Documentação - Trabalho Prático 2
Cartas de Baralho

Ryan Junio de Oliveira (5258)

[1]Universidade Federal de Viçosa (UFV)
Campus Florestal

ryan.oliveira@ufv.br

Resumo. *O propósito deste trabalho é investigar e reforçar o conteúdo abordado nos módulos dois e três da disciplina, que abordam tópicos sobre Pilhas/Filas, Algoritmos de Ordenação e Análise de Complexidade, através de um jogo de cartas. Em que foi explorado a lógica para construção do jogo com qualquer tipo de entrada.*

1. Introdução

O programa é um jogo de baralho que permite a manipulação e jogabilidade de cartas. Ele oferece funcionalidades de ordenação das cartas além de possibilitar jogos de cartas entre os jogadores. O sistema também inclui recursos de embaralhamento proporcionando uma experiência completa de baralho.

No jogo, o objetivo é atingir uma sequência de cartas de mesmo valor e, o jogador que conseguir tal feito, ganha. Dentre suas maiores dificuldades está na verificação da mão do jogador, geração de cartas e a dinâmica de implementação. Na ordenação, a maior dificuldade está em comparar naipe e valor, visto que os naipes tem uma ordem de prioridade e a quantidade grande de cartas.

O funcionamento do código está no tipo abstrato de dados, em que é dividido pelo arquivo tad_baralho.h, tad_baralho.c e main.c. Isso é, definição de struct e funções, desenvolvimento das funções e suas lógicas e o desenvolvimento do jogo, respectivamente.

2. Modelagem e Funcionamento

O jogo baseia-se em poder comprar uma carta do baralho ou capturar uma carta do topo, que havia sido descartada. Para isso, há uma variável chamada ”primeiraJogada”que carrega o valor True, após a tentativa do jogador um, essa variável recebe False.

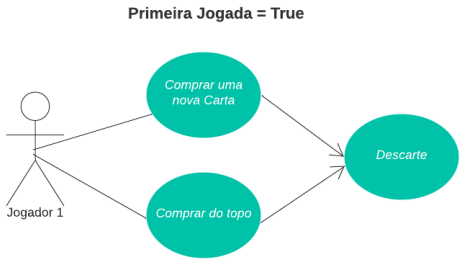

Figura 1. Diagrama da Primeira Jogada

Essa estratégia é usada para que haja um determinado começo, tendo em vista que no começo do jogo não há nenhuma carta no descarte, dessa forma, o primeiro jogador tem apenas a opção de comprar a carta e descartar para o segundo jogador. Após a primeira jogada, acontece um ciclo contínuo até que uma mão vença, sendo esse o critério para encerrar o looping.

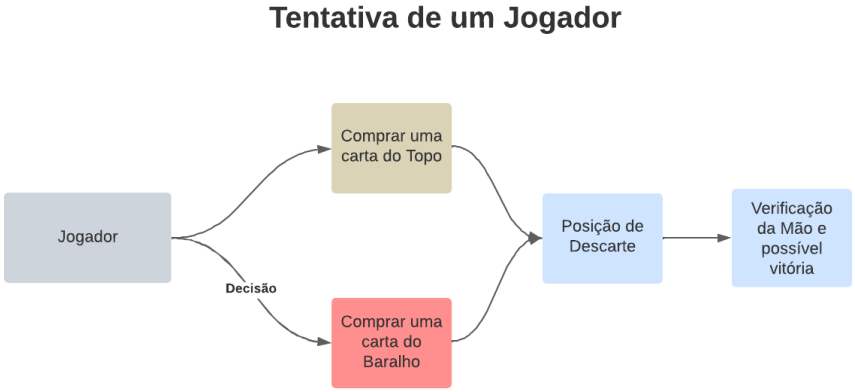

Figura 2. Mapa Conceitual de uma tentativa

3. Implementação

3.1. Leitura do Arquivo

De início temos a abertura do arquivo de forma tradicional, declarando o FILE e abrindo com a condição de que encerrar o procedimento se o arquivo não for aberto corramente.

```
15    //ARQUIVO DE ENTRADA
16    FILE *arquivo_entrada = fopen("entrada_buraco.txt","r");
17    if (arquivo_entrada == NULL) {
18      printf("Não foi possível abrir o arquivo.\n");
19      return 1;
20    }
21
22    //ARQUIVO DE SAIDA
23    FILE* arquivo_saida;
24    sprintf(nome_arquivo, "ordenacao_entrada_%s.txt", metodo);
25
26    arquivo_saida = fopen(nome_arquivo,"w");
27    if(arquivo_saida == NULL){
28       printf("Erro ao criar o arquivo.\n");
29       return 1;
30    }
```

Figura 3. Declaração e Leitura do arquivo

Após isso, começamos a ler cada linha da entrada.txt e fazendo suas determinadas operações.

```
33    // LEITURA DO ARQUIVO
34    char linha[100];
35
36    //Numero de Baralhos
37    fgets(linha, sizeof(linha), arquivo_entrada);
38    sscanf(linha, "%d", &nBaralho);
39    nCartas = quantidadeCartas(nBaralho);
40
```

Figura 4. Número de Baralhos

Para receber os valores excluídos utilizei da estratégia do strktok, que separa os caracteres com base na refência que colocamos. Neste caso, foi utilizado a virgula. É importante ressaltar que, através dessa lógica de armazenar no vetor, podemos receber qualquer quantidade de cartas excluídas.

```
41    //Cartas Excluídas
42    fgets(linha, sizeof(linha), arquivo_entrada);
43    vExcluidos nExcluidos;
44    nExcluidos.qValores = 0;
45    char *token = strtok(linha, ",");
46    while(token != NULL){
47        nExcluidos.valores[nExcluidos.qValores] = atoi(token);
48        nExcluidos.qValores++;
49        token = strtok(NULL,",");
50    }
```

Figura 5. Cartas Excluídas

A carta coringa é colocado em um array, em que seus caracteres são acessados de forma individual. Em que, há uma comparação pra ser as posições 0 e 1 são diferentes de 0, dessa forma, há a compravação de que existe ou não uma carta coringa. Se caso existe, a variável ”existeCoringa”recebe True. Ademais, existe algumas comparações para modificar os valores para os inteiros, facilitando a manipulação da carta coringa no código.

```
53    fgets(linha, sizeof(linha), arquivo_entrada);
54    sscanf(linha, "%s", &cartaCoringaLinha);
55    Carta cartaCoringa;
56    int existeCoringa = False;
57    if(cartaCoringaLinha[0] != 0 && cartaCoringaLinha[1] != 0){
58        if(cartaCoringaLinha[0] != 'A' && cartaCoringaLinha[0] != 'J' &&
          cartaCoringaLinha[0] != 'Q' && cartaCoringaLinha[0] != 'K'){
59            int aux = atoi(&cartaCoringaLinha[0]);
60            cartaCoringa.valor = aux;
61        }
62        else{
63            if(cartaCoringaLinha[0] == 'A'){
64                cartaCoringa.valor = 1;
65            }
66            else if(cartaCoringaLinha[0] == 'J'){
67                cartaCoringa.valor = 11;
68            }
69            else if(cartaCoringaLinha[0] == 'Q'){
70                cartaCoringa.valor = 12;
71            }
72            else if(cartaCoringaLinha[0] == 'K'){
73                cartaCoringa.valor = 13;
74            }
75        }
76        cartaCoringa.naipe = cartaCoringaLinha[1];
77        existeCoringa = True;
78    }
```

Figura 6. Carta Coringa

A quantidade de cartas na mão é bem simples, só recebemos ela com a função fgets e colocamos dentro da variável.

```
80    //Quantidade de Cartas na Mão
81    fgets(linha, sizeof(linha),arquivo_entrada);
82    sscanf(linha, "%d", &mao);
```

Figura 7. Mão

3.2. Struct

Temos de ínicio temos dois structs que definem o nosso baralho. São muito usados dentre todo o código, sendo de extrema importância.

```c
typedef struct{
    char naipe;
    int valor;
}Carta;

typedef int Apontador;

typedef struct{
    Carta carta[MAX_ITEM_VETOR];
    Apontador Topo;
    int quantidade_cartas;
}Baralho;
```

Figura 8. Struct Pilha do Baralho

Logo após, identifiquei a necessidade de implementar um struct para as cartas excluídas, caso haja. Isso facilitaria a manipulação delas dentro do código.

```c
typedef struct{
    int valores[MAX_ITEM];
    int qValores;
}vExcluidos;
```

Figura 9. Struct Valores Excluídos

4. Funções

Há 12 funções para o funcionamento do jogo, essa quantidade exemplifica a complexidade do código e uma maneira de otimizar seu desempenho. A construção do jogo é baseada em pilhas, o que significa que muitas dessas funções estão relacionadas à criação e manipulação das pilhas.

4.1. Função: criarBaralho

A função recebe como parâmetros um ponteiro para o baralho a ser criado (pBaralho), o número de baralhos a serem criados (nBaralho), uma estrutura que contém informações sobre as cartas a serem excluídas (nExcluidos), e um vetor de cartas (vBaralho) onde as cartas serão armazenadas.

```c
void criarBaralho(Baralho* pBaralho, int nBaralho, vExcluidos nExcluidos, Carta vBaralho[MAX_ITEM]){
    srand(time(NULL));
    Baralho* pBaralhoAux = malloc(sizeof(Baralho));
    Carta* cartaAux = malloc(sizeof(Carta));
    int vetorValor[13] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
    char vetorNaipe[4] = {'C', 'O', 'P', 'E'};
    int qCartas = quantidadeCartas(nBaralho);
```

Figura 10. Parâmetros e Definição de Variáveis

A função utiliza a função ”srand”para inicializar a semente do gerador de números aleatórios com base no tempo atual. Em seguida, são alocados dois ponteiros: pBaralhoAux, que será usado para manipular temporariamente o baralho durante a criação, e

cartaAux, que será usado para manipular temporariamente cada carta durante o processo. É importante ressaltar que, o baralho é criado em uma pilha e em vetor, com intuito de facilitar a ordenação.

Em seguida, são definidos dois vetores: vetorValor, contendo os valores das cartas (1 a 13), e vetorNaipe, contendo os naipes das cartas (C, O, P, E).

Um loop é executado para criar cada carta do baralho. Dentro do loop, as variáveis iValor e iNaipe são calculadas com base no índice atual, usando o operador de módulo para garantir que elas estejam dentro do intervalo correto. A carta é alocada dinamicamente e preenchida com o naipe e valor correspondentes. Em seguida, a carta é adicionada ao baralho auxiliar (pBaralhoAux) usando a função ”Empilha”.

```
54    for(int i=0; i<qCartas; i++){
55        int iValor = i % 13;
56        int iNaipe = i % 4;
57        Carta* carta = malloc(sizeof(Carta));
58        carta->naipe = vetorNaipe[iNaipe];
59        carta->valor = vetorValor[iValor];
60        vBaralho[i].naipe = vetorNaipe[iNaipe];
61        vBaralho[i].valor = vetorValor[iValor];
62        Empilha(pBaralhoAux, carta, nBaralho);
63    }
```

Figura 11. Cartas Aleatórias

Após a criação de todas as cartas, outro loop é executado para verificar se alguma das cartas criadas deve ser excluída, com base nas informações contidas na estrutura nExcluidos. O loop percorre cada carta do baralho auxiliar (pBaralhoAux) e verifica se o valor da carta corresponde a algum valor excluído na estrutura nExcluidos. Se a carta não for excluída, ela é copiada para a variável cartaAux e adicionada ao baralho final (pBaralho) usando a função ”Empilha”.

```
66    int cont=0;
67    for(int i=0; i<qCartas; i++){
68        cont = 0;
69        for(int j=0; j<nExcluidos.qValores; j++){
70            if(pBaralhoAux->carta[i].valor == nExcluidos.valores[j]){
71                cont++;
72            }
73        }
74        if(cont == 0){
75            cartaAux->naipe = pBaralhoAux->carta[i].naipe;
76            cartaAux->valor = pBaralhoAux->carta[i].valor;
77            Empilha(pBaralho, cartaAux, nBaralho);
78        }
79    }
```

Figura 12. Cartas Excluídas

O algoritmo de embaralhamento percorre o baralho de cartas e troca aleatoriamente as posições das cartas. Isso é feito através de um loop que percorre o baralho

do fim ao início. Para cada posição, é gerado um número aleatório j dentro do intervalo válido, e as cartas nas posições i e j são trocadas de lugar tanto no baralho principal (pBaralho-¿carta) quanto no vetor de cartas auxiliar (vBaralho). Isso garante que as cartas fiquem embaralhadas de forma aleatória

```
81      //Algoritmo de Embaralhamento
82      for(int i=(qCartas-1)-(nExcluidos.qValores*4); i>0; i--){
83          int j = rand() % (i + 1);
84          Carta temp = pBaralho->carta[i];
85          pBaralho->carta[i] = pBaralho->carta[j];
86          pBaralho->carta[j] = temp;
87
88          Carta temp2 = vBaralho[i];
89          vBaralho[i] = vBaralho[j];
90          vBaralho[j] = temp2;
91      }
92
93      free(pBaralhoAux);
94      free(cartaAux);
95
96  }
```

Figura 13. Embaralhamento

4.2. Função: descarteCarta

No funcionamento do jogo, é necessário que o jogador descarte uma carta. A função utiliza uma lógica simples, ela percorre as cartas da mão uma por uma, utilizando um loop, e verifica se cada carta é a carta que deve ser descartada. Se for a carta correta, ela é adicionada ao baralho de descarte. Caso contrário, a carta é temporariamente armazenada em um baralho auxiliar. Após percorrer todas as cartas, as cartas restantes do baralho auxiliar são transferidas de volta para a mão do jogador. Essa lógica garante que apenas a carta desejada seja descartada, mantendo as outras cartas na mão do jogador.

```
154 void descarteCarta(Baralho* pJogador, int mao, int iDescarte, Baralho* pDescarte, int nBaralho){
155     Baralho* pAux = malloc(sizeof(Baralho));
156     criarPilha(pAux);
157     Carta* cartaAux = malloc(sizeof(Carta));
158     for(int i=0; i<mao+1; i++){
159         Desempilha(pJogador, cartaAux);
160         if(iDescarte == i){
161             Empilha(pDescarte, cartaAux, nBaralho);
162         }
163         else{
164             Empilha(pAux, cartaAux, nBaralho);
165         }
166     }
167     for(int i=0; i<mao; i++){
168         Desempilha(pAux, cartaAux);
169         Empilha(pJogador, cartaAux, nBaralho);
170     }
171 }
```

Figura 14. Função descarteCarta

4.3. Função: tentativaJogo

Essa função tem como objetivo otimizar as tentativas dos jogadores, visto que são etapas que acontecem de forma repetitiva.

A função começa imprimindo a mão do jogador usando a função imprimeCartas. Em seguida, exibe a carta do topo do baralho de descarte e solicita ao jogador que faça uma decisão: escolher a carta do topo do descarte (0) ou uma carta do baralho principal (1).

```
173 void tentativaJogo(Baralho* pBaralho, Baralho* pJogador, Baralho* pDescarte, int mao, int nBaralho){
174     //IMPRESSÃO DA MÃO DO JOGADOR
175     printf("SUA MAO: ");
176     imprimeCartas(pJogador, mao-1);
177 
178     //IMPRIMIR CARTA DO TOPO E DECISAO DO JOGADOR
179     int iDecisao=0;
180     int iDescarte;
181     printf("CARTA TOPO: ");
182     imprimeCarta(pDescarte->carta[pDescarte->Topo-1]);
183     printf("CARTA TOPO (0) OU CARTA BARALHO (1): ");
184     scanf("%d",&iDecisao);
```

Figura 15. Decisão do Jogador

Se o jogador escolher a carta do topo do descarte (decisão 0), a função desempilha essa carta do baralho de descarte, armazena em uma variável auxiliar cartaAux e empilha essa carta no baralho do jogador. Em seguida, a função imprime a mão atualizada do jogador, solicita a posição da carta a ser descartada e chama a função descarteCarta para descartar a carta escolhida no baralho de descarte. Por fim, imprime a mão do jogador atualizada.

```
186     if(iDecisao == 0){
187         Carta* cartaAux = malloc(sizeof(Carta));
188         Desempilha(pDescarte, cartaAux);
189         Empilha(pJogador, cartaAux, nBaralho);
190         printf("SUA MAO: ");
191         imprimeCartas(pJogador, mao);
192         printf("POSICAO DE DESCARTE: ");
193         scanf("%d",&iDescarte);
194         descarteCarta(pJogador, mao, iDescarte, pDescarte, nBaralho);
195         printf("MAO ATUALIZADA: ");
196         imprimeCartas(pJogador, mao-1);
197     }
```

Figura 16. Decisão 0

Se o jogador escolher comprar uma carta do baralho principal (decisão 1), a função desempilha uma carta do baralho principal, armazena em cartaDesempilha e empilha essa carta no baralho do jogador. Em seguida, imprime a mão atualizada do jogador, solicita a posição da carta a ser descartada e chama a função descarteCarta para descartar a carta escolhida no baralho de descarte. Por fim, imprime a mão do jogador atualizada.

```
198     else if(iDecisao == 1){
199         Carta* cartaDesempilha = malloc(sizeof(Carta));
200         Desempilha(pBaralho, cartaDesempilha);
201         Empilha(pJogador, cartaDesempilha, nBaralho);
202         imprimeCartas(pJogador, mao);
203         printf("POSICAO DE DESCARTE: ");
204         scanf("%d",&iDescarte);
205         descarteCarta(pJogador, mao, iDescarte, pDescarte, nBaralho);
206         printf("MAO ATUALIZADA: ");
207         imprimeCartas(pJogador, mao-1);
208     }
```

Figura 17. Decisão 1

4.4. Função: verificaGanhador

A função verificaGanhador é responsável por verificar se todas as cartas na mão do jogador têm o mesmo valor. A função começa declarando duas variáveis, igual e maoJogador, ambas inicializadas com zero. Em seguida, ela percorre a mão do jogador e armazena os valores das cartas em um array chamado maoJogador.

Após armazenar os valores, a função realiza um segundo loop para comparar o primeiro valor com todos os outros valores no array maoJogador. Cada vez que encontra um valor igual ao primeiro, incrementa a variável igual.

Por fim, a função verifica se o número de cartas na mão (mao) é igual ao valor de igual. Se forem iguais, isso significa que todas as cartas têm o mesmo valor. Nesse caso, a função retorna o valor 1, indicando que o jogador ganhou.

```
98  int verificaGanhador(Baralho* pJogador, int mao, Carta cartaCoringa){
99      int igual = 0;
100     int maoJogador[3];
101     for(int i=0; i<mao; i++){
102         maoJogador[i] = pJogador->carta[i].valor;
103     }
104     for(int i=0; i<mao; i++){
105         if(maoJogador[0] == maoJogador[i]){
106             igual++;
107         }
108     }
109     if(mao == igual){
110         int i=1;
111         return i;
112     }
113 }
```

Figura 18. Decisão 1

4.5. Funções Auxiliares

Dentre todas as funções, é necessário redigir sobre as auxiliares que são implementadas. Elas tem como objetivo otimizar o código. Para o uso das cartas, foi utilizado funções como: ”quantidadeCartas”, ”imprimeCarta”e ”imprimeCartas”.

Se tratando de pilhas, foi necessário funções que contribuiam para a criação e manipulação das mesmas. Entretanto, não fugiu do que trabalhamos em sala, dessa forma,

foi alterado dentro do nosso contexto, ou seja, adaptando para nosso struct. São elas: ”Empilha”, ”Desempilha”, ”criarPilha”, ”verifiaTopoPilha”e ”verificaPilhaVazia”

5. Estudo de Complexidade

Para o Estudo de Complexidade, irei abordar as funções mais importantes do código, em que é necessário um foco maior.

5.1. Ordem de Complexidade: criarBaralho

Preencher vBaralho e empilhar cartas em pBaralhoAux: O loop for executa qCartas vezes, onde qCartas é igual a quantidadeCartas(nBaralho). Portanto, a complexidade é O(qCartas).

Verificar se cada carta em pBaralhoAux está na lista de exclusões: O loop for executa qCartas vezes. No loop interno, são feitas no máximo nExcluidos.qValores comparações. Portanto, a complexidade é O(qCartas * nExcluidos.qValores).

Empilhar cartas em pBaralho: O loop for executa qCartas vezes. Portanto, a complexidade é O(qCartas).

Algoritmo de Embaralhamento: O loop for executa (qCartas - 1) - (nExcluidos.qValores * 4) vezes. Sendo assim, complexidade é O(qCartas).

5.2. Ordem de Complexidade: verificaGanhador

Criar um array maoJogador e atribuir os valores das cartas da mão do jogador: O loop for executa mao vezes. Portanto, a complexidade é O(mao).

Verificar se todos os valores em maoJogador são iguais: O loop for executa mao vezes. Dessa forma, O(mao).

5.3. Ordem de Complexidade: descarteCarta

Desempilhar cartas do jogador e decidir se empilhar no descarte ou em pAux: O loop for executa mao+1 vezes. No pior caso, todas as cartas serão desempilhadas e uma decisão será tomada para cada uma delas. Portanto, a complexidade é O(mao).

Desempilhar cartas de pAux e empilhá-las de volta no jogador: O loop for executa mao vezes, onde mao é o tamanho da mão do jogador. Portanto, a complexidade é O(mao).

6. Testes e Resultados

6.1. Jogo

Como descrito acima, o jogo é definido em etapas, para o teste, estarei utilizando do arquivo de entrada ”Truco”. Em que há um baralho, cartas excluídas e a mão de três cartas.

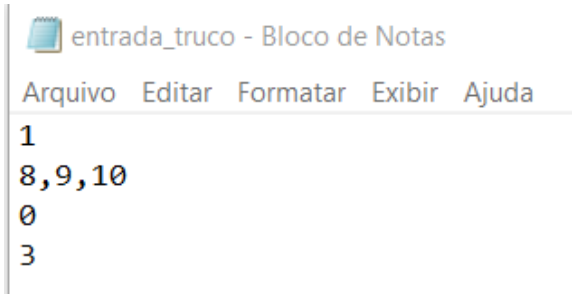

Figura 19. Entrada Truco

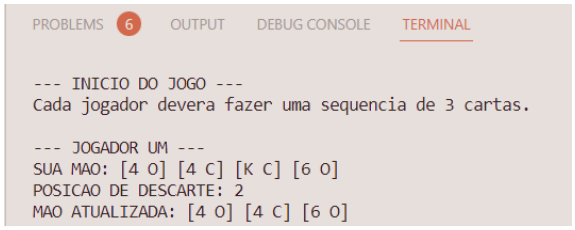

Figura 20. Primeira Jogada - Jogador 1

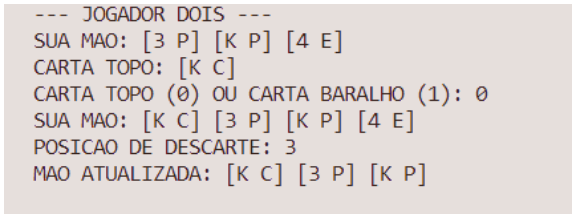

Figura 21. Tentativa Jogador 2

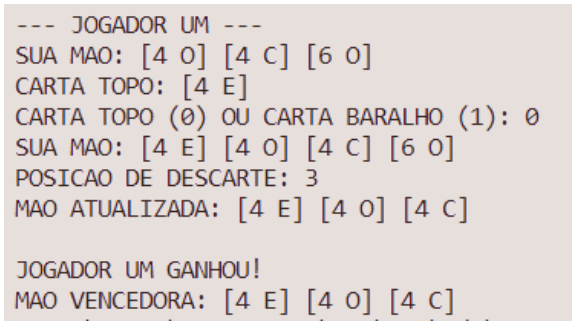

Figura 22. Tentativa Jogador 1 (VENCEDOR)

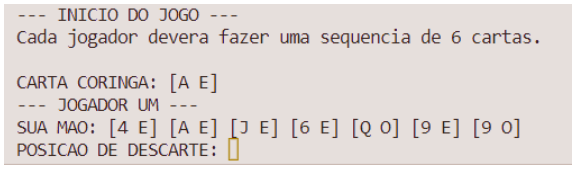

Figura 23. Entrada Genérica (Proposta no Enunciado)

7. Algoritmos de Ordenação

A ordenação proposta no enunciado, exigia o Quicksort e mais outros dois algoritmos de ordenação, em que escolhi o Selectionsort e Bubblesort. Em todos os códigos foram utilizados uma função chamada comparaCartas.

A função ComparaCartas recebe duas cartas e compara seus valores e naipes para determinar a ordem de classificação entre elas. Ela utiliza um mapeamento de naipes para números, em que a ordem de complexidade deste código é constante, ou seja, O(1). Segue a seguinte lógica:

```
267  int ComparaCartas(Carta carta1, Carta carta2) {
268      int mapping[256];
269      mapping['C'] = 1;
270      mapping['O'] = 2;
271      mapping['P'] = 3;
272      mapping['E'] = 4;
273
```

Figura 24. Mapping

1. Compara os valores das cartas. Se o valor da primeira carta for menor, retorna -1. Se for maior, retorna 1.

```
274      // Compara os naipes utilizando o mapeamento
275      if(carta1.valor < carta2.valor){
276          return -1;
277      }else if(carta1.valor > carta2.valor){
278          return 1;
279      }
280
```

Figura 25. Primeira Comparação

2. Se os valores forem iguais, compara os naipes usando o mapeamento. Se o naipe da primeira carta for menor, retorna -1. Se for maior, retorna 1.

```
281      // Se os valores forem iguais, compara os naipes
282      if(mapping[carta1.naipe] < mapping[carta2.naipe]){
283          return -1;
284      }else if(mapping[carta1.naipe] > mapping[carta2.naipe]){
285          return 1;
286      }
287
288      // Se os naipes também forem iguais, retorna 0
289      return 0;
290  }
```

Figura 26. Segunda e Terceira comparação

3. Se os naipes também forem iguais, retorna 0 para indicar que as cartas são consideradas iguais.

7.1. QuickSort

No quicksort, trabalhamos com duas funções além propriamente do Quicksort, são elas: Particao e Ordena. A função QuickSort é a função que chama a função Ordena com os parâmetros adequados para ordenar o vetor de cartas A. O parâmetro n indica o número de elementos no vetor.

```
235  void QuickSort(Carta A[MAX_ITEM_VETOR], int n){
236      Ordena(0, n-1, A);
237  }
```

Figura 27. Função: Quicksort

A complexidade do Quicksort no pior caso é O(n^2), mas no caso médio é O(n log n), onde n é o tamanho do vetor.

7.1.1. Função: Particao

A função Particao é responsável por realizar a partição do vetor. Ela recebe como parâmetros o índice da posição inicial (Esq) e final (Dir) do subvetor a ser particionado, os ponteiros i e j, que serão atualizados para indicar as posições de divisão do vetor, e o array A contendo as cartas.

```
212  void Particao(int Esq, int Dir, int *i, int *j, Carta A[MAX_ITEM_VETOR]){
213      Carta pivo, aux;
214      *i = Esq; *j = Dir;
215      pivo = A[(*i + *j)/2]; /* obtem o pivo x */
216      do{
217          while (ComparaCartas(pivo, A[*i]) > 0) (*i)++;
218          while (ComparaCartas(pivo, A[*j]) < 0) (*j)--;
219          if (*i <= *j)
220          {
221              aux = A[*i]; A[*i] = A[*j]; A[*j] = aux;
222              (*i)++; (*j)--;
223          }
224      } while (*i <= *j);
225  }
```

Figura 28. Função: Particao

Durante o loop, são feitas as seguintes comparações:

1. Enquanto a carta A[*i] for menor que o pivô, incrementa-se i (ou seja, avança para a próxima posição à direita);

2. Enquanto a carta A[*j] for maior que o pivô, decrementa-se j (ou seja, avança para a próxima posição à esquerda).

Quando os índices se cruzam, é verificado se i é menor ou igual a j. Se isso for verdade, troca-se as cartas nas posições i e j de lugar, atualizando os valores de i e j (incrementando i e decrementando j). Esse processo é repetido até que i seja maior que j.

7.1.2. Função: Ordena

A função Ordena recebe como parâmetros o índice inicial (Esq) e final (Dir) do vetor e o array A. Ela chama a função Particao para dividir o vetor em duas partes menores. Em seguida, verifica se ainda existem elementos à esquerda ou à direita da partição e chama recursivamente a função Ordena para ordenar essas partes.

```
228 void Ordena(int Esq, int Dir, Carta A[MAX_ITEM_VETOR]){
229     int i,j;
230     Particao(Esq, Dir, &i, &j, A);
231     if (Esq < j) Ordena(Esq, j, A);
232     if (i < Dir) Ordena(i, Dir, A);
233 }
```

Figura 29. Função: Ordena

7.2. SelectionSort

O algoritmo de ordenação por seleção percorre o vetor, encontrando o menor elemento em cada iteração e o coloca na posição correta. O processo se repete até que todo o vetor esteja ordenado. O algoritmo tem uma complexidade de tempo média e pior caso de O(n^2).

```
252 void SelectionSort(Carta A[MAX_ITEM_VETOR], int n){
253     int i, j, Min;
254     for(i=0; i<n-1; i++){
255         Min = i;
256         for(j=i+1; j<n; j++){
257             if(ComparaCartas(A[j], A[Min]) < 0){
258                 Min = j;
259             }
260         }
261         Carta aux = A[Min];
262         A[Min] = A[i];
263         A[i] = aux;
264     }
265 }
```

Figura 30. Função: Selectionsort

7.3. BubbleSort

O algoritmo utiliza dois loops aninhados para percorrer o vetor e comparar elementos adjacentes. A variável 'i' controla o número de iterações externas, enquanto a variável 'j' controla o número de iterações internas.

```
239 void BubbleSort(Carta A[MAX_ITEM_VETOR], int n){
240     int i, j;
241     for(i=0; i<n-1; i++){
242         for(j=0; j<n-i-1; j++){
243             if(ComparaCartas(A[j], A[j+1]) > 0){
244                 Carta temp = A[j];
245                 A[j] = A[j+1];
246                 A[j+1] = temp;
247             }
248         }
249     }
250 }
```

Figura 31. Função: Selectionsort

A cada iteração interna, o código compara duas cartas consecutivas usando a função 'ComparaCartas'. Se o resultado da comparação for maior que zero, indica que as cartas estão fora de ordem e, portanto, são trocadas de posição. O código apresentado implementa o algoritmo Bubble Sort para ordenar um vetor de cartas. A complexidade do Bubble Sort é O(n^2), onde n é o tamanho do vetor.

7.4. Resultados

A seleção do uso do método de ordenação acontece na main, em que nós colocamos qual método nós iremos utilizar através de um array. Além de ser criado o arquivo de saída, cada elemento do vetor ordenado é escrito no arquivo de saída. Utilizarei a condição do quicksort como exemplo, pois as outras duas seguem o mesmo modelo:

```
93      if(strcmp(metodo, "quicksort") == 0){
94          QuickSort(vBaralho, quantidadeCartas(nBaralho));
95          char cartaImprimida[2];
96          for(int i=0; i<quantidadeCartas(nBaralho); i++){
97              if(vBaralho[i].valor == 1){
98                  sprintf(cartaImprimida, "A%c", vBaralho[i].naipe);
99                  fprintf(arquivo_saida, "%s ",cartaImprimida);
100             }
101             else if(vBaralho[i].valor == 11){
102                 sprintf(cartaImprimida, "J%c", vBaralho[i].naipe);
103                 fprintf(arquivo_saida, "%s ",cartaImprimida);
104             }
105             else if(vBaralho[i].valor == 12){
106                 sprintf(cartaImprimida, "Q%c", vBaralho[i].naipe);
107                 fprintf(arquivo_saida, "%s ",cartaImprimida);
108             }
109             else if(vBaralho[i].valor == 13){
110                 sprintf(cartaImprimida, "K%c",vBaralho[i].naipe);
111                 fprintf(arquivo_saida, "%s ",cartaImprimida);
112             }
113             else{
114                 sprintf(cartaImprimida, "%d%c", vBaralho[i].valor, vBaralho
                    [i].naipe);
115                 fprintf(arquivo_saida, "%s ",cartaImprimida);
116             }
117         }
118         fclose(arquivo_saida);
119     }
```

Figura 32. Seleção do Quicksort

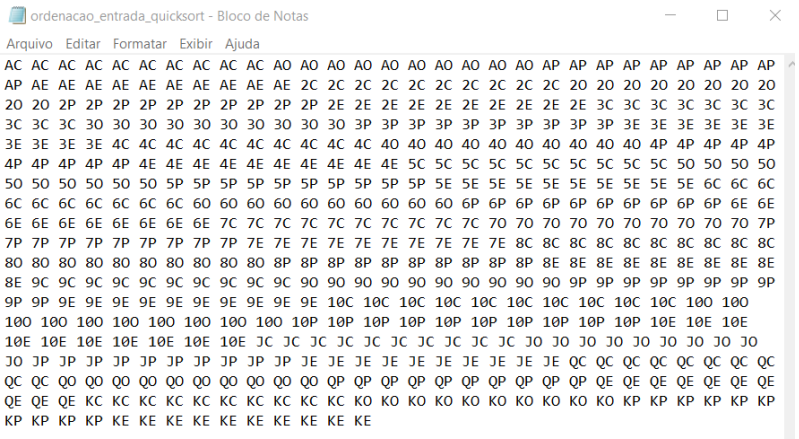

Figura 33. Quicksort: 11 Baralhos

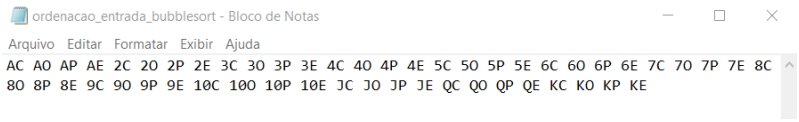

Figura 34. Bubblesort: 1 Baralhos

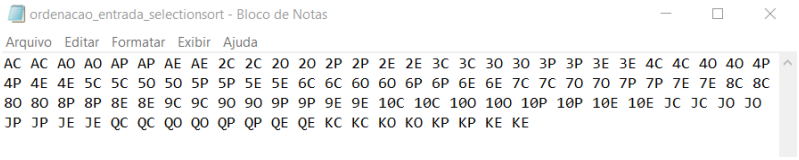

Figura 35. Selectionsort: 2 Baralhos

7.5. Tempo de Execução

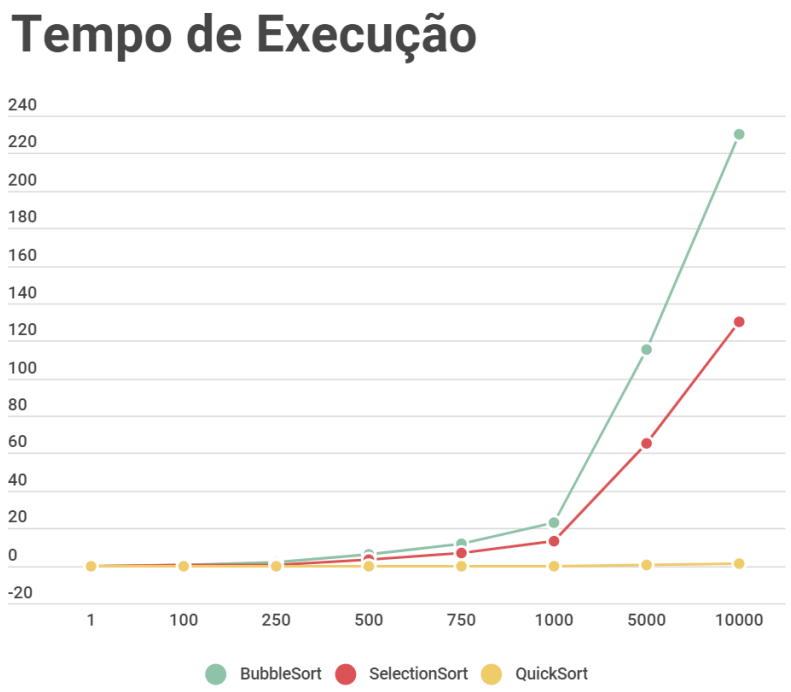

Figura 36. Quantidade de Segundos por Quantidade de Baralhos

8. Conclusão

Dado todo o código, creio que esse trabalho foi bem desafiador. A manipulação de arquivos e posições, alocação de memória e ordenação, foram as atividades mais difíceis. Entretanto, acredito ter conseguido colocar em prática o que foi aprendido em sala de aula. Ampliando os conhecimentos de Pilhas, Análise de Complexidade e Ordenação.

9. Bibliografia

https://pt.stackoverflow.com/questions/61197/map-em-c-funciona, *Mapping em C*. StackOverflow, 2015.

http://linguagemc.com.br/alocacao-dinamica-de-memoria-em-c/, *Alocação Dinâmica em C*. Eduardo Casavelha