Documentação TP1

Aluno: Henrique Daniel de Sousa

Matrícula: 2021031912

Introdução

Essa documentação tem por objetivo apresentar um programa que desenvolve um jogo que simula um jogo de Poker. Nessa aplicação, cada jogador recebe uma quantia igual de dinheiro no início e, a cada rodada, ele pode apostar ou não. A cada rodada, todo jogador participante dessa rodada recebe cinco cartas, as quais são definidas entre dez possíveis sequências: Royal Straight Flush [RSF], Straight Flush [SF], Four of a kind [FK], Full House [FH], Flush [F], Straight [S], Three of a kind [TK], Two Pairs [TP], One Pair [OP], High Card [HC]. Após definir a "mão" de cada jogador, elas são comparadas e a que tiver mais valor ganha. Caso haja empate, os vencedores dividem o pote.

Implementação

O código está organizado entre arquivos ".h", que implementam os TADs "carta", "jogador", "mesa", e arquivos ".c" que implementam as funções de cada TAD e as executam.

TAD Carta: Uma carta é definida por um número pertencente ao intervalo [1,13] e um naipe entre P (paus), E (espadas), C (copas), O (ouros).

TAD Jogador: Todo jogador possui atributos que representam seu nome, seu dinheiro, a aposta feita por ele, um vetor do tipo carta para representar sua "mão", um valor para sua "mão" e um vetor que possui as combinações de sua mão, i.e, o valor de sua trinca, dupla, quadra, caso ele tenha determinada combinação.

TAD Mesa: A mesa possui um vetor do tipo jogador, um atributo para a quantidade de jogadores no jogo, o valor do pote da rodada e os vencedores da rodada;

O programa, inicialmente recebe o número de jogadas e a quantidade de dinheiro inicial para cada jogador e, depois disso, recebe o número de jogadores n, da primeira rodada, que é o número total de jogadores. Dessa forma, uma variável do tipo "mesa" com n jogadores é inicializada. A cada rodada, contando a primeira, é uma variável auxiliar do tipo "mesa" é criada. Desse modo, as operações são realizadas na mesa auxiliar e depois as informações são substituídas na mesa principal.

Desse modo, a mesa auxiliar precisa de obter o saldo de cada jogador da rodada antes de comparar as mãos, pois, se um jogador apostar mais do que tiver ou menos do que o pingo, a rodada será invalidada. Assim, tendo passado no teste de sanidade, as mãos de cada jogador da rodada são comparadas e depois cria-se outra mesa com os vencedores da rodada, atribuindo na mesa auxiliar, o dinheiro ganhado por cada jogador.

A implementação desse problema foi realizada usando a linguagem C e compilada com o GCC. Para a documentação desse problema, a máquina utilizada tem o Windows10(WSL2) como sistema operacional, um processador Intel(R) Core(TM) i5-9300H CPU @2.40GHz 2.40 GHz e 8GB de memória RAM.

Análise de Complexidade

Inserção de jogadores:

Função '*InicializaMesa()*': Essa função aloca um espaço no vetor de jogadores para n jogadores, logo, ela é O(n).

Função '*criaJogador()*': Essa função inicializa as informações de cada jogador, logo, inserindo 5 um vetor de 5 cartas, com $O(5) = O(1)$. Depois, é usado um algoritmo de insert sort para ordenar as cartas. Nesse caso, ele é $n.O(5^2)$, para os n jogadores inseridos na mesa. Além disso, as cartas passam por uma classificação para definir o valor da combinação, com O(n) para n jogadores.

Função '*insereNaMesa()*': nessa função, cada chamada é O(1), porém, para n jogadores, ela tem O(n) como complexidade.

Logo, para inserir os jogadores, o código tem O(n) como complexidade.

Após a inserção, o vetor de jogadores é percorrido para encontrar o valor do saldo de dinheiro de cada um, com $O(n^2)$.

Sanidade da rodada:

Para garantir a sanidade da rodada, primeiro é verificado se, dentre todos jogadores, há algum sem dinheiro para o pote. Essa operação tem O(n) como complexidade.

Função *'sanidadeRodada()'*: essa função tem um loop que percorre todos jogadores da rodada e verifica três condições. Logo, tem como complexidade O(n).

Coleta do pote:

Função *'potePingo()'*: Essa função percorre todos jogadores do jogo para adicionar ao pote o valor do pingo. Portanto, essa função tem O(n) como complexidade.

Função *'poteAposta()'*: Nessa função, são coletados os valores referentes às apostas de cada jogador da rodada. Logo, um loop itera sobre todos jogadores, com complexidade O(n).

Ranqueamento de desempate:

Função *'vencedores()':* Essa função percorre o vetor de jogadores da rodada para definir a mão de maior valor e separa os jogadores que tiverem a mão desse tipo, com O(n). Caso exista mais de um jogador com esse tipo de mão, as duas mãos serão comparadas na função ***'desempateComb()'*** ou na função ***'desempateSeq()'*.** Ambas usam o mesmo algoritmo e têm a complexidade $O(n^2)$, já que são constituídas por dois loops aninhados. Nesse algoritmo, a carta escolhida é comparada entre dois jogadores do mesmo vetor e, se uma for maior que outra, o nome do jogador é substituído por '\0'.

Impressão dos resultados:

Função *'imprimeRelatorio()'*: Essa função percorre a mesa que contém os vencedores da rodada, com complexidade O(n).

Função *'ordenaMesa()'*: Essa função usa o algoritmo de insertion sort para ordenar o vetor de jogadores de acordo com seu saldo, no final do jogo. Logo, sua complexidade é $O(n^2)$.

Por fim, o nome dos jogadores e seu dinheiro final é impresso, com uma complexidade de $O(n^2)$.

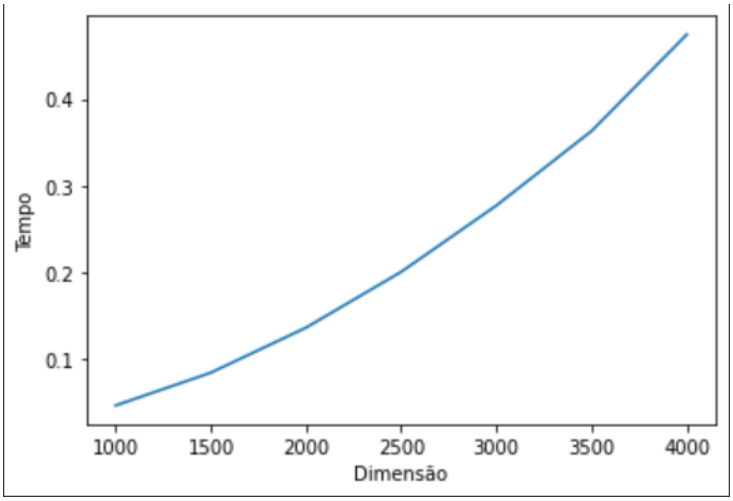
Estratégias de robustez

O programa usa a biblioteca "msgassert.h" para implementar a robustez e corretude das aplicações. Primeiramente, é verificado se os arquivos de entrada e de saída foram definidos e passados como argumentos. Posteriormente, na função ***'insereCarta()'***, para cada carta lida, é verificado se seu número pertence ao intervalo [1,13] e se seu naipe é definido por uma das letras 'P', 'E', 'C' ou 'O'.

Análise experimental

Desempenho computacional

Para a análise de desempenho computacional foram utilizadas entradas com 1000, 1500, 2000, 2500, 3000, 3500 e 4000 jogadores. Observa-se então, de acordo com o tempo de execução, o seguinte gráfico:



"Gráfico gerado usando o tempo de execução do programa e o número de jogadores, por meio das funções da biblioteca 'matplotlib', em Python"

Observa-se, então, que o programa tende a uma complexidade levemente curvada, lembrando uma parábola, i.e, tem complexidade $O(n^2)$. Porém, note que sua acentuação é bastante baixa. Isso se deve à função de ordenação da mesa de acordo com o saldo dos jogadores. Essa aplicação pode ser otimizada usando outros algoritmos de ordenação, como merge sort.

O custo da função de ordenação pode ser observado pela seguinte saída do "gprof", para uma entrada com 4000 jogadores:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 65.28      0.15     0.15        1   150.14   150.14  ordenaMesa
 30.46      0.22     0.07                             main
  4.35      0.23     0.01    24000     0.00     0.00  insereNaMesa
  0.00      0.23     0.00   100000     0.00     0.00  insereCarta
  0.00      0.23     0.00   100000     0.00     0.00  validaCarta
  0.00      0.23     0.00    39751     0.00     0.00  testS
  0.00      0.23     0.00    20000     0.00     0.00  criaJogador
  0.00      0.23     0.00    20000     0.00     0.00  identificaMao
  0.00      0.23     0.00    20000     0.00     0.00  leInsere
  0.00      0.23     0.00    20000     0.00     0.00  maioresComb
  0.00      0.23     0.00    20000     0.00     0.00  ordenaCartas
  0.00      0.23     0.00    20000     0.00     0.00  testRSF
  0.00      0.23     0.00    20000     0.00     0.00  testSF
  0.00      0.23     0.00    19997     0.00     0.00  testFK
  0.00      0.23     0.00    19965     0.00     0.00  testFH
  0.00      0.23     0.00    19950     0.00     0.00  testF
  0.00      0.23     0.00    19692     0.00     0.00  testTK
  0.00      0.23     0.00    18706     0.00     0.00  testTP
  0.00      0.23     0.00    17300     0.00     0.00  testOP
  0.00      0.23     0.00     6072     0.00     0.00  pairBetweenThree
  0.00      0.23     0.00        6     0.00     0.00  inicializaMesa
  0.00      0.23     0.00        6     0.00     0.00  limpaMesa
  0.00      0.23     0.00        5     0.00     0.00  sanidadeRodada
  0.00      0.23     0.00        1     0.00     0.00  clkDifMemLog
  0.00      0.23     0.00        1     0.00     0.00  desativaMemLog
  0.00      0.23     0.00        1     0.00     0.00  desempateComb
  0.00      0.23     0.00        1     0.00     0.00  finalizaMemLog
  0.00      0.23     0.00        1     0.00     0.00  imprimeRelatorio
  0.00      0.23     0.00        1     0.00     0.00  iniciaMemLog
  0.00      0.23     0.00        1     0.00     0.00  nomeMao
  0.00      0.23     0.00        1     0.00     0.00  parse_args
  0.00      0.23     0.00        1     0.00     0.00  pote
  0.00      0.23     0.00        1     0.00     0.00  vencedores
```

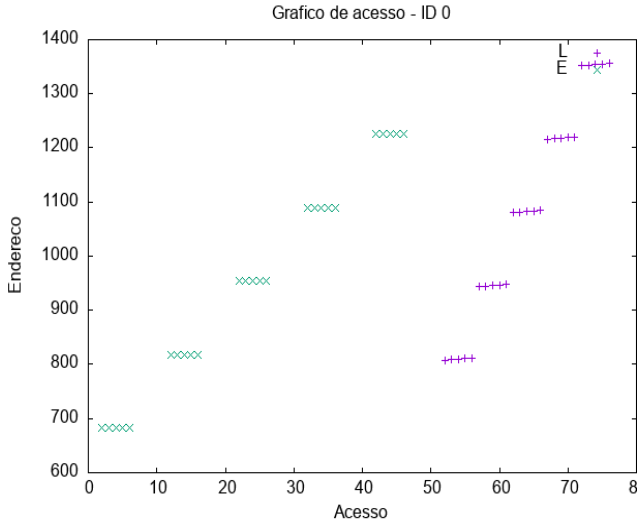
"Saida gerada pelo "gprof", aplicando uma entrada com 4000 jogadores."

Observa-se que a função mais custosa e que mais demora para ser concluída é a função ***'ordenaMesa()'*** , o que comprova a complexidade do programa como O(n²).

Registro de memória e localidade de referência

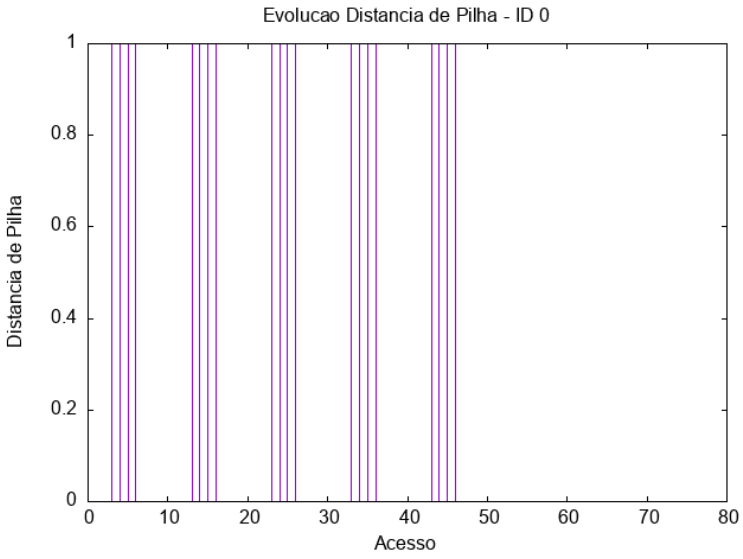
Para a análise do registro de memória foi utilizada a biblioteca *'memlog.h'* e programa *'Analisamem'*.

Assim o gráfico de acesso de memória gerado, a partir de uma entrada com 5 jogadores, teve como resultado o seguinte:



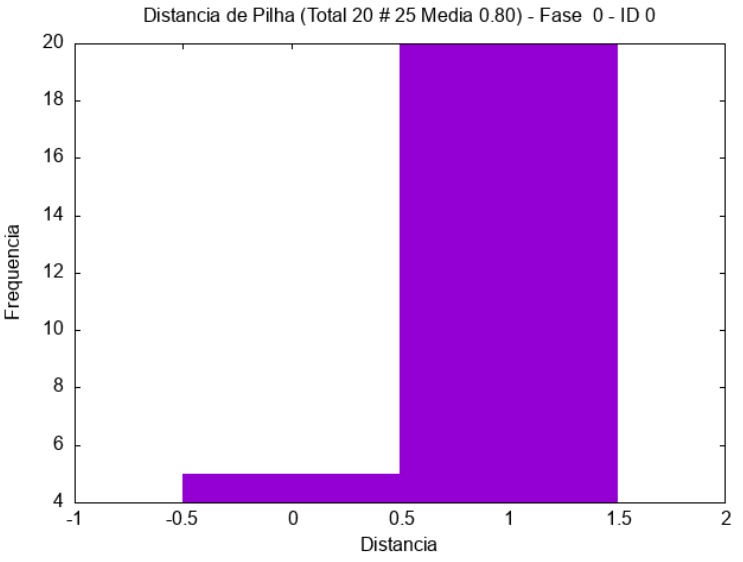
"Gráfico gerado usando o programa "gnuplot", a partir da alocação de uma mesa com 5 jogadores."

Esse gráfico representa a alocação numa variável do tipo mesa com 5 jogadores. Logo, é possível ver que, para cada jogador, um vetor de 5 posições foi alocado.



"Gráfico gerado usando o programa "gnuplot", a partir da análise dos endereços de memória alocados."

Observa-se que a distância entre os endereços de cada carta para cada jogador é muito próxima. Isso pode ser observado, também, no seguinte gráfico de distância de pilha:



"Gráfico gerado usando o programa "gnuplot", a partir da análise de distância de pilha."

Desse modo, o programa apresenta uma distância de pilha relativamente pequena, diante o problema apresentado.

Conclusão

Nesse trabalho foram criados processos de abstração de um jogo de Poker, no qual cada jogador recebia 5 cartas. Desse modo, funções

combinatórias para definir e ranquear as mãos recebidas por cada jogador foram definidas, além de funções de ordenação das cartas e dos jogadores.

Além disso, essa prática possibilitou mais aprendizados sobre o GDB Debuger, o qual foi bastante utilizado para tratar de problemas de segmentação, frequentes durante a inserção dos jogadores. Também foram aplicados algoritmos de ordenação e de comparação, usados nas funções ***'desempateComb()'*** e ***'desempateSeq()'.*** Ambas foram projetadas e desenvolvidas do início ao fim e utilizam do mesmo algoritmo, porém, a primeira utiliza um vetor auxiliar com o valor de suas combinações (triplas,duplas, etc.) e a segunda utiliza o valor de todas cartas da mão. Portanto, a maior dificuldade do projeto estava no desempate entre vários jogadores.

Vale ressaltar que esse problema possibilita um bom exercício para trabalhar a abstração. Era fundamental desenvolver tipos para a carta, para o jogador e para a mesa, pois isso facilitava muito a resolução das questões propostas e possibilitava mais legibilidade no código.

Bibliografia

https://www.geeksforgeeks.org/insertion-sort/

Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT.

Instruções para compilação e execução

1. Acesse o diretório chamado **‘TP’**.
2. Adicione o arquivo de entrada nesse diretório.
3. No Makefile, modifique o nome da entrada, em **‘ARQIN’** para o nome de seu arquivo de entrada.
4. No terminal, execute o Makefile com o comando make. A saída será gerada no diretório **‘TP’**.