Trabalho Prático **1**

Documentação

Estrutura de Dados 2022/01

Bernardo Reis de Almeida (2021032234)

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais
Belo Horizonte - MG - Brasil
bera01@ufmg.br

1. Introdução

O pôquer é um jogo de cartas universalmente apreciado e, mesmo datando de origens que remontam a séculos no passado, detém profunda fama como jogo de azar no mundo contemporâneo. Dadas suas peculiaridades, o presente trabalho prático tem por objetivo explorar a operacionalização deste jogo sob um padecer algorítmico, em que o foco é dado sobre as etapas de **decisão** e de **movimentação de capital**.

Em particular, uma entrada de dados contendo informações sobre uma **partida** (número de rounds e dinheiro inicial dos jogadores), sobre seus **rounds** (número de participantes e aposta inicial) e sobre seus **jogadores** (aposta e mão de cartas) é fornecida e, após um processamento, é gerado como saída um relatório contendo o ganhador de cada round, o montante de dinheiro envolvido, a mão de cartas vencedora, assim como um sumário geral da partida, contendo o nome e a posse de capital de cada jogador envolvido.

A problemática foi abordada pela orientação a objetos, em que, para cada elemento relevante - sejam eles uma **partida**, um **jogador** e uma **carta** -, foi elaborado um **tipo abstrato de dados** que agrega suas principais características e operações, conforme a modelagem elencada. Ainda, algumas estruturas de dados auxiliares - vulgo **listas** e **vetores** - foram adotadas para a organização de elementos anexos, mas de igual relevância - como o **conjunto de jogadores** e a **mão de cartas**. A relação entre esses vários elementos compõe um jogo de pôquer, assim como definido previamente.

Em um primeiro momento, serão tratados os aspectos de abstração, modelagem e implementação dos diversos componentes envolvidos no problema - vide seção **[Método]**. Em seguida, um foco será dado sobre a complexidade dos algoritmos implementados - vide seção **[Análise de Complexidade]** -, assim como sobre as diversas estratégias adotadas para garantir sua robustez e integridade - vide seção **[Estratégias de Robustez]**. Haverá, ainda, um estudo acerca de seu desempenho computacional e de sua localidade de referência - vide seção **[Análise Experimental]**. Por fim, um sumário sobre o processo de desenvolvimento do trabalho, a bibliografia das fontes de consulta utilizadas e as instruções para a execução e para a compilação do programa poderão ser encontradas ao final deste documento - vide, respectivamente, seções **[Conclusões]**, **[Referências Bibliográficas]**, **[Instruções para compilação e execução]**.

2. Método

a. Ambiente Computacional

O programa foi desenvolvido em **linguagem C++** (versão 11), em um ambiente computacional **Linux** virtualmente simulado no sistema operacional Windows (Debian GNU/Linux 11 (bullseye) on Windows 10 x86_64) (5.10.16.3-microsoft-standard-WSL2). O compilador utilizado foi o **G++** (versão (Debian 10.2.1-6) 10.2.1 20210110). Os testes foram executados em uma máquina com um processador de 3.60 GHz (6 núcleos, 12 threads) e com 16 gigabytes de memória RAM.

b. Implementação

O modelo de implementação adotado é baseado no paradigma da **orientação a objetos**, em que cada elemento relevante da problemática foi representado como um **tipo abstrato de dados** - mediante uma classe em C++ -, dotado de **características** - correspondente a seus atributos -, **operações** e **relações** - referente a seus métodos. Em especial, foram elencadas cinco classes no total, separadas em três arquivos de cabeçalho. No arquivo **match.hpp**, está definida a classe **Match**, sendo ela uma abstração de uma partida de pôquer. No arquivo **player.hpp**, estão definidas as classes responsáveis pela modelagem dos jogadores, a saber, **Player**, **PlayerNode** e **ListOfPlayers**. Por fim, no arquivo **card.hpp**, há a definição das classes que representam as cartas: **Card** e **CardHand**.

- **Classe Card:**

O elemento mais primitivo do pôquer, seja ele uma carta, foi representado mediante a classe **Card**, a qual contém dois atributos: **value**, que armazena um valor inteiro referente ao peso da carta (número no intervalo [1, 13], sendo 1 equivalente à carta ás e 11, 12, 13 às cartas Valete, Dama, Rei, respectivamente), e **suit**, o qual armazena um caractere indicador de seu naipe ("P" - "paus", "E" - "espadas", "C" - "copas", "O" - "ouros").

O construtor, por padrão, instancia a carta com valores nulos para ambos os seus atributos, indicando um "objeto vazio" ou "ainda não propriamente definido", mas é possível passar valores individuais como parâmetros para uma inicialização efetiva. Além disso, para cada atributo, existem funções **set()** e **get()**, responsáveis pela manipulação de seu conteúdo. Por fim, tendo em vista as regras competitivas do pôquer, foram definidos "overloads" para os operadores **>**, **<**, **==**, de modo que seja possível comparar duas cartas quanto a uma escala de grandeza (no caso, uma carta "maior" foi definida como aquela que possua o maior peso, independentemente do naipe).

- **Classe CardHand:**

No pôquer, cada jogador recebe, por round, um conjunto de 5 cartas, o qual constitui sua "mão" e o critério definitivo para a condição de vitória na partida. Cada "mão" pode ser classificada em dez diferentes "categorias", sendo cada uma dessas dotada de um "valor" dentro da lógica do jogo.

A classe **CardHand** modela uma mão de cartas por três atributos: um vetor do tipo Card de tamanho 5 - vulgo **hand** -, uma variável do tipo string e uma variável inteira - a saber, **handCategory** e **weight**, respectivamente -, as quais armazenam, em ordem, as cartas que constituem a mão, a sigla e o peso da categoria correspondente. Além disso, a classe ainda contém uma série de outros atributos inteiros que atuam como "características" (eg.: o valor do maior par ou da maior trinca, dentre outras).

Um construtor faz a inicialização de um objeto da classe assinalando aos seus atributos simples um valor nulo. No caso do vetor do tipo Card, cada carta é inicializada com valores também nulos, indicando, novamente, uma instância "inválida" ou "ainda não devidamente inicializada".

O acesso aos atributos é feito por funções **get()**, no caso de handCategory e de weight, e pelo "overload" do operador **[]**, o qual retorna uma referência para a carta correspondentemente indexada em hand.

Por fim, a classe ainda contém dois outros métodos, **orderCardHand()** e **identifyCardHand()**. O primeiro, baseado no algoritmo BubbleSort e nos critérios de comparação estabelecidos na classe Card, ordena o vetor de cartas. O segundo, pautado nas regras do pôquer para a classificação de jogadas, inicializa as variáveis handCategory e weight com base na variável hand, assim como atribui valores adequados aos demais atributos "característicos" da classe.

- **Classe Player:**

No mesmo patamar de abstração da mão de cartas, mas sendo um elemento tão fundamental como a carta, o jogador foi modelado por meio da classe **Player**, a qual o caracteriza por três atributos: **name** - variável do tipo string referente a seu nome -, **amount** - variável do tipo double que armazena a quantidade possuída de dinheiro - e **hand** - sua mão de cartas, uma instância da classe CardHand.

Inicialmente, há um construtor que faz a inicialização de um objeto da classe a partir de seu nome e de seu montante monetário, ambos passados como parâmetros. No caso do atributo hand, sua inicialização invoca o construtor da classe CardHand, o qual o instancia apropriadamente..

Todos os atributos podem ser acessados para leitura mediante funções **get()**. A escrita, porém, só é disponível para o atributo amount, mediante uma função **set()**, e para o atributo hand, por meio de sua própria função get(), a qual retorna uma referência. A parte disso, existe um conjunto de funções set()/get() - **setCard()**, **getCardValue()** e **getCardSuit()** - dedicados ao acesso direto às cartas que compõem a mão do jogador. Dados como parâmetros um índice e valores para o número e para o naipe (quando aplicáveis), elas permitem acessar e manipular o conteúdo da carta correspondente no atributo hand.

Além disso, há dois métodos: **orderHand()** e **identifyHand()**, que invocam as correspondentes funções de ordenação e de identificação sobre o atributo hand, do tipo CardHand.

Por fim, contemplando uma possibilidade de ordenação, há um "overload" do operador **>**, o qual leva em conta, primeiramente, o montante monetário e, posteriormente, a ordem lexicográfica do nome para a determinação da escala de grandeza dos jogadores.

- **Classe ListOfPlayers e PlayerNode:**

Em essência, uma partida de pôquer nada mais é do que um conjunto de jogadores, cada qual dotado de atributos relativos ao jogo, sobre os quais, a cada round, são feitas operações e movimentações. Nessa abordagem, a classe **ListOfPlayers** atua justamente como essa base para o programa, em que a estrutura de dados **lista duplamente encadeada** é utilizada para o armazenamento e para a organização dos participantes e de suas informações. A escolha da lista foi feita com pauta nas ideias de que jogadores podem entrar e sair de uma partida - tamanho variável, inserção/remoção em qualquer ponto - e de que há uma organização interna dos participantes, a critério de alguma das variáveis do jogo - ordenação. O duplo encadeamento, por sua vez, foi preferido em função da maior facilidade em se percorrer a lista, em se inserir/remover elementos em extremidades, além de que, em geral, partidas de pôquer comportam apenas de 3 a 5 jogadores, de modo que a memória adicional necessária para o armazenamento de ponteiros seja compensada por suas conveniências.

A classe **PlayerNode** atua como uma célula da lista e contém três atributos: **player** - instância da classe Player -, **next** e **before** - ponteiros do tipo PlayerNode para a próxima célula e para a anterior, respectivamente. Um construtor recebe como parâmetros um nome e um montante para a instanciação do atributo player e inicializa ambos os ponteiros com o valor nullptr.

A classe ListOfPlayers, por sua vez, detém três atributos: **size** - variável inteira que armazena o seu tamanho (número de células) -, **first** - ponteiro do tipo PlayerNode que aponta para a primeira célula -, e **last** - ponteiro que aponta para a última célula, também do tipo PlayerNode. Há um método **get()** para o atributo size e para o atributo player, sendo que este último recebe um nome para a identificação da célula desejada e retorna um ponteiro para o objeto do tipo Player nela contido.

Um construtor inicializa um objeto da classe atribuindo o valor 0 ao seu tamanho e atribuindo valores nulos a ambos os seus ponteiros. A partir daí, a população da lista é feita com base em dois métodos: **insert()** e **remove()**. O primeiro recebe como parâmetros um nome e um montante e insere uma célula - cujo atributo

player é devidamente inicializado com tais valores - em sua última posição. O segundo recebe como parâmetro um nome e remove a célula correspondente, caso exista.

O método **order()**, ordena todas as células da lista com base nos critérios de comparação estabelecidos para a classe Player. Essa ordenação, novamente, é feita com base no algoritmo Bubble Sort.

A classe também conta com métodos para o acesso e para a manipulação das informações de suas células. Um primeiro, **playerExists()**, recebe um nome como parâmetro e retorna verdadeiro caso um jogador com tal nome exista na lista, retornando falso, caso contrário. Um segundo, **verifyAmount()**, também recebe um nome e retorna o montante monetário da respectiva célula, caso ela exista. Um terceiro, **updateAmount()**, recebe um nome e um valor real e o desconta do montante do jogador correspondente, se ele existir. Um quarto, **updateHand()**, recebe, nova e finalmente, um nome, além de uma stream contendo dados referentes a uma mão de cartas, realizando a devida atualização dessa informação no atributo hand, pertencente ao objeto player, contido na célula da lista, caso ela esteja na lista.

Além disso, também foram definidos métodos que tratam do processamento de informações e da geração de resultados relativos a uma partida. Um desses métodos, **applyInitialBet()**, recebe como parâmetro um valor real com respeito à aposta inicial definida para determinado round e o deduz de todos os jogadores contidos na lista. Um outro método, **verifyWinner()**, recebe como parâmetros vetores contendo o nome dos jogadores participantes em um round e suas respectivas apostas e compara as mãos de cada um, determinando o(s) vencedor(es), realizando a distribuição conforme do montante acumulado e imprimindo tal resultado em um arquivo de saída, também passado como parâmetro. Um último método, **sanityTest()**, recebe um nome e um valor real referente à aposta do respectivo jogador e verifica se, caso ele exista na lista, seu montante monetário é suficiente para cobrir o valor apostado, retornando uma flag caso contrário.

Ainda, há dois métodos dedicados à exportação das informações contidas na lista a um arquivo de saída - passado como parâmetro. O método **outputWinner()** é um procedimento auxiliar à função verifyWinner() responsável por exportar o resultado de determinado round. Um detalhe a ser notado é que tal método ordena os nomes dos vencedores lexicograficamente, com base, mais uma vez, no algoritmo Bubble Sort. O método **outputList()**, por sua vez, realiza a exportação de um sumário contendo o nome e o respectivo montante monetário de todos os jogadores presentes na lista.

Finalmente, a classe também conta com um destrutor bem definido, o qual itera por todas as suas células e desaloca a memória previamente alocada para o seu armazenamento.

- **Classe Match:**

A partida de pôquer em si é modelada pela classe **Match**, a qual contém três atributos: **listOfPlayers** - instância da classe ListOfPlayers que armazenará as informações dos participantes -, **numOfRounds** - inteiro relativo ao número de rounds - e **initialAmount** - valor real referente ao montante inicial de cada jogador. Um atributo auxiliar, ainda, vulgo **roundCount**, é mantido para o controle do número de rounds já processados.

Um primeiro método, **initializeMatch()**, recebe como parâmetro um arquivo de entrada e faz a leitura das informações primordiais da partida, a saber, o número de rounds e o montante inicial possuído pelos jogadores, armazenando-as nos atributos correspondentes - numOfRounds e initialAmount, respectivamente.

A operacionalização dos rounds é realizada pelo método **executeMatch()**. Dados arquivos de entrada e de saída como parâmetros, ele irá iterar um número de vezes compatíveis com roundCount, realizando a leitura dos dados, seu processamento e a geração dos resultados correspondentes.

A cada round, em um primeiro momento, é inicializado um acervo de variáveis auxiliares que armazenarão suas respectivas informações. Em particular, há uma variável inteira para o número de participantes, um vetor de strings para seus respectivos nomes, um vetor de reais para suas respectivas

apostas, uma variável real para a aposta inicial e uma variável inteira que atua como uma flag de validez. Em seguida, o método auxiliar **readRoundInfo()** recebe o arquivo de entrada como parâmetro e faz a leitura da quantidade de participantes e da aposta inicial, armazenando os valores lidos nas variáveis correspondentes. Na sequência, um outro método auxiliar, **readPlayerInfo()**, também recebe o mesmo arquivo de entrada e realiza a leitura das informações de cada jogador, populando o vetor de nomes e de apostas e atualizando a lista de jogadores. Em especial, essa atualização é realizada pelos métodos **updatePlayerStatus()** e **updatePlayerHand()**. O primeiro recebe um nome e uma aposta como parâmetros e desconta o valor da aposta do respectivo jogador, caso ele exista. O segundo recebe um nome e um stream e atualiza a mão de cartas do jogador - cujos valores estão contidos na stream -, caso ele exista. No caso, readPlayerInfo() invoca, além do método updatePlayerHand(), o método updatePlayerStatus() com o valor da aposta nulo, de modo que, no geral, ele insira um novo jogador na lista caso ele ainda não o esteja e atualize a mão daqueles que já estejam. Ao longo do processamento, testes de validação do round são executados e, caso algum deles falhe, aquele é invalidado e um resultado nulo é exportado ao arquivo de saída. Caso contrário, o método updatePlayerStatus() é invocado para cada nome no vetor de participantes com a respectiva aposta do vetor de apostas para efetivamente atualizar o montante de cada um. O método applyInitialBet() também é invocado sobre a lista de jogadores para o devido desconto da aposta inicial. Após isso, o método verifyWinner() é invocado sobre esse mesmo objeto para a geração do resultado do round. Por fim, o atributo roundCount é incrementado e o processo se repete pelo número de rounds definidos para a partida.

O método **outputResults()**, finalmente, recebe como parâmetro um arquivo de saída e invoca os métodos order() e printList() sobre o atributo ListOfPlayers, de modo a exportar um sumário contendo as informações gerais da partida ordenadas (nome dos jogadores e seu montante total) a esse arquivo.

- **Função main:**

A função **main()**, presente no arquivo **main.cpp**, é responsável apenas por instanciar um objeto do tipo Match - referente à partida que se deseja processar - abrir os arquivos de entrada e saída de dados e executar os respectivos métodos de operacionalização das informações para a geração dos resultados.

3. Análise de Complexidade

A complexidade do programa reside principalmente na operacionalização da lista de jogadores. As classes Card, CardHand e Player contêm apenas métodos de custo constante, os quais não dependem de um valor variável ou da organização da entrada de dados e, portanto, não apresentam um impacto significativo quanto a custos (memória, espaço). Além disso, também é válido mencionar que a análise será feita com base em uma entrada variável de tamanho 'n', a qual corresponderá à quantidade total de jogadores participantes em uma partida, isto é, à quantidade total de células que estarão contidas na lista duplamente encadeada.

a. Algoritmo de ordenação "Bubble Sort"

A maioria das operações de ordenação foi implementada com base em um mesmo algoritmo, o **Bubble Sort**, e, portanto, apresentam uma complexidade equivalente. Em essência, tal abordagem é baseada em um aninhamento de dois laços, para o qual, a cada passo do primeiro laço, o segundo laço realiza uma série de comparações para determinar o maior elemento de um subconjunto e o movimentar até sua última posição, procedimento este realizado para todos os subconjuntos da estrutura inicial.

Em detalhes, o primeiro laço é composto por 'n' passos, para cada qual o segundo executa 'n' - 1 - 'i' operações - sendo 'i' o passo em que o primeiro laço se encontra menos uma unidade. Dessa forma, o total de

operações que será realizado é dado por **(n - 1) + (n - 2) + (n - 3) + ... + (1)**, o que equivale a **((n - 1)*n)/2 = (n^2 - n)/2**, de modo que a classe de complexidade assintótica do algoritmo, em termos de **tempo**, seja **O(n^2)**. Com relação ao **espaço**, além dos 'n' elementos armazenados na estrutura de dados, o procedimento em si cria algumas poucas variáveis auxiliares (iteradores dos laços e armazéns temporários de valores para movimentações), de modo que sua complexidade seja fundamentalmente proporcional ao próprio tamanho do objeto com o qual lida, isto é, seja da classe **O(n)**.

Esse comportamento pode ser observado especialmente no método order(), da classe ListOfPlayers. Os demais métodos que contemplam uma ordenação, a saber, outputWinner(), da classe ListOfPlayers, e orderHand(), da classe CardHand, possuem entradas de dados cujos tamanhos não são significativamente variáveis: enquanto uma mão de cartas é sempre composta por exatamente 5 cartas, uma partida só pode ter de 1 a 4 vencedores, conforme as regras do pôquer. Assim, seus custos não são escaláveis e, portanto, não impactam profundamente a complexidade do programa como um todo.

b. Métodos da classe ListOfPlayers

- **Métodos getPlayer(), remove(), playerExists(), verifyAmount(), updateAmount(), updateHand() e sanityTest():**

Recebem um nome como parâmetro e o comparam com os nomes de todos os jogadores contidos na lista até encontrarem uma correspondência, sobre a qual executam operações constantes. Dessa forma, a depender da organização dos dados, um valor no intervalo [1, n] de comparações pode ser realizado, sendo que apenas uma será feita se o jogador desejado estiver na primeira posição e 'n' serão executadas se ele estiver na última posição. Assim, em termos de **tempo**, esses métodos têm um comportamento assintótico que envolve um **melhor caso**, dado por **O(1)**, e um **pior caso**, dado por **O(n)**. Em termos de **espaço**, todos os métodos mencionados envolvem estruturas auxiliares de custo constante e a própria estrutura de dados sobre a qual operam, de modo que sua complexidade seja dada por **O(n)**.

- **Método insert():**

Apresenta complexidades de **tempo** na classe **O(1)** e de **espaço** na classe **O(n)**: no caso do tempo, um objeto do tipo ListOfPlayers possui um ponteiro para a última posição da lista, de modo que a alocação de uma nova célula possa ser feita diretamente, sem a necessidade por se percorrer todas as outras; no caso do espaço, a operação envolve estruturas auxiliares constantes e a própria estrutura de dados em si.

- **Destrutor e métodos outputList() e applyInitialBet():**

Os três executam operações constantes - a saber, descontar um valor do montante, exportar informações à saída de dados e desalocar a memória alocada para uma célula, respectivamente - sobre todos os jogadores da lista, de modo que sua complexidade **temporal** seja **O(n)**. Em termos de **espaço**, todos envolvem variáveis auxiliares constantes e a própria estrutura de dados em si, de maneira que sua ordem de complexidade seja **O(n)**.

- **Método outputWinner():**

Este método não depende da lista em si. Um vetor contendo o nome dos jogadores vencedores constitui a base de suas operações e possui tamanho, respeitadas as regras do pôquer, que pode variar entre 1 e 4 elementos, o qual, por não ser escalável, pode ser considerado como tendo um custo constante. Dessa forma, tanto em termos de **tempo**, quanto de **espaço**, esse método possui ordem de complexidade **O(1)**, já

que realiza apenas operações de custo "constante" (ou pouco escalável) e envolve apenas algumas variáveis também de custo "constante".

- **Método verifyWinner():**

Este método introduz uma nova variável para a complexidade do algoritmo, dada pela quantidade de jogadores participantes em um determinado round. Em particular, para cada nome no vetor de participantes, é invocado o método getPlayer(), o qual percorrerá a lista à procura de uma correspondência. Em um **melhor caso**, dado por um round contendo apenas um participante cuja posição na lista é a primeira, sua complexidade **temporal** será **O(1)**, pois o método getPlayer() será invocado apenas uma vez e realizará apenas uma comparação. Em um **pior caso**, dado por todos os jogadores participarem do round, sua complexidade **temporal** será **O(n^2)**, já que cada jogador será buscado pelo método getPlayer(), o qual realizará (1) comparação para o primeiro, (2) para o segundo, e assim por diante até o n-ésimo, gerando uma complexidade de busca **(1) + (2) + … + (n) = (n(n + 1))/ 2 = (n^2 + n)/2**. O método ainda faz uma operação de comparação entre as mãos de todos presentes no vetor de participantes, a qual, no melhor caso mencionado, tem complexidade O(1) e, no pior caso, O(n), de modo que sua ordem de complexidade ainda seja dada pela análise anterior. Também é realizada uma chamada ao método outputWinner(), o qual, como já mencionado, foi considerando como possuindo custos "constantes". Em qualquer caso, o método envolve algumas variáveis auxiliares, a estrutura de dados em si e um vetor com os jogadores participantes no round, o qual pode crescer até o tamanho 'n'. Dessa forma, sua ordem de complexidade **espacial** é dada por **O(n)**.

c. Métodos da classe Match

A complexidade dos métodos da classe Match está diretamente relacionada com a invocação dos métodos da classe ListOfPlayers. Aqueles que envolvem operações complexas adicionais serão detalhados, enquanto os demais devem ser assumidos como possuindo custo constante.

- **Método outputResults():**

Realiza uma invocação ao método outputList() e ao método order(), de modo que sua complexidade **temporal** seja da ordem **O(n^2)** e **espacial** seja da ordem **O(n)**.

- **Métodos updatePlayerStatus() e updatePlayerHand():**

Ambos fazem, além de operações com custo constante, chamadas aos métodos playerExists(), de modo que sua complexidade **temporal** seja, no **melhor caso**, da ordem **O(n)** e, no **pior caso**, da ordem **O(1)**. Sua complexidade **espacial**, em função da chamada aos métodos referidos, é da ordem **O(n)**.

- **Método readPlayerInfo():**

Assim como o método verifyWinner(), este também agrega à complexidade o número de jogadores participantes em um round. Em particular, para cada participante, são invocados os métodos updatePlayerStatus() e updatePlayerHand(), de modo que, em um **melhor caso** (um participante, cuja posição na lista é a primeira), sua complexidade **temporal** será **O(1)** (apenas uma comparação é realizada) e, em um **pior caso** ('n' participantes), sua complexidade **temporal** será **O(n^2)** ((1) + (2) + … + (n) comparações). Em termos **espaciais**, ambos estes métodos invocados envolvem a lista de jogadores como objeto de maior relevância, de modo que sua ordem de complexidade seja **O(n)**.

- **Método executeMatch():**

Este método, por sua vez, também introduz um terceiro detalhe particular à complexidade do programa. Suas operações, além de escalarem com o tamanho 'n' da lista de jogadores e com o número de participantes, também escalam com a quantidade de rounds a serem executados em uma determinada partida - seja esse valor dado por 'm'.

Iniciando a análise pela complexidade temporal de apenas um round, dentro deste são evocados os métodos readPlayerInfo(), sanityTest(), applyInitialBet(), updatePlayerStatus() e verifyWinner(), além de serem realizadas operações escaláveis com o número de participantes. Em particular, os métodos updatePlayerStatus() e sanityTest() são invocados em um laço para cada participante, e as operações adicionais mencionadas são uma comparação duplamente aninhada entre todos os nomes contidos no vetor de nomes e uma verificação de condição em laço para todos os valores contidos no vetor de apostas. Define-se, para efeito de análise do round, um melhor caso como aquele em que há apenas um participante que ocupa a primeira posição da lista e um pior caso como aquele em que todos os integrantes da partida participam do round. A complexidade de cada uma dos subprocedimentos pode ser vista a seguir:

- readPlayerInfo(): melhor caso - O(1); pior caso - O(n^2)
- laço + sanityTest(): melhor caso - O(1); pior caso - O(n^2)
- applyInitialBet(): O(n) em ambos os casos
- laço + updatePlayerStatus(): melhor caso - O(1); pior caso - O(n^2)
- verifyWinner(): melhor caso - O(1); pior caso - O(n^2)
- laço duplamente aninhado para o vetor de nomes: melhor caso - O(1); pior caso - O(n^2)
- laço simples para o vetor de apostas: melhor caso - O(1); pior caso - O(n)

Dessa forma, é possível concluir que a complexidade total de um round, em termos de tempo, é da ordem de O(n^2) no pior caso e O(n) no melhor caso.

Agora, no caso do método executeMatch() como um todo, um detalhe a ser notado é que, no primeiro round, fica estabelecido que todos os jogadores integrantes da partida participam, de modo que, para qualquer valor 'm' maior ou igual a 1, o pior caso do round sempre ocorra pelo menos uma vez. Assim, a complexidade geral do método em termos de **tempo** pode ser separada em **dois casos**: aquele em que **nenhum round é executado** e, portanto, a classe assintótica é **O(1)**, e aquele em que **um ou mais rounds ocorrem**. Nesse segundo caso, ainda há uma **melhor situação**, dada por todos os rounds além do primeiro terem complexidade O(n), de modo que a complexidade geral é da ordem **O(n^2 + (m-1)*n)**, e uma pior situação, dada por todos os rounds além do primeiro terem complexidade O(n^2), o que gera uma complexidade geral da ordem **O(m*n^2)**.

Com relação ao **espaço**, além da lista de jogadores de tamanho 'n', o método executeMatch() ainda cria dois vetores auxiliares com tamanhos que podem chegar a 'n' e evoca uma série de métodos com complexidade O(n), o que, independentemente, ainda o classifica na classe **O(n)**, já que c*O(n) = O(n).

d. Função main

O programa como um todo consiste na instanciação de um objeto Match, em uma série de operações constantes e na invocação dos métodos initializeMatch(), outputResults() e executeMatch(). Um detalhe a ser pontuado é que, **se nenhum round for executado**, não há a população da lista de jogadores e, portanto, apenas um conjunto de operações constantes será realizado, originando uma complexidade **temporal** e **espacial** da classe **O(1)**. Caso **haja a realização de algum round**, com base nas análises feitas previamente, sua complexidade **temporal** é dada por um **melhor caso** na classe **O(n^2)** e um **pior caso** na

classe **O(m*n^2)**, considerando todas as sub rotinas executadas sobre o objeto Match, enquanto sua complexidade espacial é da ordem de **O(n)**, dado que a lista de jogadores de tamanho 'n' é seu objeto de maior significância.

4. Estratégias de Robustez

A robustez do programa foi construída com o auxílio de duas bibliotecas externas: **msgassert** e **regex**. A primeira oferece recursos para detecção de erros e para o tratamento correspondente, seja ele pelo interrompimento da execução ou por uma mensagem de aviso. A segunda, por sua vez, oferece funções para a verificação da formatação de entradas de texto, comparando-as com um padrão pré-estabelecido. Um outro detalhe a ser pontuado é que o programa como um todo assume uma série de definições com relação ao jogo de pôquer e, em especial, aos dados que serão fornecidos como entrada. Para casos que não sigam os critérios estabelecidos, não foram elaboradas estratégias específicas de prevenção, de modo que o estado de operação do programa seja inesperado.

A **integridade da manipulação e do armazenamento dos dados**, tanto de entrada, quanto de saída, foi garantida mediante condicionais que verificam o estado do arquivo em uso e das estruturas em memória. A cada abertura, fechamento, escrita e leitura, a condição do arquivo quanto a erros é verificada. Além disso, para cada peça de informação extraída e armazenada com sucesso em alguma variável interna, a adequação desta ao formato de dados pré-definido também é verificada. Como um último detalhe, a movimentação dos dados no programa foi intermediada por streams, cuja condição de integridade também é verificada a cada leitura/escrita. Em todos os casos, a execução do programa é interrompida caso existam anormalidades, dado que as informações são uma peça essencial do funcionamento do algoritmo como um todo.

A **integridade da manipulação de informações pertencentes às classes** também foi elaborada com base em verificações condicionais que avaliam o estado de operação do programa. No caso, tem-se que as definições mencionadas previamente garantem grande parte da corretude, mas ainda assim há margem para possíveis e potencialmente perigosas exceções. Na classe ListOfPlayers, para grande parte das funções que envolvem a busca por um determinado jogador na lista - sejam elas remove(), verifyAmount(), updateAmount() e updateHand() -, caso este não seja encontrado, um aviso é emitido. A escolha dessa abordagem ao invés do interrompimento da execução se deve ao fato de o "não encontrar" um determinado jogador não necessariamente invalidar uma possível operacionalização da lista. Além disso, é provido um método dedicado à verificação da condição de existência de células, permitindo que, se coerente, excepcionalidades sejam tratadas de maneira externa à classe. Nas classes Card e CardHand, por sua vez, há asserções para métodos que dependam das informações contidas em um objeto carta ou mão de cartas - sejam eles o overload dos operadores >, <, ==, orderCardHand(), identifyCardHand() e compare() - que avaliam se eles foram devidamente inicializados, já que ambos podem ser instanciados com valores nulos para seus atributos, sendo esta uma escolha de implementação.

O **funcionamento de uma partida**, por sua vez, também depende em grande parte das definições de uso do programa. Porém, também foi realizado um tratamento de excepcionalidades que podem porventura ocorrer e gerar resultados inesperados/incorretos. Em particular, a função sanityTest(), definida sob a classe ListOfPlayers, garante que, a cada round, um jogador sem dinheiro o suficiente ou um jogador que não pertença àquela partida (os jogadores de uma partida são definidos no primeiro round) não seja permitido de participar, invalidando a rodada caso este evento ocorra. Além disso, como mencionado previamente, há o uso da função playerExists() para garantir a correta manipulação dos jogadores que integram a lista de participantes da partida, evitando acessos, o uso ou a criação de informações para jogadores inexistentes (ou não propriamente inicializados no primeiro round). Por fim, é verificado, a cada round, se há algum jogador que

fez duas apostas (assume-se que cada jogador possuirá um nome único), se o número definido de jogadores participantes é 0 ou se há alguma aposta cujo valor não é um múltiplo de 50, invalidando-se a rodada em qualquer dos casos.

5. Análise Experimental

A análise experimental foi realizada por meio de três ferramentas externas auxiliares: a biblioteca **memlog** e as aplicações **analisamem** e **gprof**. A primeira oferece recursos para o registro - em um arquivo de texto externo - do momento de início e de finalização da execução, assim como dos diversos acessos realizados pelo programa à memória. A segunda permite a transformação dos registros realizados pela anterior em gráficos analíticos. A terceira, por sua vez, realiza o registro do tempo gasto para a execução de cada função individualmente, assim como do tempo total gasto para a execução do programa como um todo.

a. Desempenho Computacional

A análise de desempenho computacional foi realizada para duas configurações distintas da entrada de dados, cada qual pertencente a uma classe assintótica única. Uma observação pertinente é que restrições com relação ao domínio e à cardinalidade do conjunto "naipe" das cartas não foram relevadas (análogo a ter sido utilizado um número infinito de baralhos), a fim de que fosse possível escalar o problema. Além dos resultados práticos de cada teste, é fornecido um resultado "teórico", cujos valores foram obtidos tendo em vista um primeiro resultado e um cálculo estimativo levando em conta a ordem de complexidade do algoritmo. A seguir, tem-se as informações coletadas:

1°) Apenas um round com 'n' participantes é fornecido, de modo que o comportamento assintótico esperado para o algoritmo seja O(n^2). Duas baterias de testes foram realizadas, cada uma escalando o valor inicial da entrada por um valor inteiro constante no intervalo [1, 5] (isto é, uma execução é realizada com tamanho 'n', uma segunda com tamanho 2*'n' e assim por diante até a quinta). Seguem os resultados:

Entrada (n° de jogadores)	**Tempo prático**	**Tempo teórico O(n^2)**
5000	15.929769054	15.929769054
10000	43.103075787	63.719076216
15000	79.890159138	143.367921486
20000	128.143618349	254.876304864
25000	185.797539246	398.24422635
20000	126.424071459	126.424071459
40000	425.223116337	505.696285836
60000	892.088174113	1,137.816643131
80000	1546.804508434	2,022.785143344
100000	2380.412594701	3,160.601786475

2°) Um número 'n' de rounds, cada qual com 'n' participantes, é fornecido, de modo que o comportamento assintótico esperado para o algoritmo seja O(n^3). Novamente, duas baterias de testes foram realizadas da mesma forma que a primeira configuração. Seguem os resultados:

Entrada (n° de jogadores/rounds)	Tempo prático	Tempo teórico O(n^3)
50	5.401911117	5.401911117
100	21.541000545	43.215288936
150	48.428793912	145.851600159
200	84.419917657	345.722311488
250	132.503656169	675.238889625
100	21.439970578	21.439970578
200	88.699368872	171.519764624
300	194.231485637	578.879205606
400	349.728771424	1,372.158116992
500	561.361768303	2,679.99632225

Os dados coletados expõem de maneira clara que os resultados práticos obtidos para ambas as configurações destoam consideravelmente do esperado para a classe assintótica do algoritmo. Enquanto **a razão para isso não foi encontrada**, um detalhe que é possível de ser observado é que, conforme a escala dos valores de entrada aumenta (entre a primeira e a segunda bateria de testes, o valor da entrada inicial é consideravelmente incrementado), os valores práticos se aproximam dos valores teóricos, o que é especialmente evidente na primeira configuração. Esse detalhe, por sua vez, é compatível com a teoria por detrás da classificação assintótica de um algoritmo: conforme o tamanho de sua entrada de dados cresce, seu comportamento se assemelha cada vez mais àquele de sua ordem funcional. Assim, é teorizado que, conforme a entrada aumente, os resultados apresentem a tendência de se aproximar dos valores esperados (isso não foi testado tendo em vista o exagero no tempo de execução do programa).

Mesmo assim, ao se analisar cada função individualmente, é possível ver que há uma certa obediência à complexidade assintótica definida. A seguir, tem-se os resultados obtidos para testes realizados nas duas configurações anteriores, mas para tamanhos distintos de entrada, considerando o tempo de execução da função executeMatch() - esta foi escolhida tendo em vista que é a função mais relevante em termos de custo computacional para a execução do programa. Novamente, as entradas são escaladas por múltiplos inteiros constantes, mas desta vez no intervalo [1, 3], além de que são fornecidos tanto o resultado prático, quanto um resultado "teórico". A seguir, seguem-se as informações coletadas:

Entrada (n° de jogadores)	Tempo prático	Tempo teórico O(n^2)
10000	0.23	0.23
20000	0.98	0.92
30000	2.17	2.07

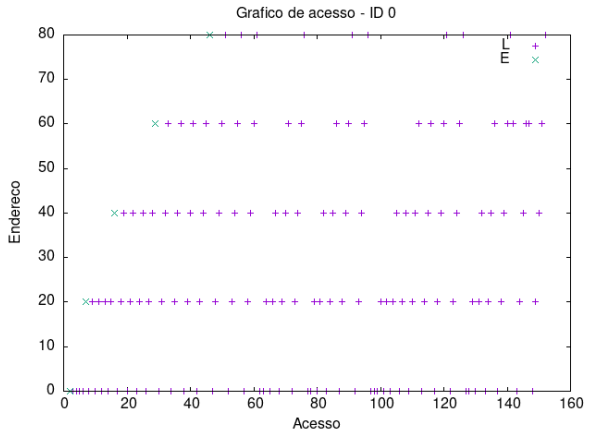
Entrada (n° de jogadores/rounds)	Tempo prático	Tempo teórico O(n^3)
200	0.02	0.02

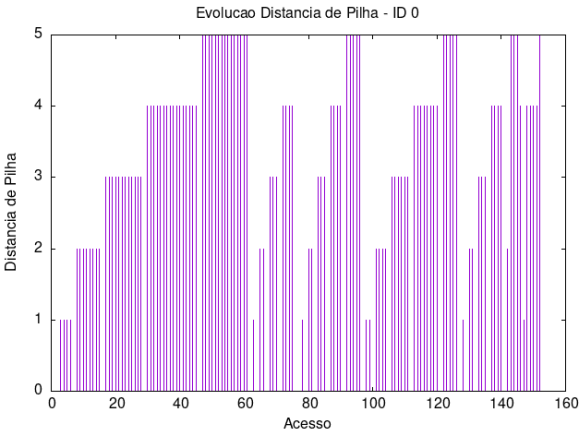
400	0.16	0.16
600	0.54	0.54

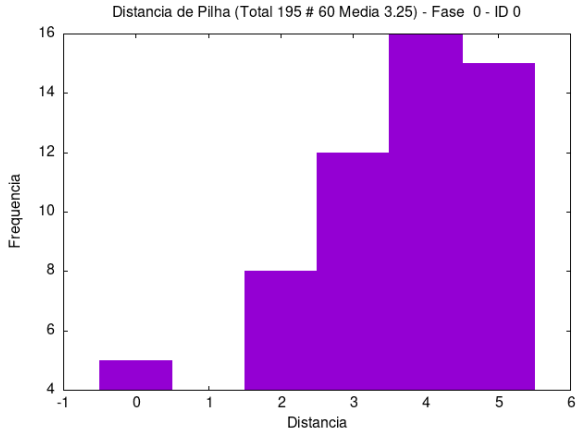
Novamente, **a razão dessa divergência nos tempos de execução do programa com relação a sua ordem de complexidade em contrapartida com a obediência das funções individuais não foi encontrada**.

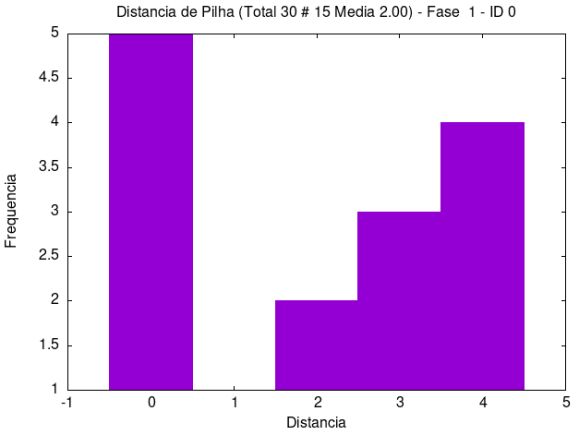
b. Localidade de Referência

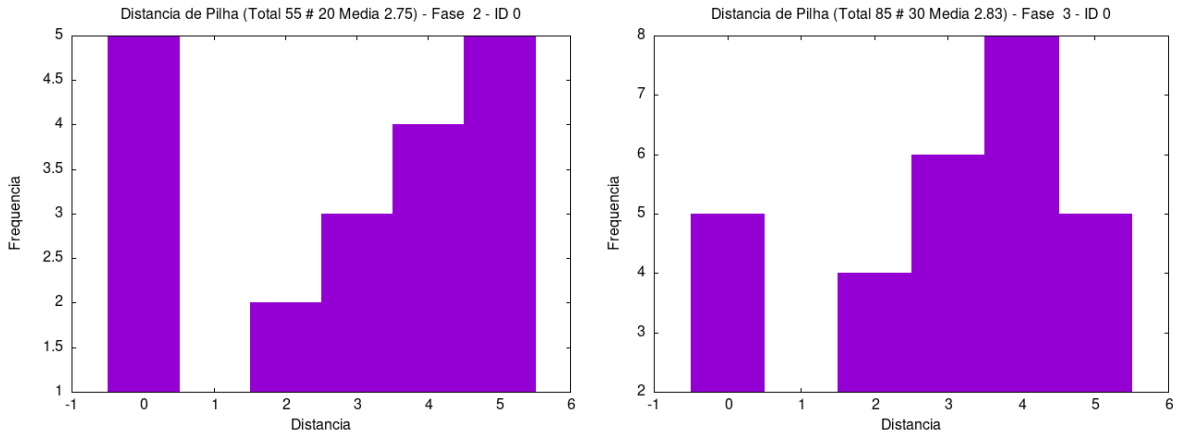
A análise da localidade de referência foi realizada para uma execução normal do programa. Uma entrada de dados contendo apenas 1 round e 5 participantes foi utilizada. Ademais, foram definidas 5 fases distintas de estudo, cada qual referente a um passo no processamento da partida, sejam eles: a leitura e a atualização das informações preliminares de cada participante, a execução do teste de sanidade, o desconto da aposta inicial, a atualização do montante com base nas apostas individuais e a verificação do vencedor do round. Além disso, o registro dos acessos à memória foi feito com base na leitura e na escrita dos nomes dos jogadores. A razão para isso se dá por esse atributo descrever com alguma precisão a maneira com que as manipulações da lista são realizadas. O nome é escrito apenas uma vez na criação da célula e, para cada operação, este é usado para se identificar o jogador desejado. Assim, a partir dele, é possível visualizar com alguma coerência os acessos realizados a essa estrutura de dados, seja ela o principal objeto de estudo da presente análise. Seguem os resultados:

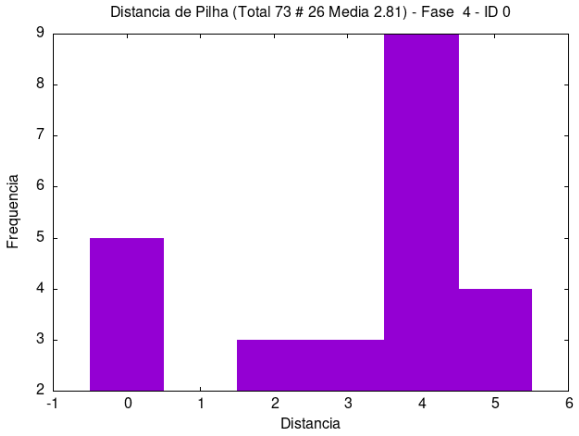












Os dados obtidos expõem a natureza sequencial de uma lista duplamente encadeada. A cada acesso realizado no processamento da partida, as células são percorridas de maneira ordenada até que se encontre a desejada. No mapa de acessos, esse comportamento fica claro pela maneira com que, ao longo do tempo de execução, os jogadores são percorridos (seus nomes são lidos) um a um até que algum específico seja encontrado. A partir daí, alguma operação é realizada e, para uma eventual próxima, a lista é novamente percorrida do início (a sequência de leituras recomeça da primeira célula). Nos diagramas de distância de pilha, por sua vez, esse comportamento fica evidente pelo aspecto visual de "escada" comum a cada fase da execução. Para o acesso a um jogador, todos os outros antes dele devem também ser acessados, de modo que seus dados são carregados na pilha de maneira crescente, originando a silhueta de uma "escada". Um último detalhe a ser mencionado é que essa abordagem de acessos sequenciais não é muito eficiente. A partir do mapa de acessos, é possível ver que a localidade de referência do programa é relativamente esparsa, enquanto que os valores presentes nos diagramas revelam uma distância de pilha razoavelmente elevada, ambas essas características resultantes da necessidade por todas as células anteriores a uma desejada terem de ser carregadas em memória.

6. Conclusões

O presente trabalho prático abordou o projeto e a análise de algoritmos sob o contexto do desenvolvimento de uma aplicação que operacionaliza as etapas de decisão e de movimentação de capital de uma partida de pôquer. A partir de um conjunto de especificações, foi elaborado um programa que recebe como entrada um acervo de informações relativas ao jogo e que, após um processamento, gera uma saída

contendo seus resultados. Foi adotada uma abordagem orientada a objetos em que toda a ideia da partida foi modelada a partir de uma estrutura de dados sobre a qual manipulações e consultas são exercidas, com suporte em classes que abstraem os principais elementos do problema.

A principal dificuldade observada diz respeito justamente à maneira como a realidade poderia ser abstraída em termos computacionais. Mesmo que houvesse especificações, essa questão detém um campo extremamente amplo de possibilidades e a escolha de uma implementação que contemplasse simultaneamente quesitos como eficiência, robustez e integridade não foi uma questão trivial. Além disso, outra grande dificuldade foi com relação à revisão dos algoritmos implementados. O jogo de pôquer, como foi requisitado, abre margem para uma série de casos de execução particulares, os quais agregam um maior nível de complexidade à garantia de corretude e à avaliação de desempenho/localidade de referência.

Contudo, os maiores ganhos vieram justamente das oportunidades induzidas pelas dificuldades mencionadas. A proposta temática do trabalho, mais ampla e, de certo modo, mais "real" - no sentido de trazer um problema um tanto distante ao mundo da computação - permitiu o desenvolvimento de habilidades relacionadas ao pensamento crítico e às etapas de decisão no desenvolvimento de software, além de que esse maior nível de detalhamento na revisão e na análise algorítmica permitiu ampliar o escopo de visão sobre os problemas práticos da programação, que, em geral, envolvem diversas particularidades.

7. Bibliografia

Stack Exchange, Inc.. Stack Overflow [Online]. Disponível em: https://stackoverflow.com/ (Acessado em: 03/06/2022).

CPlusPlus. CPlusPlus [Online]. Disponível em: https://www.cplusplus.com/ (Acessado em: 03/06/2022).

GeeksforGeeks. GeeksforGeeks [Online]. Disponível em: https://www.geeksforgeeks.org/ (Acessado em: 03/06/2022).

Refsnes Data. W3Schools [Online]. Disponível em: https://www.w3schools.com/ (Acessado em: 03/06/2022).

Wikimedia Foundation. Wikipedia, The Free Encyclopedia [Online]. Disponível em: https://en.wikipedia.org/ (Acessado em: 03/06/2022).

8. Instruções para compilação e execução

1. Extraia o conteúdo do arquivo .zip disponibilizado;
2. Acesse o diretório **</TP>** por meio do comando **<$ cd TP>**;
3. Execute no terminal o comando **<$ make all>**;
4. Esse comando deverá gerar um executável denominado **tp1** no diretório **<TP/bin>** ;
5. O arquivo de texto contendo a entrada de dados para o programa deve ser colocado no diretório raiz **</TP>** com o nome **entrada**;
6. Execute no terminal o comando **<$ ./bin/tp1>**;
7. Um arquivo denominado **saida** deverá ter sido gerado no diretório raiz **<$ TP>** contendo a respectiva saída do programa para o arquivo de entrada fornecido.