TRABALHO PRÁTICO 1
POKER FACE

Tarcizio Augusto Santos Lafaiete

Universidade Federal de Minas Gerais(UFMG)
Belo Horizonte - MG - Brasil

tarcizio-augusto@hotmail.com

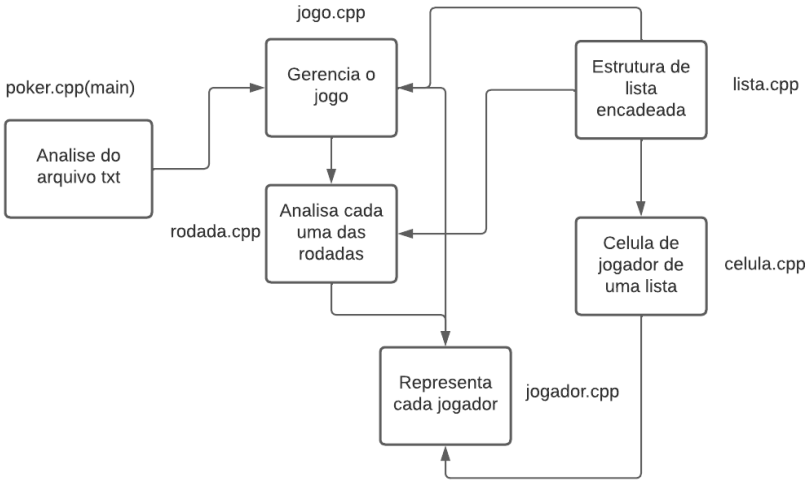
1.Introdução

O problema proposto para implementação neste trabalho prático foi o desenvolvimento de um programa que recebesse um arquivo txt de entrada com o descritivo de um jogo de poker com diversas jogadas e retornasse um arquivo txt de saída com o resultado das partidas jogadas e o resultado final do jogo, com os jogadores ordenados por ordem decrescente do montante de dinheiro que possuem ao final do jogo. Para a resolução deste problema foi seguida uma linha de dividir os passos do jogo de poker entre algumas classes com o objetivo de facilitar a implementação do código.

Esta documentação inicia na seção 2 explicando as decisões de implementação tomadas pelo aluno e explicações do funcionamento básico das classes implementadas. Em seguida na seção 3 as informações da seção 2 são complementadas trazendo as estratégias de robustez usadas pelo desenvolvedor do programa para tratar exceções. Na seção 4 tratamos da complexidade espacial e temporal de diversos métodos implementados. Assim, após finalizar a explicação do código na seção 5 é falado sobre as análises experimentais feitas em relação ao código. Por fim, na seção 6 explica-se os processo para a compilação e execução correta do programa e a seção 7 conclui a documentação com as considerações finais do autor.

2.Implementação

Para a elaboração do jogo de poker solicitado neste trabalho prático, primeiro foi-se planejado como seria o fluxo de informações partindo do arquivo de texto com os descritivos das rodadas até cada um dos participantes do jogo de poker. Com isso o esboço a seguir foi criado:



Tendo em mente este diagrama além do programa principal foram criadas cinco classes, sendo três delas diretamente relacionadas com o problema do jogo de poker, enquanto as outras duas sendo a estrutura de dados de uma lista encadeada e suas células para auxiliar na manipulação das variáveis que serão analisadas.

2.1 Estruturas de dados

Dentro da implementação do programa foi utilizada a estrutura de uma lista simplesmente encadeada para armazenar os jogadores criados pela classe jogador também implementada dentro do jogo de poker. Para fins de simplificar o uso da lista foi definido que por padrão o seus métodos de inserção e remoção funcionam semelhantes aos métodos de uma fila, ou seja, há apenas inserções no final da lista e remoções no início, o que tem como vantagem a complexidade O(1) destas duas operações. Além disso,houve a necessidade de criar um método de remoção que utilizasse o nome do jogador como chave.

Para além dos métodos de inserção e remoção, foram desenvolvidos diversos métodos para facilitar a manipulação dos jogadores, como pesquisar, verificar a existência de um jogador e alterar o seu montante. Todos estes métodos têm como característica comum utilizar o nome do jogador como chave para caminhar na lista.

2.2 Classes e main

2.2.1 Classe Jogador

Um jogador é representado dentro do programa por um nome, o montante de dinheiro que ele possui e as cartas que ele tem em mãos na rodada. Nesta classe seus principais métodos tem como objetivo definir qual jogada o participante tem em mão e quais suas cartas mais valiosas. Na definição da jogada que o jogador tem em mãos é verificado os seguintes critérios: se o jogador possui uma sequência, se ele tem cartas de mesmo valor na mão e se ele tem todas as cartas do mesmo naipe. Combinando a resposta destas três questões conseguimos chegar a definição de uma das jogadas do poker.Tendo a jogada em mãos podemos definir

com base nisso as cartas mais valiosas que serão usadas como critério de desempate posteriormente.

2.2.2 Classe Rodada

Nesta classe temos a representação de todos os acontecimentos necessários para uma rodada, assim temos um número de participantes, o pingo pago na rodada, o valor total do pote e a lista de jogadores que irão participar de uma rodada. Partindo destas informações é que iremos extrair o(s) vencedor(es) e o valor ganho por eles.

Para definição dos vencedores há a realização de chamadas sucessivas de métodos que são responsáveis por comparar a mão dos jogadores e filtrar sempre pelos mais fortes que vão sendo salvos em uma lista de ganhadores e a medida que adentramos neste filtro criado pelos métodos temos cada vez menos membros na lista até que reste apenas os reais ganhadores.

2.2.3 Jogo

Por fim, na última classe implementada, temos a classe jogo como a responsável por receber as informações vindas da main do programa e executar o jogo de poker. Nela temos uma lista que armazena todos jogadores que já participaram de uma partida e um objeto da classe rodada que executa a rodada mais recente jogada. É aqui que distribuímos da main para as outras classes os dados necessários para cada uma delas. Além disso é nesta classe que executamos o processo de escrita do resultado das rodadas e o resultado do jogo no arquivo “saida.txt”.

2.2.4 Main

Finalizando este subtópico, temos no nosso programa principal a função de capturar as informações recebidas pelo arquivo “Entrada.txt” e conversar com a classe jogo para distribuir as informações, executar as partidas e dar o resultado final do jogo. Além disso, na main realizamos alguns testes, que serão melhor explicados no tópico de estratégias de robustez, com os valores do arquivo de entrada para evitar a inserção de lixo dentro do programa.

2.3 Dados do ambiente

Este programa foi desenvolvido em C++, compilado pelo G++ da GNU Compiler Collection em computador com o processador Intel Core i3-6006 U, com 4GB de memória RAM e o Linux Mint 20.2 como sistema operacional.

3.Estratégias de Robustez

Para que o processo de análise do jogo de poker implementado no projeto funcione é necessário que haja garantias de que as entradas estão em formato correto, garantido isso o resto do código deverá funcionar da forma desejada. Por isso foram implementadas estratégias de robustez para verificar se os dados enviados para os métodos não serão lixo.

Com isso em mente, na primeira linha é realizada a análise do número de rodadas para verificar se ela é maior que zero. Em seguida é verificado se o dinheiro inicial dos participantes é maior que 50 e se ele é um múltiplo de 50.Indo

para a segunda linha, é verificado se o pingo da rodada,assim como o dinheiro inicial é múltiplo de 50, maior ou igual a 50.

Nos casos de o número de rodadas ou o dinheiro inicial ter um valor invalido o arquivo é fechado automaticamente. Já no caso do pingo da rodada a rodada é apenas invalidada.

Há também o teste de sanidade realizado pela classe Jogo para garantir que não haverá apostas acima do valor que o participante possui. Caso isso aconteça a rodada é invalidada.

4.Análise de Complexidade

As análises de Complexidade do programa serão dividas em partes, sendo então analisada a complexidade de cada método das classes até chegarmos na análise de complexidade da main.

4.1 Classe jogador

numeroCartas - complexidade de tempo : Ele possui complexidade temporal O(1), pois apesar de possuir dois loops aninhados eles tem valores de iteração constantes assim sendo repetidos sempre 5 vezes, para o loop externo e 2 ou 3 vezes para o loop interno.

numeroCartas - complexidade espacial : A complexidade espacial é O(1), pois sempre é utilizado um espaço de memória constante para alocar o array de 5 posições usados na função

naipeCartas - complexidade de tempo : Este método tem como operação mais relevante um dois laços aninhados que iteram 5 vezes no loop externo e 2 ou 3 vezes no loop interno, logo a complexidade da função é O(1).

naipeCartas - complexidade espacial : Tem complexidade espacial O(1), pois aloca de maneira constante um array de 5 posições.

cartaMesmoValor - complexidade de tempo: Possui complexidade de tempo igual a O(1), pois sua operação mais relevante é um laço que constantemente repete 5 vezes.

cartaMesmoValor - complexidade espacial: Sua complexidade é O(1), já que realiza alocações constantes em suas chamadas.

cartasSequencia - complexidade de tempo: Este método tem como operações mais relevantes um loop simples de complexidade O(n) e um algoritmo de ordenação pelo método bolha que sabe-se ter complexidade $O(n^2)$, logo tempos $O(n) + O(n^2) = O(n^2)$. Contudo este loop sempre tem como n o número 5, ou seja, independente da entrada ele realiza a mesma quantidade de iterações, assim sua complexidade é O(1).

cartasSequencia - complexidade espacial: Como este método sempre aloca a mesma quantidade de memória independente da entrada, ele é O(1).

cartasMesmoNaipe - complexidade de tempo: Ele é O(1) em complexidade temporal, uma vez que possui como operação um laço que se repete de forma constante sempre 5 vezes.

cartasMesmoNaipe - complexidade espacial:

jogada - complexidade de tempo: Possui complexidade temporal O(1), pois suas operações mais relevantes são as chamadas dos métodos numeroCartas (O(1)), numeroNaipes (O(1)) e cartasSequencia (O(1)), logo temos O(1) + O(1) + O(1) = 3O(1) = O(1).
jogada - complexidade espacial: Tem complexidade O(1), uma vez que sempre é alocado o mesmo espaço de memória para o método.
cartaMaisForte - complexidade de tempo: Ele tem complexidade O(1), uma vez que em todos os casos diferentes de cada jogada as instruções rodam de maneira constante.
cartaMaisForte - complexidade espacial: Possui uma complexidade O(1), já que sempre é alocado o mesmo espaço de memória para este método.
cartaDesempate - complexidade de tempo: Em todos os diferentes casos de jogada temos uma complexidade O(1), uma vez que sempre repetem as mesmas operações de forma constante.
cartaDesempate - complexidade espacial: Ela possui complexidade O(1), pois independente da entrada sempre é alocado o mesmo espaço de mémoria.

4.2 Classe ListaJogadores

pesquisaJogador e existeJogador - complexidade de tempo: Tem a complexidade temporal O(n) no pior caso quando o elemento pesquisado está no último item lista ou não se encontra na lista e O(1) no melhor caso quando o jogador procurado está na primeira célula.
pesquisaJogador e existeJogador - complexidade espacial: Possui complexidade O(1), já que este método aloca apenas um jogador auxiliar para realizar as comparações.
insereJogador - complexidade de tempo: A complexidade de tempo é O(1), uma vez que a inserção necessita apenas movimentar o ponteiro do último elemento.
insereJogador - complexidade espacial: A complexidade espacial é O(1), já que é alocado apenas o espaço para adicionar mais um jogador na lista.
removeJogador - complexidade de tempo: Tem complexidade temporal igual a O(1), já que a remoção é feita no início da lista, assim só precisando mover o ponteiro do primeiro para o próximo e remover a antiga primeira célula.
removeJogador - complexidade espacial: Tem complexidade espacial O(1), devido alocação de memória constante e independente da entrada.
removeJogadorNome - complexidade de tempo: Sua complexidade de tempo é O(n) no pior caso, pois caminha na lista através de um laço que repete n vezes até terminar a lista ou encontrar o jogador na última posição e no melhor caso é O(1) quando o jogador está na primeira posição da lista.
removeJogadorNome - complexidade espacial: Sua complexidade espacial é O(1), uma vez que sempre o mesmo espaço alocado para o método.
limpa - complexidade de tempo: Tem complexidade de tempo igual a O(n), pois chama o método removeJogador, que tem complexidade O(1),n vezes.

limpa - complexidade espacial: Tem complexidade espacial O(1), devido a apesar de receber a saída de removeJogador n vezes, ela o faz em uma única variável jogador.
debitaJogador - complexidade de tempo: Sua complexidade temporal no melhor caso é O(1), quando o jogador procurado se encontra na primeira célula analisada e tem como pior caso O(n) quando o jogador está na última célula ou não se encontra na lista, pois é realizado o caminhamento na lista n vezes.
debitaJogador - complexidade espacial: Sua complexidade espacial é O(1), uma vez que apenas aloca para memória um jogador auxiliar para executar as operações do método.
achaMaisRico - complexidade de tempo: Possui complexidade temporal O(n), pois precisa realizar através de um laço a comparação entre todos os elementos da lista para definir o jogador com o maior montante.
achaMaisRico - complexidade espacial: Possui complexidade espacial O(1), pois aloca na memória apenas um jogador auxiliar para realizar as comparações dentro do laço.

4.3 Classe Rodada

montaPote - complexidade de tempo: Este método tem complexidade temporal O(n), pois este algoritmo é composto por um laço que sempre itera através dos n participantes.
montaPote - complexidade espacial: Tem complexidade espacial O(1), uma vez que ocupa espaço de memória apenas para alocar o somatório das n apostas dos jogadores.
insereJogadorRodada - complexidade de tempo: A complexidade temporal deste método é O(1), uma vez que ele apenas chama o método da classe lista insereJogador, que como foi explicado anteriormente possui complexidade O(1).
insereJogadorRodada - complexidade espacial: Tem complexidade espacial O(1) devido a complexidade do método insereJogador.
jogadaVencedora - complexidade de tempo: Possui complexidade de tempo igual a O(1), pois a operação mais valiosa é a chamada do método jogada da classe jogador, que como sabemos tem complexidade O(1).
jogadaVencedora - complexidade espacial: Tem complexidade O(1), pois aloca um jogador auxiliar que é O(1) e aloca espaço para a execução do método jogada que também é O(1), assim temos O(1) + O(1) = O(1).
limpaVencedores - complexidade de tempo: Tem uma complexidade igual a O(n), pois realiza apenas a chamada do método limpa da classe lista, que já foi dito ter O(n) como complexidade temporal.
limpaVencedores - complexidade de espacial: Tem complexidade O(1) devido a chamada do método limpa que é O(1).
criterioDesempateCartaFinal - complexidade de tempo: Este método tem como operações relevantes um laço com complexidade O(n) e a chamada da função insereJogador da classe lista que possui complexidade O(1), sendo assim O(n) + O(1) = O(n).

criterioDesempateCartaFinal - complexidade espacial: Tem complexidade O(n), pois se utiliza de uma lista de Jogadores auxiliar que precisa suportar n jogadores.
criterioDesempateCartaForte - complexidade de tempo: As operações relevantes deste método são um loop for com complexidade O(n) e a chamada do método criterioDesempateCartaFinal de complexidade O(n) também, com isso tempos O(n) +O(n) = 2O(n) = O(n).
criterioDesempateCartaForte - complexidade espacial: Tem complexidade O(n), uma vez que se utiliza de uma lista auxiliar para alocar os n jogadores vencedores e precisa do espaço de execução para criterioDesempateCartaFinal que é O(n), temos então O(n) + O(n) = O(n).
criterioDesempateCartaJogada - complexidade de tempo: Ele tem como operações relevantes a chamada do método criterioDesempateCartaForte de complexidade O(n) e um loop for de complexidade O(n), assim tempos O(n) + O(n) = 2O(n) = O(n).
criterioDesempateCartaJogada - complexidade espacial: Sua complexidade espacial é O(n) devido ao uso de uma lista de Jogadores auxiliar para ajudar na alocação de n jogadores e também do espaço para a execução de criterioDesempateCartaForte de complexidade O(n), temos então O(n) + O(n) = O(n).
decideJogador - complexidade de tempo: Este método possui como operações mais relevantes um loop de complexidade O(n) e a chamda do método criterioDesempateCartaJogada que possui complexidade O(n), logo teremos O(n) + O(n) = 2O(n) = O(n).
decideJogador - complexidade espacial: Tem complexidade espacial igual a O(n), pois apesar de alocar apenas o espaço para 2 jogadores, também precisa alocar o espaço para a execução do método criteriosDesempateCartaJogada que é O(n), assim temos O(n) + O(1) = O(n).
Vencedores - complexidade de tempo: Este método tem como instrução mais relevante a chamada do método decideGanhador logo sua complexidade é dada por decideGanhador, que sabe-se ser de complexidade O(n), então Vencedores tem complexidade temporal igual a O(n).
Vencedores - complexidade espacial: Tem complexidade O(n) devido a execução de decideJogador que é O(n).

4.4 Classe Jogo

criaJogador - complexidade de tempo: Este método possui complexidade temporal O(n), pois se existejogador estiver seu melhor caso que é ter um jogador na primeira posição da lista, será chamado em seguida os métodos pesquisaJogador,que terá complexidade igual a de existerJogador por funcionarem pelo mesmo algoritmo, setMao da classe jogador de complexidade O(1) e insereJogadorRodada da classe rodada de complexidade O(1) assim teremos: O(1) + O(1) + O(n) + O(1) = O(n). Já no caso de existeJogador estiver em seu pior caso que é O(n) ele irá chamar os métodos insereJogador da classe listaJogadores e insereJogadorRodada, assim teremos O(N) + O(1) + O(1) = O(n).

criaJogador - complexidade espacial: Tem complexidade espacial O(1), pois todos os seus métodos alocam O(1) de espaço de memória para serem executados.
criaRodada - complexidade de tempo: Possui complexidade O(1), uma vez tem como única instrução a chamada do construtor de rodada.
criaRodada - complexidade espacial: Tem complexidade espacial O(1), uma vez aloca apenas espaço para receber a construção de uma rodada.
AtualizaAposta - complexidade de tempo: Tem a sua operação mais relevante sendo a chamada do método debitaJogador, assim tendo O(1) quando debitaJogador estiver no seu melhor caso e O(n) quando este método estiver em seu pior caso.
AtualizaAposta - complexidade espacial: Tem complexidade espacial O(1), pois sua alocação mais relevante é para debitaJogador que tem complexidade O(1).
testeSanidade - complexidade de tempo: Este método tem como operações relevantes um if e a chamada do metódo pesquisaJogador da listaJogadores, assim tendo no melhor O(1) + O(1) = O(1) e pior caso O(n) + O(1) = O(n).
testeSanidade - complexidade espacial: Por apenas realizar a alocação de memória para um jogador auxiliar tem complexidade espacial O(1).
restituiJogadores - complexidade de tempo: Sua complexidade temporal é O(n), pois seu algoritmo é composto por um laço de complexidade O(n).
restituiJogadores - complexidade espacial: Tem complexidade O(1), por alocar espaço apenas para a execução de debitaJogador.
preparaPote - complexidade de tempo: Este metódo possui como única operação a chamada do método montaPote da rodada Atual, sendo assim como sabemos que sua complexidade temporal era O(n), preparaPote então é O(n).
preparaPote - complexidade espacial: Como este método aloca espaço apenas para chamar montaPote, ele possui complexidade O(1) igual a montaPote.
pagaGanhadores - complexidade de tempo: Ele chama o método Vencedores de complexidade O(n), em seguida chama valorGanho de complexidade O(1), tem um loop de complexidade O(n), no qual há as chamadas de debitaJogador de O(n) e getJogador de O(n), por fim o método termina com a chamada de limpaVencedores. Assim teremos a complexidade de pagaGanhadores igual a O(n) + O(1) + O(n).(O(n)+O(n)) + O(n) = 2O(n) + O(n²) + O(1) = O(n²).
pagaGanhadores - complexidade espacial: Tem compleixdade O(n), uma vez que precisa alocar uma lista auxiliar que precisa conter n elementos, além disso suas outras alocações são O(1).
imprimePartida - complexidade de tempo: Este método chama os métodos Vencedores de O(n), númeroVencedores e valorGanho de O(1), jogadaVencedora de O(n²), limpaVencedores de O(n) e um laço de complexidade O(n) no qual há a chama de getJogador de O(n), então teremos que imprimePartida é igual a O(n) + O(1) + O(1) + O(n²) + O(n).O(n) + O(n) = O(n²).
imprimePartida - complexidade espacial: Devido a necessidade de alocar espaço para uma lista de Jogadores auxiliares de n posições então sua compleixidade é O(n).

imprimeResultado - complexidade de tempo: Neste metódo temos como operações relevantes um laço externo de complexidade O(n), dentro deste laço há a chamada de achaMaisRico de complexidade O(n) e removeJogadorNome também de O(n), logo teremos O(n).(O(n) + O(n)) = O(n²).
imprimeResultado - complexidade espacial: Este método tem complexidade espacial O(1), pois apenas aloca espaço constante para dois jogadores auxiliares.
imprimeRodadaInvalida - complexidade de tempo: Este método somente imprime um aviso de rodada inválida no arquivo de “saida.txt”, sendo assim tem complexidade O(1).
imprimeRodadaInvalida - complexidade espacial: Tem complexidade O(1), pois apenas imprime uma string.

4.5 Main(poker.cpp)

Na main do jogo poker nós temos um fluxo de dados no qual utilizamos 3 laços aninhados para enviar os dados de "entrada.txt" para dentro dos métodos da classe jogo. Analisando essa estrutura de dentro para fora, temos como loop mais interno o loop, no qual são armazenadas as cartas na mão do jogador, sendo assim temos uma estrutura O(1), já que repete sempre 5 vezes independente da entrada, juntamente com este loop temos a chamada dos métodos criaJogador, que possui complexidade O(n), o testeSanidade de complexidade O(n) também e dependendo do resultado do teste temos a chamada do método atualizaAposta ou restituiJogadores ambos O(n).

Todo este processo de composição de mão e a chamada destes métodos ocorrem dentro de um loop que é executado n vezes, sendo n igual ao número de participantes, com isso temos então que a complexidade deste loop é O(n).O(n) = O(n²), Juntamente com este loop é executado, caso não haja falha do teste de sanidade, os métodos preparaPote de complexidade O(n), pagaGanhadores de O(n²) e imprimePartida de O(n²). Caso haja uma falha no teste, só é chamado o método imprimeRodadaInvalida de complexidade O(1).

Finalmente indo para o laço mais externo do programa, teremos que ele executa todo o processo anteriormente citado n vezes, sendo n neste caso o número de rodadas, como sabemos que a operação mais custosa que ele executa é O(n²) podemos então afirmar que sua complexidade é O(n) . O(n²) = O(n³). Juntamente desse loop temos a chamada do método ImprimiResultado de complexidade O(n²), assim temos que a complexidade do programa é dada por O(n²) +O(n³) = O(n³), logo a sua complexidade temporal é O(n³).

5.Análise de Experimentos

5.1 Desempenho computacional de tempo

Os experimentos realizados para medir o desempenho computacional do jogo de poker foram feitos utilizando a biblioteca de c, time.h, com ela criamos duas variáveis start e end do tipo clock_t. O clock_t é uma estrutura de dados implementada pela biblioteca time.h, no qual as variáveis deste tipo são capazes de

guardar o número de sinais de clock utilizados pelo processador até o momento pelo programa. Então a start foi chamada logo antes do início do algoritmo implementado na main e a end no final do código.

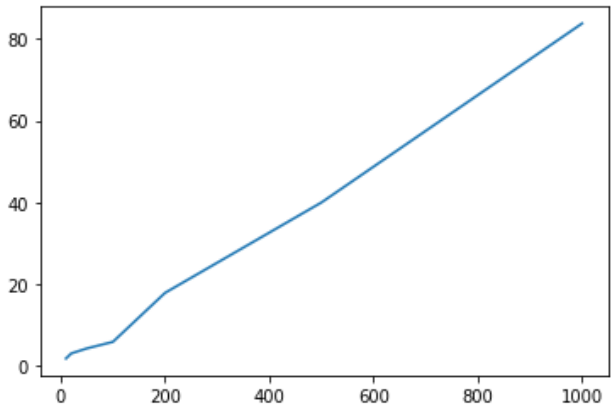
Com esses valores armazenados, fazemos a diferença deles e dividimos por CLOCKS_PER_SEC que é um define da biblioteca time.h que representa o valor do número de clocks que o processador gera por segundo. Dessa divisão então conseguimos o tempo de execução do programa.

Explicado como foi feita a medição de tempo, agora pode-se detalhar como e em quais condições foram feitas as medidas. Estas medidas foram feitas levando em consideração uma entrada padrão de 10 rodadas onde o número de jogadores varia de 2 a 10. Em seguida replicamos esta mesma saída para termos entradas de 20, 50, 100,500 e 1000 rodadas e em cada saída com o computador sem nenhuma aplicação aberta foram medidos o tempo de execução do programa 5 vezes para cada tamanho de entrada.

Deste experimento conseguimos a seguinte tabela:

Entrada	Medida 1	Medida 2	Medida 3	Medida 4	Medida 5	Média
10	1,297ms	1,31ms	1,339ms	0,971 ms	3,973ms	1,778ms
20	2,117ms	2,313ms	2,622ms	1,823ms	7,225ms	3,020ms
50	3,981ms	4,542ms	4,982ms	3,517ms	4,223ms	4,251ms
100	5,914ms	7,024ms	5,858ms	3,445ms	7,046ms	5.857ms
200	13,351ms	17,254ms	22,702ms	17,285ms	16,689ms	17,856ms
500	40,301ms	38,712ms	39,445ms	43,228ms	38,272ms	39,992ms
1000	82,017ms	100,019ms	80,159ms	79,065ms	77,881ms	83,828ms

Utilizando então os dados apresentados na tabela e as funções da biblioteca matplot da linguagem python, criamos um gráfico com o tamanho das entradas como eixo x e a média de tempo obtida como eixo y. Assim tivemos este gráfico:



A princípio pode parecer estranho o fato da função do tempo de execução do programa se comportar de forma linear, uma vez que ao se analisar o programa

definimos sua complexidade temporal como O(n³). Contudo se analisarmos estes “n’s” que compõe esta complexidade iremos perceber que um deles é correspondente as n iterações que criaJogador, testeSanidade,AtualizaAposta, e restituiJogador realiza que são dependentes do número de participantes. Outro dos “n’s” é o próprio número de participantes e este n varia de forma limitada tendo seu intervalo entre 2, que é o número mínimo para se jogar uma partida, e 10 que é o número maximo de jogadores visto que com 52 cartas do baralho podemos apenas distribuir 5 cartas para 10 participantes. Logo o último n que corresponde a ao número de rodadas é o único que realmente pode tender ao infinito, sendo assim o crescimento do tempo de execução sofre uma influência muito maior deste parâmetro do que dos outros 2, tornando então o comportamento da complexidade semelhante a uma complexidade O(n).

6. Instruções de compilação

- Entre no diretório poker
- Usando o arquivo Makefile, execute o comando make all.
- Coloque o arquivo txt de entrada junto do makefile e das outras pastas
- Entre na pasta bin
- Execute o programa com o comando ./tp.out

7.Conclusão

Neste trabalho foi desenvolvido um software que analisa um arquivo .txt de entrada, e distribui seus dados entre as classes implementadas para a realização de um jogo de poker a fim de definir os ganhadores de cada rodada e ao final gerar um ranking com todos os jogadores em ordem decrescente do valor ganho. Durante a execução do trabalho grandes aprendizados foram adquiridos tanto pelos erros, quanto pelos acertos durante o processo. Além do óbvio ganho de prática com o desenvolvimento de programas do zero.

Com o trabalho prático o autor pode ganhar mais habilidade e entendimento sobre a alocação dinâmica e pensar em quais locais ela deve ser implementada e em quais seu uso só traria problemas devido ao maior cuidado que se deve tomar ao manusear ponteiros no código. Um importante aprendizado também foi o manuseio da estrutura de uma lista encadeada e aproveitar do que foi apresentado nas aulas de Estrutura de Dados para expandir esta TAD de forma a atender as necessidades do programa de uma forma inteligente usufruindo das características intrínsecas de uma lista encadeada.A depuração de código utilizando ferramentas como o gdb, também pode ser exercitada durante o desenvolvimento do trabalho.

Em linhas gerais apesar dos contratempos e dos intermináveis erros a cada nova mudança nas implementações do código, a experiência de ter a liberdade de pensar e arquitetar um software mesmo que ainda não muito complexo foi realmente positiva e muito diferente daquilo que foi visto dentro da matéria de Programação e Desenvolvimento de Software 2.