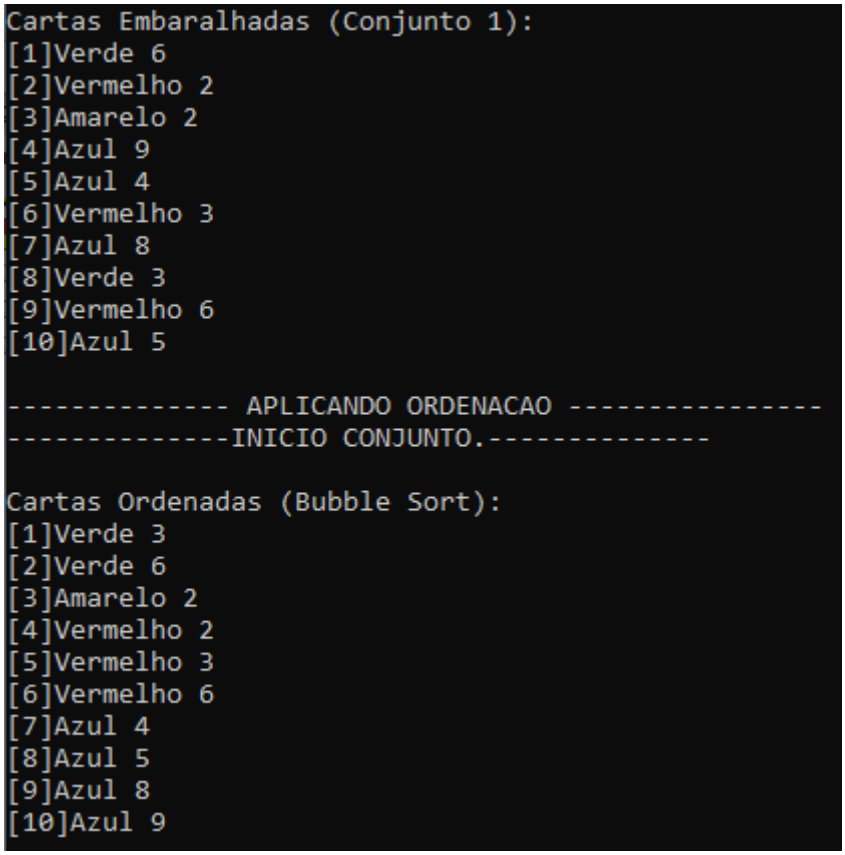

Figura 23 – Conjuntos.

Vale ressaltar que o código mostra o conjunto e logo após mostra sua ordenação pelos diferentes algoritmos, sendo um caso análogo ao da figura 23.

→Escolha 4:

Para a escolha 4, o código irá gerar um conjunto de 10 cartas e embaralhar o mesmo, depois o código irá ordenar este conjunto por todos os seis algoritmos de ordenação presentes no código, e exibir seus resultados na tela, tal qual a figura 20, onde para todos os seis são casos análogos, logicamente dependendo do algoritmo e do conjunto.

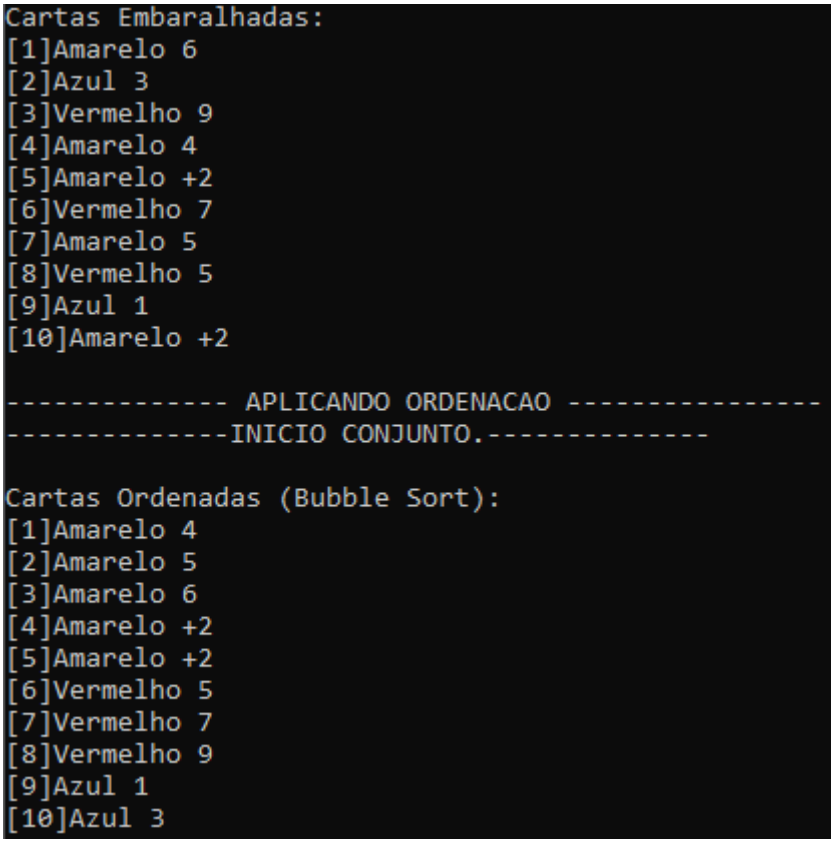

Figura 24 – Opção 4.

OBS: Na imagem foi apresentado apenas o Bubble Sort, para fins ilustrativos.

6. Conclusão

O trabalho trouxe consigo desafios na implementação e na decisão de como os valores das cartas seriam representadas, principalmente nas cartas especiais e pretas, já que as mesmas não possuem valores numéricos, além disso, enfrentamos grande dificuldade na leitura do arquivo, e de novo nas cartas especiais, mas com empenho e esforço tal problema foi solucionado.

Com a obrigação de mostrar o numero de movimentações e comparações, foi possível visualizar a forma de cada algoritmo trabalhar e sua forma de lidar com o conjunto de cartas, além de promover certa comparação entre os seis algoritmos presentes no código, podendo o usuário visualizar qual foi o melhor para ele.

Por fim, foi possível entender a forma de funcionamento e como cada um dos seis algoritmos funciona e realiza sua ordenação, isso foi fundamental para o conhecimento e entendimento dos mesmos.

7. Referências

[1] Github: https://github.com/matheus-junio-da-silva/tp3_AEDS_ordenacao

Acesse este link para visualização do código e de suas funcionalidades, já que as imagens presentes no arquivo são ilustrativas.