Trabalho Prático 1

Poker Face

Carlos Henrique Brito Malta Leão

Matrícula: 2021039794

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte – MG – Brasil

chbmleao@ufmg.br

1. Introdução

Pôquer é um famoso jogo de azar que alimenta o imaginário de muitos com promessas de riqueza rápida e, infelizmente, muitas pessoas não estão dispostas a gastar suas economias dessa forma. Pensando nisso, o problema proposto por esse trabalho prático foi a criação de um sistema onde qualquer um pode jogar, mesmo aqueles que não sabem quase anda das regras do jogo. Dessa forma, esta documentação explica como foi criado e estruturado esse sistema, em que o programa cuida da etapa de decisão, onde vê-se quem ganhou, e do montante de dinheiro virtual que cada jogador terá ao final das diversas rodadas.

Essa documentação tem como objetivo explicitar como foi realizada a implementação desse sistema, além de realizar uma análise explicativa sobre como o sistema do jogo funciona. Além disso, também serão explicitados a robustez, abstração e desempenho do programa, possibilitando um melhor entendimento de como o sistema funciona na prática.

Para resolver o problema supracitado, foi criado um programa na linguagem C++, que utiliza três principais classes: *Card*, *Player* e *Round*, que representam, respectivamente, uma carta comum de baralho, um jogador e uma rodada de pôquer. Além disso, também são utilizadas algumas classes auxiliares: *Vector*, *Node* e *List*, que representam um vetor alocado dinamicamente e uma implementação de lista encadeada alocada dinamicamente, utilizando a classe *Node* para representar cada nó encadeado. Ademais, temos a estrutura *memlog*, que auxilia no processo de análise de desempenho do programa e as funções *msgassert*, que são utilizadas de forma a aumentar a robustez do programa.

Por fim, ao decorrer dessa documentação, alguns aspectos sobre o trabalho serão melhor explicados, como a descrição da implementação na seção 2 e a análise de complexidade na seção 3. Em seguida, na seção 4 e 5 serão explicitadas as estratégias de robustez e a análise experimental. Por fim, teremos a conclusão do trabalho, sumarizando o que foi aprendido durante seu desenvolvimento.

2. Método

Durante essa seção, será realizada uma descrição da implementação do programa estruturado na linguagem C++. Dessa forma, serão detalhadas as estruturas de dados, classes e funções implementadas.

2.1 Estruturas de Dados

A implementação do programa teve como base duas estruturas de dados, um vetor dinâmico e uma lista encadeada alocada dinamicamente. Primeiramente, o vetor foi escolhido para armazenar conjuntos de objetos com uma quantidade estática, já que durante o processo de leitura do arquivo de entrada, entrada.txt, já sabemos a quantidade de rodadas, jogadores e cartas, o que torna desnecessária a implementação de uma estrutura de dados dinâmica. Porém, não há como saber, incialmente, o número de ganhadores de cada jogada, por isso, foi escolhida a implementação de uma lista encadeada que possui custo constante para a remoção de elementos no fim dela.

2.1.1 Vetor

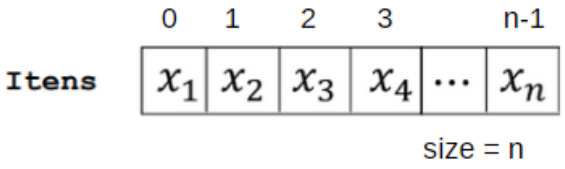
Primeiramente, o vetor foi implementado utilizando alocação estática, recebendo

um tamanho máximo. Dessa forma, essa estrutura permite acesso aleatório a qualquer posição em tempo $O(1)$, além de permitir percorrer a lista em ambas direções caso necessário, começando na posição 0 e terminando com o i-ésimo item na posição $i - 1$. Além disso o vetor é implementado de forma parametrizada, utilizando *templates*, dessa forma, é possível armazenar sequências de rodadas, jogadores ou cartas.

Por fim, esta classe apresenta algumas funções específicas, que abstraem e aumentam a robustez do código:

- *Vector():* Inicializa o objeto vetor com atributos inválidos.
- *Vector(int size):* Inicializa o objeto vetor com tamanho igual ao recebido como parâmetro e aloca o espaço na memória para os itens que serão inseridos no vetor.
- *~Vector():* Destrutor da classe, atribui valor inválido para o tamanho do vetor.
- *getItems():* Retorna o vetor de itens.
- *accessVector():* Função utilizada exclusivamente para o processo de análise de desempenho do programa, acessando todos os itens alocados no vetor.
- *writeElement(T item):* Armazena um item na próxima posição do vetor que ainda não foi armazenada, e soma 1 ao atributo position, para armazenar a posição do próximo item a ser inserido.
- *readElement(int pos):* Retorna o elemento da posição desejada acessando a posição do item direto no vetor.
- *printVector():* Função utilizada exclusivamente no processo de desenvolvimento para a depuração do código. Ela chama a função *print()* de cada um dos itens armazenados no vetor (só funciona para vetores de jogadores ou cartas).
- *bubbleSort():* Ordena vetores de jogadores (de acordo com seu dinheiro final) ou cartas (de acordo com sua numeração). O *Bubble Sort* é um algoritmo de ordenação que funciona da seguinte forma: é utilizado um loop que itera as posições do vetor, assim, a posição atual é comparada com a posição anterior, se a posição atual for menor, é realizada a troca dos valores nessa posição. Caso contrário, não é realizada a troca, apenas passa-se para o próximo par de ordenações.
- *swapCard e swapPlayer:* Funções auxiliares da função *bubbleSort* que trocam os itens de posição para sua ordenação.

Um diagrama esquemático do vetor estático implementado pode ser visto na *Figura 1*.



2.1.2 Lista Encadeada

Em seguida, a implementação da lista encadeada demandou uma complexidade maior que o vetor. Nela, itens da lista são armazenados em posições não contíguas da memória, dessa forma, foi necessária a utilização de células (classe *Node*) que são encadeadas usando apontadores. Essa lista utiliza alocação dinâmica, permitindo o crescimento e redução de tamanho, o que é muito útil para armazenar os vencedores de cada rodada, já que cada rodada pode apresentar mais de um vencedor.

Por fim, essa classe apresenta alguns atributos privados, como *size*, que armazena o tamanho atual da lista, *first,* que armazena a primeira célula, e *last*, que armazena a última célula da lista. Ademais, a classe apresenta também algumas funções específicas, que abstraem e aumentam a robustez do código:

- *List():* Inicializa a lista com tamanho 0 e cria um nós que será tanto o atributo *first* quanto o atributo *last*, ou seja, essa célula será o começo e o fim da lista, já que a

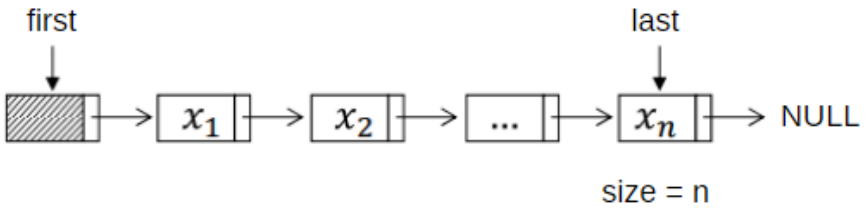
lista está vazia.

- *~List():* Chama a função clean (que será melhor explicada a seguir) e exclui a primeira célula da lista.
- *getSize():* Retorna o atributo de tamanho atual da lista.
- *isEmpty():* Retorna se a lista está vazia.
- *SetPositon(int pos):* Cria um nó auxiliar p, com o endereço da primeira posição, que será iterado por toda a lista encadeada. Essa iteração será realizada utilizando um loop que será iterado o número da posição que deve ser retornado, todas as vezes atribuindo para p o endereço do próximo item da lista. Ao final, o nó p é retornado pela função. Em alguns processos, é necessário receber o nó anterior ao desejado, então é possível retornar esse nó também.
- *getPlayer(int pos):* Retorna o item localizado na posição indicada utilizando a função *setPosition().*
- *insertAtStart(Player* player):* Insere o item na primeira posição. Primeiramente, uma "nova célula" é criada e inicializada com o endereço do objeto jogador que deseja inserir na lista. Dessa forma, o atributo "próximo" da primeira célula é atribuído ao "próximo" da "nova célula". Em seguida, o "próximo" da primeira célula é atribuído o endereço na "nova célula". Por fim, o tamanho da lista é incrementado. Caso o "próximo" da "nova célula" for um endereço nulo, significa que a fila só possui um elemento, então o novo final da fila é atribuído para a "nova célula".
- *insertAtEnd(Player* player):* Insere o item na última posição. Primeiramente, uma "nova célula" é criada e inicializada com o endereço do objeto jogador que deseja inserir na lista. Dessa forma, no atributo "próximo" da última célula é atribuído o endereço da "nova célula". Em seguida, o endereço da última posição da célula é mudado para o endereço da "nova célula". Por fim, o tamanho da lista é incrementado.
- *insertAtPosition(Player* player, int pos):* Insere o item na posição indicada. Primeiramente, é preciso usar a função *setPosition* para posicionar na posição anterior a desejada e atribuir a célula retornada a variável "p". Em seguida, o endereço do objeto do jogador é atribuído a "nova célula", e o seu "próximo" apontado para o "próximo" da célula "p". Por fim, o "próximo" de "p" é apontado para a "nova célula" e o tamanho da lista é incrementado. Caso o "próximo" da "nova célula" seja um endereço nulo, é preciso apontar o endereço "último" para a "nova célula".
- *removeAtStart():* Remove o item da primeira posição. Primeiramente, o endereço da célula a ser removida é armazenado na variável "p", então o endereço do "próximo" da primeira célula é apontado para o "próximo" de "p" e o tamanho da fila é incrementado. Caso o "próximo" da primeira célula seja nulo, então o último elemento também é o primeiro elemento. Por fim, a célula "p" é excluída e o item da célula excluída é retornado.
- *removeAtEnd():* Remove o item da última posição. Primeiramente, diferente da remoção ao início, é preciso utilizar a função *setPosition* para encontrar o elemento anterior ao último "p". Em seguida, é atribuído nulo ao "próximo" de "p" e o tamanho da lista decrementado. Por fim, o último elemento é excluído, o endereço do último da lista é apontado para p e o item da célula excluída é retornado.
- *removeAtPosition():* Remove o item da posição indicada. Primeiramente, assim como a remoção da última célula, é preciso usar a função *setPosition* para posicionar "p" na posição anterior a célula que deseja remover. Em seguida, fazemos com que a célula auxiliar "q" aponte para o "próximo" de "p" e o próximo de "p" aponte para o "próximo" de "q", dessa forma, a célula alvo já foi retirada da lista. Ao final, basta excluir a célula "q" e verificar se o "próximo" endereço de "p" é nulo, caso seja, é preciso atribuir o endereço do último termo da lista para o endereço de "p". Por fim, o item da célula excluída é retornado.
- *search(string name):* Retorna o item com o nome igual ao que está sendo procurado. A busca é realizada passando por todos os valores da lista até que ela acabe. Caso encontre um jogador com o nome recebido como o parâmetro, a função termina e ele é retornado, caso contrário a célula passa para a próxima da lista. Caso o loop chegue

ao final da lista e não encontre nenhum jogador com o nome igual ao recebido como parâmetro, um jogador com atributos inválidos é retornado.

- *print():* Função utilizada exclusivamente para o processo de depuração do código. Chama a função *print* de todos os jogadores armazenados na lista.
- *clean():* Função que limpa a lista encadeada, excluindo todas as suas células.
- *setPosition(int pos):* Função privada auxiliar que participa do processo de inserção e remoção de itens.
- *bubbleSort():* Ordena a lista de vencedores em ordem alfabética utilizando o algoritmo de ordenação bolha.

O diagrama a seguir (*Figura 2*) explicita visualmente a lista encadeada implementada.



2.2 Classes

Para modularizar a implementação, e tornar o programa mais abstrato e prático para o usuário, foram construídas seis classes. Três delas já foram apresentadas no tópico anterior, sendo elas: *List*, *Node* e *Vector*, representando respectivamente a lista encadeada, a célula da lista e o vetor. Além dessas, também são de extrema importância as classes: *Card*, *Player* e *Round*, que representam uma carta de baralho, um jogador e uma rodada do jogo. Analisaremos cada uma delas de forma separada.

2.2.1 *Card*

A classe *Card* abstrai uma carta de baralho do mundo real, ela apresenta dois atributos principais, sendo eles o número e o naipe da carta. Cada carta pode assumir um número de 1 a 13, em que o número 1 representa o As, os números de 2 até 10 representam os mesmos no baralho, e os números 11, 12 e 13 representam o valete, a dama e o rei, respectivamente. Ademais, existem quatro possíveis valores para o naipe, Copas, Espadas, Paus ou Ouros, representados no programa pelos caracteres C, E, P e O.

Nesse sentido, essa classe apresenta algumas funções importantes que precisam ser melhor explicadas:

- *Card():* Inicializa um objeto carta com número, naipe e chave inválidos, utilizado para inicializar objetos auxiliares.
- *Card(int number, char naipe):* Inicializa um objeto carta atribuindo o número e naipe recebidos como parâmetros.
- *~Card():* Destrutor da classe que torna seus atributos inválidos.
- *getKey():* Retorna a chave da carta chamada.
- *setNumber():* Atribui um novo valor para o atributo de número da carta.
- *getNumber():* Retorna o atributo de número da carta.
- *setNaipe():* Atribui um novo valor para o atributo de naipe da carta.
- *getNaipe():* Retorna o atributo de naipe da carta.
- *print():* Função utilizada para depuração do código que imprime o número e o naipe da carta.

2.2.2 *Player*

A classe player abstrai um jogador de pôquer do mundo real, em que ele possui um nome, uma quantidade de dinheiro virtual para apostar, sua aposta e uma mão de cinco cartas (um vetor de *Cards* de tamanho 5). O nome do jogador pode ser composto, seu dinheiro deve ser maior que zero e sua aposta em cada rodada de ver ser múltipla de 50 e positiva. Essa classe apresenta algumas funções muito importantes para o funcionamento do jogo de pôquer, em que o objeto jogador sabe decidir qual a melhor jogada que pode ser feita utilizando sua mão de cartas, adicionar e decrementar sua quantia de dinheiro, além de outras funções.

Dessa forma, explicaremos melhor o comportamento do objeto jogador e suas funções no programa:

- *Player():* Inicializa um objeto jogador com nome, dinheiro inicial e chave inválidos.
- *Payer(string name, double money, int bet):* Inicializa um objeto jogador atribuindo o seu nome, dinheiro inicial e aposta. A sua mão de cartas também é inicializada com tamanho cinco.
- *~Player():* Destrutor da classe que atribui valores inválidos para os atributos do jogador.
- *getKey():* Retorna o valor da chave (identificador) do jogador.
- *getName():* Retorna o nome do jogador.
- *setName(string name):* Atribui um novo valor para o nome do jogador.
- *getMoney():* Retorna o valor do dinheiro atual do jogador.
- *setMoney(double money):* Atribui um valor positivo para a quantidade de dinheiro atual do jogador.
- *decreaseMoney(int value):* Decrementa certo valor da quantidade atual de dinheiro do jogador.
- *addMoney(int value):* Adiciona certo valor à quantidade atual de dinheiro do jogador.
- *setBet(int bet):* Atribui um valor positivo múltiplo de 50, ao atributo de valor de aposta do jogador.
- *getBet():* Retorna o valor apostado pelo jogador.
- *getHand():* Retorna o vetor de cartas do jogador, sua mão de cartas.
- *print():* Função utilizada exclusivamente para depuração do código. Imprime o identificador, nome, dinheiro final, jogada e mão de cartas do jogador.
- *insertCardOnHand(Card card):* Insere carta recebida como parâmetro no vetor de cartas chamando a função writeElement do vetor de cartas.
- *resetHand():* Faz a posição de inserção de cartas resetar, ou seja, a mão antiga de cartas será sobrescrita por novas cartas. Utilizado quando inicia uma nova rodada.
- *sortCards():* Ordena a mão de cartas do jogador em ordem crescente em relação ao número da carta. Para isso, a função do vetor bubbleSort é chamada.
- *setMove():* Atribui um novo valor ao atributo jogada da classe. Esse valor é encontrado chamando a função *chooseMove*, que será explicada nos próximos tópicos.
- *getMove():* Retorna o número da jogada encontrado.
- *getMoveString():* Retorna o código da jogada encontrado. (HC, OP, TP, TK, S, F, FH, FK, SF, RSF).

Ao decorrer do jogo, a classe jogador, ao receber suas cartas, pode escolher uma entre nove possíveis jogadas, cada uma com seu devido valor e código no jogo. As possíveis jogadas são as seguintes:

Royal Straight Flush [RSF]: São 5 cartas seguidas do mesmo naipe do 10 até ao Ás;

Straight Flus [SF]: São 5 cartas seguidas do mesmo naipe que não seja do 10 até ao Ás;

Four of a kind [FK]: São 4 cartas iguais;

Full House [FH]: Uma tripla e um par;

Flush [F]: São 5 cartas do mesmo naipe sem serem seguidas;

Straight [S]: São 5 cartas seguidas sem importar o naipe;

Three of a kind [TK]: São 3 cartas iguais mais duas cartas diferentes;

Two Pairs [TP]: São 2 pares de cartas;

One Pair [OP]: São 2 cartas iguais e três diferentes;

High Card [HC]: Ocorre quando as cartas não encaixam em nenhuma das jogadas acima.

As três seguintes funções são utilizadas para determinar qual jogada o jogador escolherá, preferindo sempre a melhor opção de jogada.

- *chooseMove():* Função principal para a escolha da jogada do jogador. Primeiramente, as cartas são ordenadas chamando a função *sortCards*. Em seguida, é utilizado um loop que itera todas as cinco cartas da mão do jogador.

  Nesse sentido, dentro do loop, é verificado se existem pares, tripla e quadra na mão do jogador. Primeiramente, para verificar a existência de pares, é verificado se a carta anterior possui o mesmo número da carta atual, caso verdadeiro, significa que existe um par na mão. A iteração continua, caso encontre outra carta igual, a quantidade de pares é decrementada e a quantidade de triplas aumenta, já que o antigo par passou a ser uma tripla. No mesmo sentido, ocorre algo semelhante se encontrar mais uma carta igual, a quantidade de triplas é decrementada e a quantidade de quadras aumenta. Dessa forma, ao final da iteração, sabemos a quantidade de pares, triplas e quadras.

  Ao final dessa iteração já é possível retornar **FK**, se houver uma quadra, ou retornar **FH**, se existir uma tripla e um par. Isso ocorre porque os dois outros casos mais valiosos não podem ocorrer se existir uma tripla ou uma quadra.

  Em seguida, precisamos verificar se todas as cartas apresentam o mesmo naipe. Para isso, utilizaremos outro loop que também iterará sobre a mão do jogador, armazenando a quantidade de Ouros, Espadas, Paus e Copas presentes no vetor. Ao final das iterações, caso a quantidade de qualquer um dos naipes seja igual a cinco (o tamanho da mão), significa que todas as cartas possuem o mesmo naipe e precisamos entrar em um caso especial, implementado na função *verifyFlush*.

- *verifyFlush():* Função auxiliar que verifica a existência de três possíveis jogadas. Primeiramente, é verificado se a primeira posição da mão é um Ás e se a segunda é um 10, caso positivo, a função já começa a "suspeitar" de um caso de Royal Straight Flush, e atribui o valor 2 para a variável "i" que será utilizada para a iteração. Mais um loop é utilizado, nele é verificado se a diferença do valor da carta atual com a última carta é igual a 1, ou seja, verifica se a mão apresenta uma sequência, caso contrário, a função retorna **F**. Caso a iteração termine, significa que a mão apresenta uma sequência de cartas, nesse caso, se a última carta for um rei (valor 13), a função retorna **RSF**, caso contrário, retorna **SF**.

  Dessa forma, já garantimos a possibilidade de mais três possíveis jogadas, todos os casos possíveis de Flush. Em seguida, caso nenhum caso de Flush seja identificado, é preciso verificar se é possível se existe uma sequência de cartas sem importar o naipe. Para isso, é utilizada a função *verifyStraight*.

- *verifyStraight():* Função auxiliar que verifica a existência de uma sequência de cartas sem importar o naipe. Nesse sentido, o procedimento é muito semelhante ao da função *verifyFlush*, em que se a diferença do valor do item atual com o valor do item anterior for diferente de 1, a função retorna falso. Caso o loop termine, a função retorna verdadeiro.

  Nesse sentido, se a função retorna verdadeiro, a função *chooseMove* retorna **S**.

  Em seguida, faltam apenas algumas jogadas para serem verificadas. Se a mão apresenta uma tripla, a função retorna **TK**. Se a mão apresenta dois pares, a função retorna **TP**. Se a mão apresenta apenas um par, a função retorna **OP**. Caso nenhuma das condições anteriores sejam verdadeiras, a função retorna **HC**. Dessa forma, todas as 9 possibilidades de jogada foram analisadas e podem ser retornadas.

- *getCardIndex(Card card):* Função que recebe um objeto tipo carta e retorna o index dele

na mão do jogador. Para isso, é utilizado um loop que itera todos os itens da mão do jogador, caso o número e o naipe da carta sejam iguais, a função retorna a variável “i” que está sendo utilizada para a iteração no loop. Caso o loop termine, a função retorna o inteiro -1.

2.2.3 *Round*

A classe *Round* abstrai uma rodada de pôquer no mundo real. Nesse sentido, um objeto de rodada apresenta, como atributos, um número de jogadores, uma aposta mínima, o total de dinheiro arrecadado, um vetor de jogadores da rodada e uma lista encadeada de vencedores da rodada. Nesse caso, é possível utilizar um vetor para armazenar os dados dos jogadores, mas para armazenar os vencedores, é preciso utilizar uma lista encadeada, já que durante a criação da classe *Round* não é possível saber a quantidade de vencedores. A quantidade de jogadores por rodada deve ser maior que zero e a aposta mínima da rodada deve ser maior que 0 e múltipla de 50.

Dessa forma, explicaremos a seguir o comportamento do objeto rodada e suas funções no programa:

- *Round():* Inicializa um objeto rodada com valores inválidos.
- *Round(int numberOfPlayers, int minimumBet):* Inicializa um objeto rodada atribuindo a quantidade de jogadores e a aposta mínima da rodada. Além disso, a quantia arrecadada é atribuída com o valor zero inicialmente, além de inicializar o vetor de jogadores e a lista encadeada de vencedores.
- *insertPlayer(Player *player):* Utiliza a função do vetor *writeElement* para adicionar um novo jogador ao vetor de jogadores da rodada.
- *print():* Função utilizada exclusivamente para o processo de depuração do programa. Chama a função *print* de cada um dos jogadores no vetor de jogadores da jogada.
- *writePlayers(ofstream &outputFile):* Função que imprime no arquivo recebido como parâmetro o nome dos jogadores ordenados de acordo com sua quantia de dinheiro. Para isso, primeiro utiliza-se a função *bubbleSortPlayers,* que ordena os jogadores. Em seguida, é utilizado um loop que itera por todos os jogadores da lista de jogadores da rodada, imprimindo no arquivo de saída o nome e dinheiro virtual do jogador.
- *receiveBets():* Função que transfere o dinheiro dos jogadores para a quantia total da rodada. Para isso, é utilizado um loop que itera todos os jogadores da rodada e utiliza a função *decreaseMoney*, para diminuir o valor da aposta do jogador de sua quantia virtual. Em seguida, a função *addTotalAmmount* é utilizada para adicionar à quantia total da rodada o valor retirado da quantia virtual do jogador.
- *getNumberOfPlayers():* Retorna o atributo de número de jogadores da rodada.
- *getMinimumBet():* Retorna o atributo de aposta mínima da rodada.
- *getTotalAmmount():* Retorna o atributo de total arrecadado de apostas durante a rodada.
- *addTotalAmmount(int bet):* Adiciona um valor de aposta para o atributo de total arrecadado durante a rodada.
- *bubbleSortPlayers():* Ordena o vetor de jogadores os colocando em ordem decrescente de quantidade de dinheiro virtual. Para isso, a função do *bubbleSort* é chamada.
- *getWinner():* Determina o vencedor ou vencedores da rodada. Esta função é mais complexa e demanda a utilização de várias funções durante sua execução.

  Primeiramente, é preciso saber qual a melhor jogada da rodada. Para isto, é utilizado um loop que itera todos os jogadores da rodada. Nesse sentido, cada jogador tem a sua jogada determinada pela função já explicada na seção 2.2.2, a chamada da função *setMove*. Dessa forma, já possuímos o valor da jogada do jogador, então é utilizado um algoritmo simples de encontrar o valor máximo de um vetor, para encontrar o maior valor de jogada da rodada.

  Em seguida, outro loop de iteração é criado para verificar qual jogador possui o mesmo valor de jogada que o maior valor de jogada da rodada. Dessa forma, se o jogador atual possui um valor de jogada igual ao maior valor da rodada, esse jogador é inserido ao final da lista de vencedores, utilizando a função *insertAtEnd*. Se o tamanho da lista é

igual a 1, significa que só existe um jogador com o valor máximo de jogada, então é possível retornar a lista e encerrar a função. Caso contrário, existe mais de um jogador com possibilidade de vitória, dessa forma, é preciso realizar um processo de desempate.

O processo de desempate ocorre da seguinte forma, é realizado um loop em que um jogador é comparado com todos os outros jogadores. Para realizar essa comparação, já que todos os jogadores da lista apresentam o mesmo valor de jogada, é preciso utilizar a função auxiliar de desempate *breakTheTie* e a função *auxBreakTheTie*.

- *auxBreakTheTie(Player* player1, Player* player2, int mostSignificantPosition):* Recebe dois objetos de jogadores como parâmetros e um inteiro que indica a posição mais relevante. Essa função trata, de forma recursiva, o último critério de desempate, em que se verifica a carta de valor mais alto fora da jogada. A posição recebida como parâmetro é importante, ela indica quais cartas devem ser comparadas, dessa forma, na primeira chamada, a função compara as duas maiores cartas dos jogadores, em seguida, as segundas maiores cartas, e assim por diante, até a mão de cartas acabar. Se encontrar que a carta do jogador 1 é maior que a do jogador 2, a função retorna 1, caso contrário, a função retorna 2. Caso as duas cartas tenham o mesmo valor, a função recursiva é chamada novamente com a posição decrementada em um.
- *breakTheTie():* Recebe dois objetos de jogadores como parâmetros, em que, dependendo do valor da jogada desses jogadores, existe um método de desempate diferente, que é tratado utilizando um switch. Se o jogador 1 vencer o desempate, a função retorna 1, se o jogador 2 vencer, a função retorna 2, se o empate continuar, a função retorna 0. Os casos de desempate serão melhor explicados em seguida:

  HC: Caso ocorra um empate com essa jogada, é preciso verificar qual jogador possui a carta mais alta, apenas verificando a carta mais alta pela função *auxBreakTheTie*.

  OP: Caso ocorra um empate com essa jogada, ganha aquele que possuir o maior par, caso permaneça o empate, ganha aquele que possuir a maior carta fora do par. Para isto, é percorrido um loop iterando as cartas da mão dos jogadores, até encontrar duas cartas seguidas que apresentam o mesmo valor, ou seja, até encontrar o par. Em seguida, é preciso comparar os valores dos pares dos dois jogadores e retornar o número de quem ganhar a comparação. Caso os valores dos pares dos dois jogadores sejam iguais, é preciso chamar a função *auxBreakTheTie*.

  TP: Caso ocorra um empate com essa jogada, ganha aquele que possuir o maior par maior, caso permaneça o empate, ganha aquele que possuir o maior par menor, caso permaneça o empate, ganha aquele que possuir a carta mais alta. Para isto, as posições do vetor de cartas que compõe o maior e menor vetor, respectivamente, serão as posições 2 e 4, para qualquer caso da existência de dois pares. Dessa forma, basta primeiro comparar as cartas dos dois players de posição 4, se elas ainda forem iguais, é preciso comparar as cartas da posição 2 e, por fim, caso a disputa continue empatada, é preciso chamar a função *auxBreakTheTie*.

  TK: Caso ocorra um empate com essa jogada, ganha aquele jogador com a maior tripla, caso permaneça o empate ganha aquele que possuir a carta mais alta. Para isto, para encontrar o valor das cartas da tripla, basta verificar o valor da carta do meio do vetor, de posição 3. Em seguida, basta comparar o valor das cartas de posição 3 dos dois jogadores. Caso siga empatado, é preciso chamar a função auxBreakTheTie.

  S: Caso ocorra um empate com essa jogada, é preciso verificar qual jogador possui a carta mais alta, apenas verificando a carta mais alta pela função *auxBreakTheTie*.

  F: Caso ocorra um empate com essa jogada, é preciso verificar qual jogador possui a carta mais alta, apenas verificando a carta mais alta pela função *auxBreakTheTie*.

  FH: Caso ocorra um empate com essa jogada, é preciso verificar qual jogador possui a trinca de valor mais alto, caso o empate permaneça ganha aquele que possuir o maior par. Para isto, conseguimos a valor da trinca facilmente ao acessar o elemento do meio do vetor de cartas. Já o valor do par pode ser encontrado ao comparar uma carta que possui um valor diferente da trinca. Dessa forma, é possível comparar os valores das trincas dos dois jogadores, e, se necessário comparar os valores dos pares dos dois jogadores.

  FK: Caso ocorra um empate com essa jogada, é preciso verificar qual jogador possui a

maior quadra, caso permaneça empate, ganha aquele que possuir a carta mais alta. Para encontrar o valor da quadra, basta verificar o valor da carta do meio do vetor, em seguida, comparar as quadras dos dois jogadores. Se o empate permanecer, é preciso chamar a função *auxBreakTheTie*.

SF: Caso ocorra um empate com essa jogada, é preciso verificar qual jogador possui a carta mais alta, apenas verificando a carta mais alta pela função *auxBreakTheTie*.

RSF: Caso ocorra um empate com essa jogada, não existe critério para desempate, então a função retorna 0.

Dessa forma, todos os possíveis casos de desempate foram tratados e é possível dar continuidade a explicação da função *getWinner*. Se a função de desempate retornar 1, é preciso remover o jogador 2 da lista de vencedores. Ademais, caso a função de desempate retornar 2, é preciso remover todos os jogadores que já foram comparados anteriormente e não foram retirados da lista de vencedores. Porém, caso a função retorne 0, significa que o empate continuou, então nenhum dos dois jogadores são removidos da lista de vencedores e a variável de iteração "i" é somada. Dessa forma a função termina e a lista de vencedores termina apenas com os "verdadeiros" vencedores.

- *writeWinners(ofstream &outputFile):* Imprime os dados dos vencedores no arquivo de saída. Para isto, primeiramente imprime a quantidade de vencedores, pela função *getSize* da lista de vencedores, em seguida o total de dinheiro da rodada é divido entre os vencedores e esse valor também é impresso no arquivo. Além disso, o código da jogada dos vencedores é impresso. Em seguida, um loop percorre o vetor de ganhadores imprimindo o nome de cada um deles.

3 Análise de Complexidade

Esta seção apresenta a análise de complexidade de tempo e espaço para a execução de um jogo completo de pôquer, desde a criação das cartas, jogadores e rodadas até a execução da rodada e distribuição do dinheiro aos vencedores. Para um melhor entendimento, realizaremos a análise da complexidade de tempo primeiro e em seguida a análise de espaço.

3.1 Análise de Complexidade de Tempo

Para entender melhor a funcionalidade do programa, primeiramente analisaremos a análise de complexidade de cada função de forma individual, para em seguida analisar a complexidade do programa em si como um todo. Além disso, para uma melhor organização, analisaremos uma classe implementada de cada vez.

LIST

A classe *List* é uma lista encadeada que armazena n elementos, jogadores. Dessa forma, utilizaremos a quantidade de elementos n para realizar a análise de complexidade.

- *List():* Inicializa todos os atributos da classe *List*, $O(1)$. Dessa forma, teremos a complexidade assintótica da função como: $O(1)$.
- *~List():* Destrutor da classe que chama a função *clean*, $O(n)$ e exclui a primeira célula da lista, $O(1)$. Dessa forma, teremos a complexidade assintótica da função como: $O(1) + O(n) = O(\max(1, n)) = O(n)$.
- *getSize():* Apenas retorna um atributo, $O(1)$. Dessa forma, teremos a complexidade assintótica da função como: $O(1)$.
- *isEmpty():* Apenas realiza uma comparação, $O(1)$. Dessa forma, teremos a complexidade assintótica da função como: $O(1)$.
- *setPosition():* Realiza uma iteração até chegar na posição desejada dessa forma, o melhor caso é $O(1)$, caso procure o primeiro elemento, e o pior caso é $O(n)$, caso procure o último elemento da lista. Dessa forma, teremos a complexidade assintótica da função como: Melhor caso: $O(1)$ e Pior caso: $O(n)$.

- *getPlayer():* Inicializa um objeto da classe *Node*, $O(1)$, e chama a função *setPosition*, melhor caso $O(1)$ e pior caso $O(n)$. Dessa forma, teremos a complexidade assintótica da função como Pior caso: $O(1) + O(n) = O(\max(1, n)) = O(n)$ e Melhor caso: $O(1) + O(1) = O(1)$.
- *insertAtStart():* Realiza operações constantes em tempo $O(1)$, apenas mudando o endereço de algumas variáveis. Dessa forma, teremos a complexidade assintótica da função como $O(1)$.
- *insertAtEnd():* Realiza operações constantes em tempo $O(1)$, apenas mudando o endereço de algumas variáveis. Dessa forma, teremos a complexidade assintótica da função como $O(1)$.
- *insertAtPostion():* Chama a função *setPosition* com melhor caso $O(1)$ e pior caso $O(n)$. Além disso, realiza operações constantes em tempo $O(1)$, apenas mudando o endereço de algumas variáveis. Dessa forma, teremos a complexidade assintótica da função como Pior caso: $O(1) + O(n) = O(\max(1, n)) = O(n)$ e Melhor caso: $O(1) + O(1) = O(1)$.
- *removeAtStart():* Realiza operações constantes em tempo $O(1)$, apenas mudando e excluindo o endereço de algumas variáveis. Dessa forma, teremos a complexidade assintótica da função como $O(1)$.
- *removeAtEnd():* Chama a função *setPosition* com melhor caso $O(1)$ e pior caso $O(n)$. Além disso, realiza operações constantes em tempo $O(1)$, apenas mudando e excluindo o endereço de algumas variáveis. Dessa forma, teremos a complexidade assintótica da função como Pior caso: $O(1) + O(n) = O(\max(1, n)) = O(n)$ e Melhor caso: $O(1) + O(1) = O(1)$.
- *removeAtPosition():* Chama a função *setPosition* com melhor caso $O(1)$ e pior caso $O(n)$. Além disso, realiza operações constantes em tempo $O(1)$, apenas mudando e excluindo o endereço de algumas variáveis. Dessa forma, teremos a complexidade assintótica da função como Pior caso: $O(1) + O(n) = O(\max(1, n)) = O(n)$ e Melhor caso: $O(1) + O(1) = O(1)$.
- *search():* Realiza operações constantes em tempo $O(1)$. Além disso, chama a função setName e imEmpty, ambas $O(1)$. Por fim, realiza um loop que itera os elementos da lista até encontrar o elemento desejado, melhor caso $O(1)$ e pior caso $O(n)$. Dessa forma, teremos a complexidade assintótica da função como Pior caso: $O(1) + O(1) + O(1) + O(n) = O(\max(1, n)) = O(n)$ e Melhor caso: $O(1) + O(1) + O(1) + O(1) = O(1)$.
- *print():* Realiza operações constantes em tempo $O(1)$. Além disso, realiza um loop que itera todos os elementos da lista, $O(n)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + O(n) = O(\max(1, n)) = O(n)$.
- *clean():* Realiza operações constantes em tempo $O(1)$. Além disso, realiza um loop que itera todos os elementos da lista, $O(n)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + O(n) = O(\max(1, n)) = O(n)$.
- *bubbleSort():* Realiza operações constantes em tempo $O(1)$. Além disso, realiza dois loops que itera todos os elementos da lista, $O(n)$. Porém, como a lista de vencedores possui no máximo 4 jogadores, a complexidade assintótica da função é $O(1)$.

VECTOR

A classe *Vector* é um vetor que armazena n elementos, que podem ser endereços de jogadores, cartas ou rodadas. Dessa forma, utilizaremos a quantidade de elementos n para realizar a análise de complexidade.

- *Vector():* Inicializa todos os atributos da classe *Vector*, $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *~Vector():* Realiza operações constantes com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *getItems():* Realiza operações constantes com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *accessVector():* Realiza operações constantes com tempo $O(1)$. Além disso, realiza uma

iteração que percorre todos os elementos do vetor, $O(n)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + O(n) = O(\max(1, n)) = O(n)$

- *writeElement():* Realiza operações constantes com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *readElement():* Realiza operações constantes com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *printVector():* Realiza operações constantes com tempo $O(1)$. Além disso, realiza uma iteração que percorre todos os elementos do vetor, $O(n)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + O(n) = O(\max(1, n)) = O(n)$
- *bubbleSort():* Realiza operações constantes com tempo $O(1)$. Além disso, apresenta dois laços aninhados, que compara todos os elementos do vetor com todos os elementos do vetor, assim, possui tempo $O(n^2)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + O(n^2) = O(\max(1, n^2)) = O(n^2)$
- *swapCard e swapPlayer:* Realiza operações constantes com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$

CARD

Como veremos a seguir, a classe *Card* é muito simples, ou seja, não apresenta funções muito complexas. Dessa forma, todas as suas funções apresentam complexidade de tempo constante, igual a $O(1)$.

- *Card():* Realiza operações constantes com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *~Card():* Realiza operações constantes com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *getKey():* Realiza uma operação constante com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *setNumber():* Realiza uma operação constante com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *getNumber():* Realiza uma operação constante com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *setNaipe():* Realiza uma operação constante com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *getNaipe():* Realiza uma operação constante com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *print():* Realiza uma operação constante com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$

PLAYER

A classe jogador, é um pouco mais complexa que a classe de cartas, em questão do comportamento de suas funções. Porém, em questão de complexidade assintótica, ambas classes são bem semelhantes, já que a classe jogador também apresenta complexidade $O(1)$ em todas as suas funções. Isso ocorre já que a classe de jogadores sempre lida com cinco cartas, o que faz seu custo de tempo ser constante.

- *Player():* Realiza uma operação constante com tempo $O(1)$.
- *~Player():* Realiza uma operação constante com tempo $O(1)$.
- *getKey():* Realiza uma operação constante com tempo $O(1)$.
- *getName():* Realiza uma operação constante com tempo $O(1)$.
- *setName(string name):* Realiza uma operação constante com tempo $O(1)$.

- *getMoney():* Realiza uma operação constante com tempo $O(1)$.
- *setMoney(double money):* Realiza uma operação constante com tempo $O(1)$.
- *decreaseMoney(int value):* Realiza uma operação constante com tempo $O(1)$.
- *addMoney(int value):* Realiza uma operação constante com tempo $O(1)$.
- *setBet(int bet):* Realiza uma operação constante com tempo $O(1)$.
- *getBet():* Realiza uma operação constante com tempo $O(1)$.
- *getHand():* Realiza uma operação constante com tempo $O(1)$.
- *print():* Realiza uma operação constante com tempo $O(1)$. Além disso, chama a função *printVector*, que geralmente apresenta tempo $O(n)$. Porém, como o vetor que está sendo chamado é um vetor de cartas, que sempre possui 5 cartas, a complexidade de tempo dessa função passa a ser $5 * O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + 5O(1) = O(1)$
- *insertCardOnHand(Card card):* Chama a função *writeElement*, que apresenta complexidade de tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *resetHand():* Realiza uma operação constante com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *sortCards():* Chama a função *bubbleSort* que apresenta tempo $O(n^2)$. Dessa forma, teremos a complexidade assintótica da função como $O(n^2)$
- *verifyFlush():* Realiza operações constantes com tempo $O(1)$. Além disso, realiza um laço que é percorrido apenas 4 vezes, assim, possui complexidade de tempo $4O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + 4O(1) = O(1)$
- *verifyStraight():* Assim como a função *verifyFlush*, realiza operações constantes com tempo $O(1)$. Além disso, realiza um laço que é percorrido apenas 4 vezes, assim, possui complexidade de tempo $4O(1)$. Dessa forma, teremos a complexidade assintótica da função $O(1) + 4O(1) = O(1)$
- *chooseMove():* Realiza operações constantes com tempo $O(1)$. Além disso, primeiramente realiza um laço que é percorrido apenas 4 vezes, assim possui complexidade de tempo $4O(1)$. Em seguida, mais um laço é percorrido 4 vezes com tempo $4O(1)$. Por fim, as funções verifyFlush e verifyStraight são chamadas, ambas com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + 4O(1) + 4O(1) + 2O(1) = O(1)$
- *setMove():* Chama a função *chooseMove* que apresenta tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *getMove():* Realiza uma operação constante com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *getMoveString():* Realiza operações constantes com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *getCardIndex():* Realiza operações constantes com tempo $O(1)$. Além disso, realiza um laço percorrendo 4 elementos, com tempo $4O(1)$. Dessa forma, teremos a complexidade assintótica da função como $4O(1) + O(1) = O(1)$

ROUND

- *Round():* Realiza operações constantes com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *insertPlayer():* Chama a função do vetor *writeElement* com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *print():* Utiliza um laço que é percorrido pelo número de jogadores da rodada, cada vez chamando a função *print* de jogador com tempo $O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$

- *writePlayers():* Chama a função *bubbleSortPlayers* com tempo $O(n^2)$. Além disso, percorre um laço n vezes, com tempo $O(n)$. Por fim, realiza algumas operações constantes com tempo $O(1)$. $O(n^2) + O(n) + O(1) = O(\max(n^2, n, 1)) = O(n^2)$. Como o número de jogadores não pode passar de 10, a complexidade assintótica da função é $O(1)$.
- *receiveBets():* Percorre um laço que realiza n iterações, com tempo $O(n)$. Além disso, realiza operações constantes com tempo $O(1)$, e chama as funções *decreaseMoney* e *addTotalAmmount*, ambas com tempo $O(1)$. $O(n) + O(1) + 2O(1) = O(\max(n, 1)) = O(n)$. Como o número de jogadores não pode passar de 10, a complexidade assintótica da função é $O(1)$.
- *getNumberOfPlayers():* Realiza uma operação constante com tempo $O(1)$.
- *getMinimumBet():* Realiza uma operação constante com tempo $O(1)$.
- *getTotalAmmount():* Realiza uma operação constante com tempo $O(1)$.
- *addTotalAmmount():* Realiza uma operação constante com tempo $O(1)$.
- *bubbleSortPlayers():* A função *bubbleSort* é chamada com tempo $O(n^2)$. Como o número de jogadores não pode passar de 10, a complexidade assintótica da função é $O(1)$.
- writeWinners(): Chama as funções *getSize*, *getTotalAmmount*, *getMoveString* e *getName*, todas apresentam complexidade de tempo $O(1)$. Além disso, a função *getPlayer* também é chamada, porém, como a lista de vencedores só pode apresentar no máximo 4 jogadores, a função apresenta pior caso $4O(1)$. Por fim, um laço também é utilizado, mas assim como anteriormente, como a lista de ganhadores só pode ter 4 jogadores, o seu pior caso é $4O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + 4O(1) + 4O(1) = O(1)$
- *auxBreakTheTie():* Função recursiva que realiza operações constantes com tempo $O(1)$. Além disso, essa função só pode realizar no máximo cinco chamadas recursivas, com tempo de $5O(1)$. Dessa forma, teremos a complexidade assintótica da função como $O(1)$
- *breakTheTie():* À primeira vista, imaginamos que a função apresenta pior caso e melhor caso. Nesse sentido, o melhor caso ocorre quando a jogada dos jogadores empatados é um **RSF**, assim, não é possível desempatar e a função retorna o valor 0, com tempo $O(1)$. Porém, o pior caso ocorre quando a jogada dos jogadores empatados é um **OP**, assim, é preciso percorrer um laço que é iterado 5 vezes, com tempo $5O(1)$, caso o empate continue mesmo comparando os dois pares de cartas, é preciso chamar a função auxBreakTheTie, com tempo $O(1)$. Dessa forma, encontramos que tanto o melhor quanto o pior caso são $O(1)$. Assim, teremos a complexidade assintótica da função como $O(1)$
- *getWinner():* Realiza operações constantes com tempo $O(1)$. Além disso, utiliza dois laços em sequência, cada um com tempo $O(n)$. Dentro destes laços são chamadas as funções setMove, getMove e insertAtEnd, todas com tempo $O(1)$. Por fim, é utilizado um laço que pode ser iterado no máximo quatro vezes, dessa forma, o laço apresenta tempo no pior caso $4O(1)$. $O(1) + 2O(n) + 4O(1) = O(\max(1, n)) = O(n)$. Como o número de jogadores não pode passar de 10, a complexidade assintótica da função é $O(1)$.

MAIN

Agora que já analisamos todas as funções de todas as classes, torna-se mais simples o processo de análise da complexidade assintótica da função *main*. Durante toda a execução do programa, a função *main* realiza operações constantes com tempo $O(1)$, como a abertura de arquivos, criação e atribuição de variáveis. Dessa forma, podemos seguir para o primeiro laço do programa que é iterado n vezes, em que n representa o número de rodadas que serão jogadas, o chamaremos de "laço rodadas".

Dentro do "laço rodadas", ocorre a execução de uma rodada. Nesse sentido, primeiramente é preciso chamar a função *Round* que criará uma rodada com tempo $O(1)$. Em seguida, é realizado outro laço que é executado m vezes, em que m é a quantidade de jogadores da rodada, o chamaremos de "laço jogadores". Nesse laço, o construtor da classe Player é chamado e em seguida a função *createRoundPlayer*.

Na função *createRoundPlayer*, a função *search* é chamada para verificar se um

jogador já foi adicionado à lista de todos os jogadores, em que apresenta tempo $O(1)$. Se o jogador já foi adicionado, a função tem tempo $O(1)$, caso contrário, a função também apresenta tempo $O(1)$. Por fim, a função *createRoundPlayer* apresenta tempo $O(1)$.

Depois da função *createRoundPlayer*, um laço é executado apenas cinco vezes, tendo complexidade $5O(1)$. Portanto, o "loop jogadores" que é executado m vezes, em que m é o número de jogadores, apresenta seguinte complexidade assintótica, já que o número de jogadores não pode ser maior que 10:

$$O(1) + O(1) + O(1) + 5O(1) = O(1)$$

Continuando a execução do "laço rodadas", em que n é o número de rodadas, ocorre a execução da função *executeRound*. Essa função realiza operações constantes com tempo $O(1)$, como *getMinimumBet* e *addTotalAmmount*. Além disso, ocorre um laço que é executado n vezes, com operações constantes, $O(n)$. A função *receiveBets* tem complexidade $O(n)$ e a função *getWinner* com tempo $O(n)$ também. O número n representa a quantidade de jogadores. Por fim, como o número de jogadores não pode ser maior que 10, a função executeRound apresenta a seguinte complexidade assintótica:

$$O(1) + O(1) + O(1) + O(1) = O(1)$$

Por fim, podemos terminar de analisar a complexidade do "laço rodadas" como um todo, que é executado n vezes em que n representa a quantidade de rodadas:

$$O(n)$$

Portanto, nosso programa como um todo apresenta complexidade assintótica $O(n)$, apresentando uma complexidade linear em que n representa o número de rodadas. Isso ocorre já que o número de cartas de cada jogador é fixo em 5, além de que o número de jogadores não pode ser maior que 10.

3.1 Análise de Complexidade de Espaço

Para realizar a análise de complexidade de espaço, precisamos analisar a função *main* como um todo, já que ela lida com a execução do programa em si como um todo. Nesse sentido, a função *main* apresenta três principais partes, em que no programa, são representados em laços no código. Para realizar esta análise, precisamos analisar os laços de dentro para forma, de forma indutiva, partindo de uma premissa menor para uma premissa mais geral.

Primeiramente, o laço mais interior do programa é o "laço cartas". Esse loop é executado exatamente cinco vezes, já que cada jogador apresenta exatamente cinco cartas sempre. Dessa forma, a complexidade de espaço do "laço cartas" pode ser definida da seguinte forma:

$$5O(1) = O(1)$$

Em segundo lugar, o próximo laço a ser analisado é o "laço jogadores". Esse loop lida com a criação dos jogadores da rodada e a criação das cartas, que já foi analisada anteriormente. Esse loop é executado n vezes, em que n é o número de jogadores da rodada do jogo de pôquer. Porém, como a quantidade de jogadores não pode ser maior que dez, a complexidade de espaço do "laço jogadores" pode ser definida assim?

$$O(1) * O(1) = O(1)$$

Por fim, o último laço a ser analisado é o "laço rodadas". Esse loop lida com a criação e execução das rodadas, além de criar os jogadores e suas respectivas cartas. Nesse sentido, esse loop é executado n vezes, em que n é o número de rodadas que serão executadas no jogo de pôquer. Além disso, os dois laços analisados anteriormente são executados de forma aninhada. Dessa forma, a complexidade de espaço do "laço rodadas" pode ser definida da seguinte forma:

$$O(n) * O(1) = O(n)$$

Nesse sentido, a função *main*, realiza a chamada de todos esses laços de forma aninhada, $O(n)$. Além disso, realiza também suas operações considerando estruturas auxiliares unitárias $O(1)$. Portanto, a complexidade de espaço final da função *main* pode ser definida da seguinte forma:

$$O(n) + O(1) = O(\max(1, n)) = O(n)$$

4 Estratégias de Robustez

Robustez é a capacidade de um sistema funcionar mesmo em condições anormais. Nesse sentido, as funções implementadas no programa criado, se necessário, apresentam verificações de entradas inadequadas e mal funcionamento do programa. Para tornar o código mais robusto, foram utilizadas funções da biblioteca *msgassert.h*, disponibilizada no moodle da matéria estruturada de dados.

Primeiramente, analisaremos a robustez das funções das duas estruturas de dados implementadas, lista encadeada e vetor. Em seguida analisaremos as classes implementadas e, por fim, a robustez da função *main*.

4.1 List

- *setPosition(): Verifica se a posição passada como parâmetro é maior que 0 e menor que o tamanho da lista. Caso a verificação seja falsa, a execução do programa é interrompida.*
- *Todas as funções de remoção: Verifica se a lista está vazia. Caso a verificação seja falsa, a execução do programa é interrompida.*

4.2 Vector

- *~Vector():* Verifica se o tamanho do vetor é maior que zero. Caso a verificação seja falsa, é impresso “Vector has already been destroyed” no terminal.
- *Vector():* Verifica se o tamanho do vetor passado como parâmetro é maior que zero. Caso a verificação seja falsa, a execução do programa é interrompida. Além disso, é verificado se o vetor *items* tem o endereço diferente de nulo. Caso contrário, a execução do programa é interrompida.
- *readElement():* Verifica se a posição recebida como parâmetro é maior ou igual a zero e menor que o tamanho do vetor. Caso contrário, a execução do programa é interrompida e é impresso “Invalid index” no terminal.
- *writeElement():* Verifica se o vetor está não está cheio. Caso contrário, a execução do programa é interrompida e é impresso “Full list” no terminal.

4.3 Card

- *Card():* Verifica se o número recebido como parâmetro é maior ou igual a 1 e menor ou igual a 13, o naipe é colocado em caixa alta e verifica se o naipe é igual a C, E, P ou O. Caso contrário, a execução é interrompida.
- *~Card():* Verifica se a chave é diferente de -1. Caso contrário, é impresso “Card has already been destroyed”.
- *setNumber():* Verificação semelhante ao construtor.
- *setNaipe():* Verificação semelhante ao construtor.

4.4 Player

- *Player():* Verifica se o nome passado como parâmetro não está vazio, se a aposta é múltipla de 50 e positiva e se o dinheiro do jogador é maior que zero. Caso contrário, a execução é interrompida.
- *~Player():* Chama a função *avisoAssert*, verificando se a chave é diferente de -1. Caso contrário, é impresso “Player has already been excluded” no terminal.
- *setName():* Verificação semelhante ao construtor.
- *setMoney():* Verificação semelhante ao construtor.
- *decreaseMoney():* Verifica se o dinheiro final do jogador menos o valor recebido como parâmetro é maior ou igual a zero. Caso contrário, a execução do programa é interrompida.
- *setBet():* Verifica se a aposta do jogador é maior ou igual a zero e se é múltipla de 50. Caso contrário, a execução do programa é interrompida.

- *insertCardOnHand():* Robustez dessa função é tratada pela função *writeElement*.
- *getCardIndex():* Caso a carta não seja encontrada no vetor, é chamada a função *avisoAssert* que imprime “Card not found!” no terminal.

4.5 Round

- *Round():* Chama a função *erroAssert*, verificando se a quantidade de jogadores é maior ou igual a 1, caso contrário, a execução é interrompida e é impresso “Number of players must be greater than zero” no terminal. Chama a função *erroAssert*, verificando se a aposta mínima é maior que zero e múltipla de 50, caso contrário, a execução é interrompida e é impresso “Invalide minimum bet. Minimum bet must be biggger than 0 and multiple of 50” no terminal.
- *writePlayers():* Verifica se o arquivo de output está aberto, caso contrário, a execução é interrompida e é impresso “Output file is not open” no terminal.
- *receiveBets():* Chama a função *erroAssert*, verificando se a aposta de cada jogador é maior que a aposta mínima da rodada, caso contrário, a execução é interrompida e é impresso “Player bet smallest than minimum bet” no terminal.
- *breakTheTie():* Chama a função *erroAssert*, verificando se a jogada do jogador é maior ou igual a 0 e menor ou igual a 9, caso contrário, a execução é interrompida e é impresso “Move value invalid” no terminal.
- *distributeAmmount():* Chama a função *erroAssert*, verificando se o tamanho da lista de vencedores é maior do que zero, caso contrário, a execução é interrompida e é impresso “Empty winner list” no terminal.
- *writeWinners():* Chama a função *erroAssert*, verificando se o tamanho da lista de vencedores é maior do que zero, caso contrário, a execução é interrompida.

5 Análise Experimental

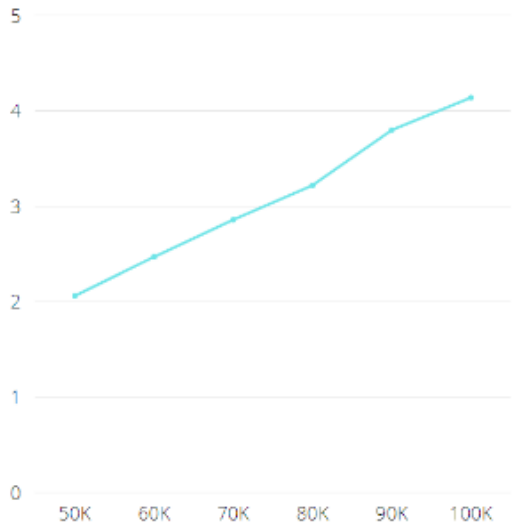
Essa seção apresenta os experimentos realizados em termos de desempenho computacional e localidade de referência. Os resultados desses experimentos foram gerados utilizando as bibliotecas disponibilizadas no Moodle da disciplina de estrutura de dados *analisamem* e *memlog*. Por fim, a seção também apresenta uma análise desses resultados, para um melhor entendimento sobre a execução do programa.

5.1 Análise de Desempenho

Primeiramente, realizaremos a análise de desempenho. Para isso, foram utilizados arquivos de entrada gerados pelo programa *geracarga.c*, também disponibilizado no Moodle da disciplina de estrutura de dados. Foram utilizadas entradas variando o número de rodadas, já que o número máximo de jogadores é 10 e não é possível realizar alterações no número de cartas por jogador, que deve ser sempre 5. O número de rodadas varia de 50 mil para 100 mil, variando de 10 mil em 10 mil. A seguir temos a tabela comparando o tempo de execução de cada partida, além de um gráfico ilustrativo do crescimento do tempo de execução em relação ao crescimento do número de rodadas e jogadores.

NÚMERO DE RODADAS	RESULTADO DE DESEMPENHO
50K	I 1 873.632668500 F 2 875.701755600 R 2.069087100
60K	I 1 875.704785700 F 2 878.176248800 R 2.471463100
70K	I 1 878.179776800 F 2 881.041076100 R 2.861299300
80K	I 1 881.045258400 F 2 884.274285100 R 3.229026700
90K	I 1 884.278614000 F 2 888.080741400 R 3.802127400
100K	I 1 888.085402400 F 2 892.234940000 R 4.149537600

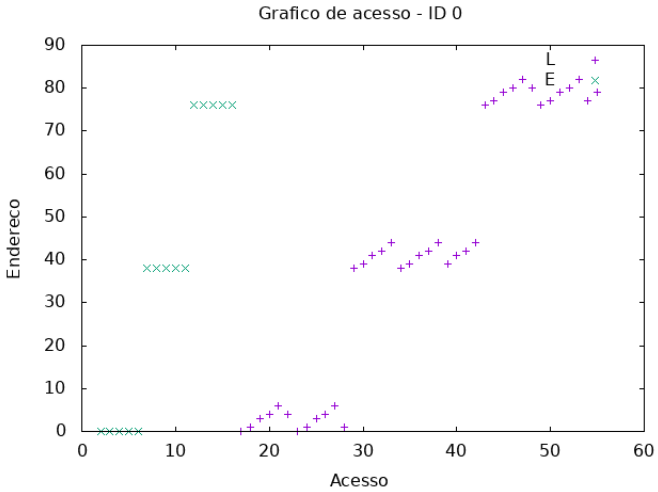
Analisando o gráfico, percebemos que ele apresenta um crescimento semelhante à um crescimento linear, tal qual já foi encontrado na seção 3, durante a análise assintótica do programa, $O(n)$. Dessa forma, confirmamos a complexidade de tempo do programa como linear, em relação ao número de rodadas.



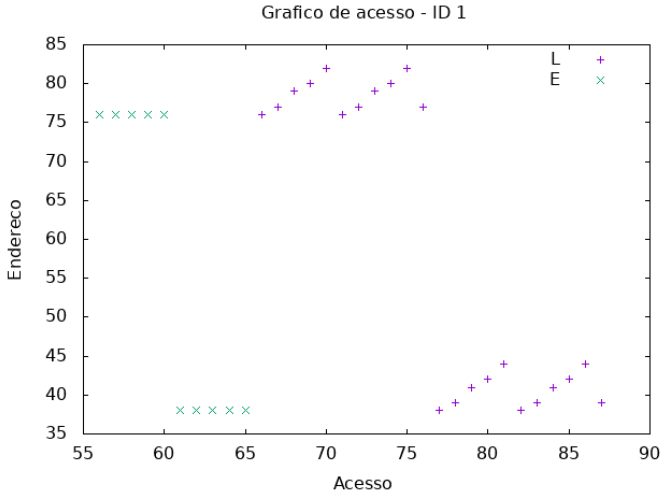
5.1 Localidade de Referência

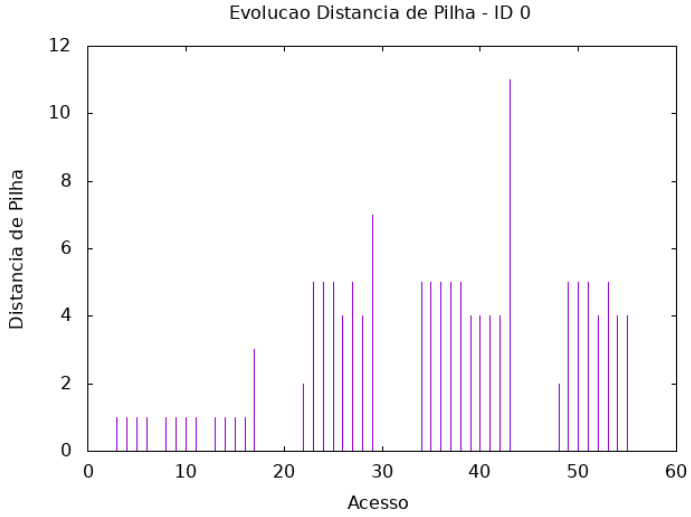
Em seguida, para realizar a análise de localidade de referência, analisaremos especificamente as cartas de cada jogador, como cada jogador possui 5 cartas, será fácil localizar a localização do endereço de cada jogador na memória. Nesse sentido, o arquivo de entrada utilizado para realizar essa análise possui duas rodadas, em que na primeira rodada jogam três jogadores, enquanto na segunda jogam apenas dois. Nos gráficos abaixo, a primeira rodada é identificada com a chave 0, enquanto a segunda rodada é identificada com chave 1.

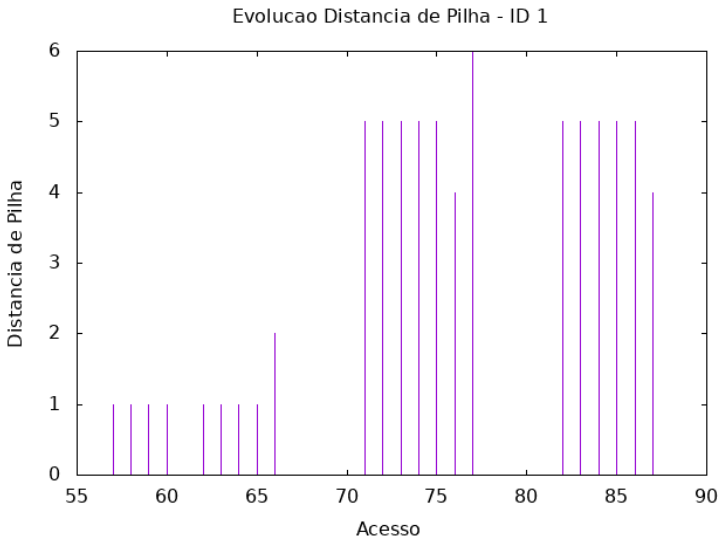
Os gráficos de acesso a seguir, tratam da execução da primeira e da segunda rodada. Primeiramente, as cartas dos três jogadores são inicializadas em endereços de memória próximos, o que não ocorre em relação ao endereço dos jogadores, que se encontram mais distantes.



Em seguida, o processo de definição de valor de jogada começa, em que é preciso realizar três iterações sobre todas as cartas da mão dos jogadores. Dessa forma, todos os jogadores apresentam suas mãos de cartas lidas três vezes.







Em seguida, os dois gráficos acima os dois gráficos acima mostram a evolução da distância de pilha com o decorrer da execução do programa. Primeiramente, enquanto as cartas estão sendo escritas, por serem endereços novos, elas sempre apresentam a mesma distância de pilha. Porém, durante o processo de escolha de valor de jogada, um jogador escolhe a carta de cada vez, dessa forma, é preciso percorrer toda a pilha que apresenta a carta de outros jogadores para chegar até a sua carta.

Uma possível alternativa para diminuir a distância de pilha, seria ordenar os jogadores ao contrário para realizar a escolha de sua jogada, assim, o último jogador a ter suas cartas escritas, seria o primeiro a tê-las lidas durante o processo de escolha de jogada, diminuindo a distância de pilha. Outra forma, seria realizar a análise de jogada de todos os jogadores juntos, dessa forma, sempre haveria a mesma distância de pilha estre as leituras de cartas.

6 Conclusão

Este trabalho lidou com o problema de implementação de um jogo de pôquer, utilizando as estruturas de dados aprendidas ao decorrer do curso de estrutura de dados. Nesse sentido, a abordagem utilizada foi a criação de um programa na linguagem C, que utiliza duas estruturas de dados, sendo elas um vetor e uma lista encadeada, além da implementação de classes para implementar o jogo em si como um todo, que são a classe de cartas, jogadores e rodadas.

Com a solução adotada, pode-se verificar que é possível realizar partidas de pôquer com um número arbitrariamente grande de rodadas e jogadores, como foi exemplificado durante os testes de performance (seção 5), em que foram utilizados TODO jogadores e TODO rodadas. Essa solução permitiu realizar uma análise sucinta de como a execução do jogo ocorre em relação à memória, tanto em relação ao tempo como em relação ao espaço, observando os gráficos e resultados de desempenho gerados pela execução do Makefile juntamente com as bibliotecas *analisamem* e *gprof*.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados às estruturas de dados trabalhadas durante o curso, além da implementação do recurso *typename* na implementação da classe vetor, que permitiu praticar os conceitos de polimorfismo paramétrico. Além disso, durante a implementação da solução para o problema, um dos grandes desafios foi realiza a análise de complexidade espacial e temporal e análise experimental, que ainda não possuo muito experiência, mas o Trabalho Prático 1 ajudou muito no melhor entendimento destes conceitos. Além disso, a realização do trabalho também sumarizou no melhor entendimento da utilização de *friend classes,* que eu particularmente não havia tido contato.

7 Bibliografia

Slides virtuais e códigos da disciplina de estruturas de dados disponibilizados via moodle.

Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

8 Instruções para Compilação e Execução

- Primeiramente, utilizando um terminal, acesse o diretório TP.
- Ao digitar o comando ***make all*****,** o comando *build* do arquivo Makefile é executado no terminal, e, dessa forma, o executável do programa será criado com nome 'tp1.exe' na pasta *bin*.
- Além disso, outros comandos também podem ser executados:
    - ***perf:*** Lê os arquivos "entrada50k.txt" até "entrada100k.txt", executando o "jogo de pôquer" de cada um, variando de 50 mil até 100 mil rodadas. Esses arquivos são retirados da pasta *assets*. Além disso, os resultados de desempenho são armazenados, retornando o tempo de execução de cada jogo. Os resultados de desempenho são armazenados na pasta *tmp*, para ver os resultados, basta acessar a pasta *tmp* no terminal e escrever comandos no seguinte formato: 'more entrada50k.out'.
    - ***mem:*** Executa o arquivo "entrada.txt" localizado na raiz do diretório, de preferência um arquivo com poucos jogadores e um menor número de rodadas. Além disso, os acessos a memória são registrados e podem ser utilizados para a plotagem de gráficos utilizando a biblioteca disponibilizada *analisamem*, juntamente com o *gnuplot*.
    - ***run:*** Executa o arquivo 'bin/tp1.exe', dessa forma, o "jogo de pôquer" contido no arquivo "entrada.txt", que deve estar no diretório raiz, será executado. Além disso, o registro de memória do programa será armazenado no arquivo */tmp/roundlog.out*.
- Caso o usuário prefira, é possível executar um arquivo específico de jogo de pôquer. Para isso, estando acessando o diretório raiz do programa, primeiramente é necessário executar o comando 'make build'. Em seguida, é preciso executar uma linha semelhante a seguinte, substituindo o nome do arquivo txt pelo arquivo desejado. Além disso, é preciso inserir o destino do arquivo de acesso de memória, após o '-p'.
    - 'bin/tp1.exe -1 entrada.txt -p /tmp/roundlog.out'
- Em caso de dúvidas, é possível executar o comando 'bin/tp1.exe -h' para visualizar a mensagem de uso do programa.