Trabalho Prático 1
Poker

Pedro Gomes Santiago Pires Beltrão - 2021039760

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

pedrosantiago@ufmg.br

1. Introdução

O problema proposto para o trabalho é desenvolver um programa que leia uma entrada de um jogo de poker de 5 cartas e através das especificações do jogo, realizar todas as rodadas, encontrar seus vencedores e imprimir o dinheiro dos jogadores ao final de todas as jogadas. Depois de obter as saídas com esses valores, é preciso analisar e interpretar os dados para compreender o comportamento do programa de acordo com a quantidade de jogadores e rodadas. Dessa forma o programa é modularizado com os headers e classes utilizadas para sua realização e o programa com o desenvolvimento dessas funções e o main que as executa. Entre os módulos tem-se 'poker.h' com os headers, 'poker.c' com o desenvolvimento do jogo e, por fim, 'memlog' que auxilia na interpretação dos dados e desempenho do programa. As classes foram pensadas para facilitar a realização do jogo e sistematizar os atributos de cada uma. A 'mesa' possui os atributos de entrada que são iguais para todas as rodadas e devem ser utilizados durante a partida, os 'jogadores', possuem todos atributos que vão sendo alterados ao longo de cada rodada e, por fim, as cartas possuem número e naipe, representando as cartas do baralho.

2. Implementação

O desenvolvimento foi realizado na linguagem C e compilado pelo G++ da GNU Compiler Collection.

As estruturas de dados, objetivo de estudo do programa, foram os métodos de ordenação, além da utilização de classes e vetores para organização e funcionamento do programa. Em relação aos vetores: é muito utilizado em diversos programas pois é uma forma simples de organizar dados, já que eles podem ser lidos como vetores e, depois de serem organizadas dessa forma, é facilitado seu uso e estudo. Pois é possível facilmente acessar, comparar e realizar operações com os elementos desse vetor. Os vetores são compostos por uma linha, então para acessá-los basta percorrer os índices da forma que for mais eficiente. Além disso, os métodos de ordenação também são utilizados de forma recorrente em pesquisas, estudos e no cotidiano de forma geral. Isso ocorre, porque para interpretar dados, é preciso, muitas das vezes, ordená-los, já que dessa

forma, teremos uma visualização gráfica mais simples de como eles estão ocorrendo e se comportando. Após a ordenação temos os limites mínimos e máximos nos extremos, facilitando dessa forma coletar dados para determinadas ocasiões em que sejam desejados valores específicos. Pelo fato da ordenação ser uma manipulação realizada de forma frequente, esse método já é conhecido e estudado, durante a realização do programa será utilizado o método de seleção, visto que sua principal vantagem é ser utilizado em pequenas listas, como é o caso da mão dos jogadores (com 5 cartas), porém, dependendo da quantidade de rodadas, esse método pode se tornar custoso pro programa. Portanto, entende-se que os vetores são estruturas de dados essenciais, pois facilitam a manipulação e interpretação de dados e que os métodos de ordenação são essenciais para programas que precisam organizar dados e buscam um bom desempenho.

O programa foi dividido em diversas funções que facilitam seu funcionamento dentre as principais, podem ser listadas: 1. '**rodaPartida**': responsável, primeiramente, por ler os parâmetros da partida, com número de rodadas, dinheiro, apostas, nomes dos jogadores e cartas, depois por chamar a função rodaPartida, ou seja, quase todo o programa ocorre dentro dessa função, com exceção dos construtores. Recebe como parâmetros: a mesa, os jogadores (10 que é o valor máximo permitido com um baralho), os nomes dos arquivos de entrada e saída. 2. '**escreveSaida**': essa função escreve todos os dados de saída computados no arquivo de saída, mostrando os vencedores de cada rodada e, ao final da última rodada, todos os jogadores e seu montante. Existe um tratamento especial para o caso de sanidade e apostas não múltiplas de 50, em que a saída é uma invalidação da rodada e os atributos dos jogadores não são alterados. Recebe a mesa, os jogadores, o arquivo de saída, um indicador do teste de sanidade, número da rodada e quantidade total de jogadores. *observação: todas as funções que se seguem recebem os jogadores e a mesa, algumas também recebem o número total de jogadores, porém não será especificado para poupar redundância na escrita. 3. '**escreveSaidaDesafio**': também realiza a escrita de dados, porém apenas para o jogador que mais blefa e melhor jogador. 4. '**ordenaCartas**': responsável por ordenar a mão dos jogadores, como já explicitado foi utilizado o método de seleção, com auxílio do define para trocar as cartas de posição. Através da ordenação é mais fácil classificar as frequências e sequências, além das cartas mais altas . 5. '**frequênciaCartas**': essa função cria um vetor que armazena a frequência que cada valor de carta aparece na mão do jogador, para facilitar a classificação das mãos dos jogadores. 6. '**sequenciaCarta**': similar a anterior, porém apresenta se as cartas formam uma sequência, além disso, também identifica se os naipes das cartas são iguais, assim como a última serve para a classificação da mão do jogador. Isso é feito através de um atributo do jogador que recebe um somador, ou seja para seq = 5, tem-se uma sequência de 5 cartas, para seqnipe = 5, tem-se uma sequência de 5 naipes. 7. '**normaliza**': realiza a normalização da mão, ou seja, transforma o valor de 1 em 14, visto que os áses em sua maioria têm

maior peso(nos casos que não possui é tratado separadamente. 8. '**jogaRodada**': executa as funções responsáveis por jogar uma rodada da partida. 9. '**inicializaJogadorMesa**': é o construtor tanto para mesa como para os jogadores, inicializa seus atributos antes de fazer a leitura e jogar a partida. 10. '**atualizaRodada**': ao final da rodada, essa função atualiza todos os valores necessários, como montante dos jogadores, valores de aposta e valores do pote. 11. '**classificaBlefe**': essa função é responsável por identificar os jogadores que mais blefaram durante a partida, vale ressaltar, que isso não significa que o blefe foi efetivo (o jogador pode ter perdido a rodada em que blefou). Para caracterizar o blefe, foi considerado uma pontuação que se dá pela razão entre o valor da aposta do jogador na rodada (dividido por 50 para padronizar valores) e a pontuação de sua mão (também normalizada). Vale ressaltar que a jogada só é considerada blefe caso seja um 'highcard', visto que dado o jogo com um baralho, a probabilidade de um jogador obter um 'highcard' é cerca de 50% enquanto 'onepair' é cerca de 42% e os outros 8% são divididos entre as outras mãos, ou seja, um jogador com 'onepair' já possui mais de 50% por cento de chance de ser mais forte que os demais, sendo assim uma mão forte. Então é provável que não esteja blefando dado que as apostas são feitas com a mão completa. Com isso, para os jogadores com 'highcard' caso sua aposta seja alta, isso aumentará sua pontuação de blefe, porém caso ele tenha um 'highcard' com valor alto (por exemplo quando o jogador possui um ás como maior carta), isso também diminuirá sua pontuação de blefe. O mesmo vale para apostas baixas e mãos com baixo valor. 12 '**classificaJogador**': a função para classificar o melhor jogador utiliza basicamente três critérios que são multiplicados, uma razão entre o dinheiro final e o dinheiro inicial do jogador, o quanto ele blefa e uma razão de quantas rodadas esse jogador ganhou pelo número de rodadas. Para ser justo e nenhum critério ter peso extremamente maior, todos valores foram normalizados antes da multiplicação. Todos esses critérios são consideráveis para ser um bom jogador de poker. O dinheiro recebido, mostra que o jogador apostou nas horas certas, a taxa de blefe, mostra que o jogador não joga apenas com boas mãos, mas também consegue extrair boas situações de mãos ruins, por fim, o número de rodadas vencidas mostra que tanto para o dinheiro recebido e para a taxa de blefe, o jogador foi efetivo, sendo assim muitas vezes o jogador que saiu com maior dinheiro da mesa, pode ter sido por sorte, porém o melhor jogador terá melhores resultados a longo prazo, por saber jogar com todos os tipos de mãos e apostas nas horas certas. Como tanto no caso do blefe, quanto do melhor jogador, os jogadores podem empatar, esses métodos aceitam mais de um "melhor jogador e melhor blefador". 13: **funções de classificação da mão**: existe uma função para cada classificação de mão dos jogadores, como mostrado anteriormente, elas utilizam as frequências e sequências para definir cada mão, para isso utilizam condições e operadores AND e OR, para garantir todos os casos. Vale ressaltar que foram tratados os casos especiais, como a sequência de 1 até 5 em que o ás tem peso menor. É importante compreender a forma de desempate, que é dada pela pontuação de

cada uma das mãos. Para funcionar, foi aplicada uma pontuação a cada mão, o importante é que a pontuação de uma mão nunca seja ultrapassada por outra mais fraca, além disso, para o desempate, as cartas também devem ter valores, portanto, as mãos assumem valores muito altos para que as cartas não interfiram nessa valor. Cada carta, de peso maior a menor assume diferentes valores que devem ser separadas por duas casas decimais, para que nos casos em que as maiores cartas tenham valores maiores que 10, não ultrapasse a pontuação da carta de menor valor. 14 '**classificaMao**': função responsável por rodar todas funções de classificação, é importante ressaltar que a ordem deve ser feita da mais fraca para a mais forte, pois existem mãos que possuem diferentes classificações e a mais forte deve prevalecer.

Como foram utilizadas as três classes já explicitadas, as funções não possuíam retorno, já que todos os dados necessários estavam armazenados em atributos das classes, que eram alterados de forma dinâmica durante as próprias funções.

O main, apenas cria os jogadores com o valor máximo já definido, cria a mesa, chama a função 'inicializaJogadorMesa' e chama a função 'rodaPartida'.

3. Análise de Complexidade

No tópico de implementação foram citadas as funções mais importantes para o pleno funcionamento do programa, com isso também é preciso compreender qual a complexidade dessas funções, para analisar qual o impacto que isso terá no tempo de execução do programa. Vale ressaltar que a complexidade das funções do programa estão relacionadas principalmente aos loops e aos métodos de ordenação. Como muitos dos loops que serão analisados para a complexidade do programa possuem limite máximo de voltas no caso específico de aplicação, serão explicitados os limites máximos para compreender a verdadeira complexidade. Os limites máximos de 10 estão ligados à quantidade de jogadores e os limites 5 estão ligados às cartas da mão do jogador.

Nas funções 'ordenaCartas' e 'frequenciaCarta': tem-se 3 loops, ou seja, complexidade proporcional aos loops O(n^3), loops com limite máximo de 10, 5 e 5. Apesar de possuir alta complexidade, no caso do programa exige pouco tempo de processamento. Nas funções 'sequenciaCarta' e 'normaliza': 2 loops, complexidade O(n^2), com limite máximo 10 e 5. Todas as funções de classificação possuem apenas um loop e condições que não alteram sua complexidade, sendo assim são O(n), porém esse loop tem limite máximo 10. As funções 'inicializaJogadorMesa', 'classificaBlefe' e 'classificaJogador' têm complexidade O(n) com limite máximo 10. O mesmo para 'atualizaRodada', que possui dois loops porém separados um do outro, sendo assim O(n) + O(n) = O(n), com limite máximo de 10. Nos casos de saída 'escreveSaída' possui diversas complexidades diferentes, sendo o resultado a maior entre a soma delas, O(n^2), com limites 10 e 10. Já para 'escreveSaídaDesafio': tem-se O(n) com limite 10.

Funções mais complexas:

‘classificaMao’: por ter que rodar todas as funções de classificação de mão, possui complexidade O(n) + O(n) +O(n) …. + O(n), que dá O(n), porém se for analisada a função de complexidade essa função tem gasto maior em relação a cada classificação separada.

‘jogaRodada’: como roda todas as funções necessárias para acontecer a rodada, essa função é a soma da complexidade de todas funções que ela chama, porém, sabe-se que a soma de diversas complexidade resulta na complexidade de maior ordem, sendo assim, a complexidade é O(n^3), lembrando a importância de ressaltar os limites máximos.

Após a análise de todas as funções com exceção da função de rodar a partida, é preciso explicar que, como esses casos foram analisados explicando o limite máximo, na realidade o impacto que eles têm no programa é baixo, por exemplo, o maior número de casos explorado é da função de ordenação e frequência das cartas, que são O(n^3), por se tratarem do pior caso do método de seleção e realizá-lo para a mão de todos os jogadores. No entanto, como os limites máximos são 10, 5 e 5, no pior dos casos serão 250x no loop, que é considerado um valor baixo de atribuições realizados, ou seja, no caso específico do programa, onde o número de jogadores e número de cartas são limitados, todas essas funções podem ser vistas com O(1), por serem valores baixos e constantes. Porém, caso fossem permitidos, por exemplo, mais jogadores, onde o limite máximo era 10, passa a ser infinito, então a complexidade realmente seria O(n) e O(n^2), nos casos em que o limite máximo é 10.

Analisando, por fim, a função ‘rodaPartida’, nota-se que essa função tem a complexidade ditada por uma variável que não é limitada, o número de rodadas, sendo assim, sua complexidade é dada pelo loop que comtempla esse valor. Como apenas um loop utiliza esse limite, a complexidade real do programa é O(n), pois é a complexidade desse loop multiplicado pela complexidade de ‘jogaRodada’, que foi considerada como O(1). No entanto, no caso de poder variar o número de jogadores e cartas ilimitadamente, essa função teria complexidade O(n^4), que é extremamente grande. Nesse caso não seria indicado a utilização do método de seleção devido ao alto custo de desempenho e memória que exigiria.

Com isso considerando as matrizes quadradas de dimensão nxn, chega-se à conclusão que todas as funções possuem, nesse caso específico a mesma ordem de complexidade, exceto pela função ‘rodaPartida’, que apresenta grande peso nesse contexto, por ser uma ordem de complexidade realmente afetada pela infinidade que sua entrada pode receber. Assim, entende-se que a função ‘rodaPartida’ será a principal responsável pelo tempo de execução do programa.

4. Estratégia de robustez

Sabe-se que a programação defensiva tem papel essencial no desenvolvimento de programas, à medida que a sua complexidade é aumentada, por isso, na realização deste, foram utilizadas técnicas que permitiram a confecção do código e identificação de erros ao longo do desenvolvimento. Sendo assim segue a lista de bons modos e programação defensiva pensados para o programa.

Um dos pilares da programação defensiva, é utilizar mensagens de erros para possíveis problemas que podem aparecer no programa, com isso busca-se compreender quais trechos podem apresentar erros, ou quais trechos são críticos no programa e utilizar mensagens. No caso do programa, foi utilizado o header ‘msgassert.h’, que permitiu a utilização de ‘erroAssert’ durante o programa, que permite estabelecer uma condição e uma mensagem de erro caso essa condição seja infligida. Esse método gerou grande impacto na identificação de erros, permitindo sua correção de forma eficiente.

Outro fator interessante explorado na realização dos códigos foi a modularização, que permite que o programa seja dividido em partes, cada uma delas tendo uma função diferente para o programa. Como foi explicado anteriormente, a divisão e o funcionamento. Esse processo facilita a visualização e compreensão de execução do problema, facilitando, consequentemente, a identificação de possíveis problemas. Além disso, também foram utilizadas diferentes pastas para separar os arquivos ‘.c’ e os ‘.h’, que carregam os cabeçalhos das funções e construtores do TAD, também facilitando a compreensão do código.

A análise de complexidade também foi crucial para conseguir um programa otimizado, visto que, já sabendo quais eram as funções necessárias pro programa, poderiam também ser estudadas quais as complexidades para descobrir como seria o melhor desempenho em relação a tempo e gasto de memória. Sendo assim as funções foram pensadas para ter a menor complexidade e uma execução ótima.

Os comentários realizados ao longo do programa, ajudam na compreensão das funções, do que está sendo realizado e especificidades de determinados trechos do código. A sua utilização foi frequente nas partes que eram essenciais, para que o código possa ser interpretado facilmente pelo programador e por outras pessoas que tenham acesso.

Frequentemente a compilação e execução do programa pode ser um problema, devido a sua complexidade. Com objetivo de suprir a necessidade de contornar essa situação, o makefile auxilia e facilita no entendimento da compilação e da execução.

A análise de localidade e referência permitiram que código fosse otimizado, facilitando a visualização de trechos que eram muito custosos e que poderiam ser melhorados. Além disso, também facilitaram na análise de complexidade das funções.

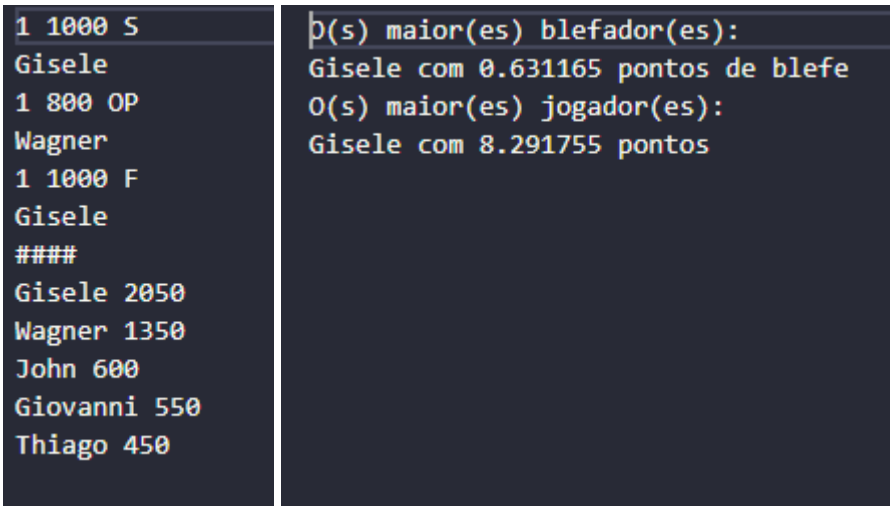
O uso de funções é uma técnica que permite que o mesmo trecho do código não seja reescrito diversas vezes, facilitando sua leitura. Todo o programa é dividido em funções que permitiram que o código ficasse mais enxuto e legível.

O uso de TADs, assim como as funções, facilita a leitura do código e diminui a incidência de reescrita. O TAD permite atribuir valores e funções ao tipo abstrato criado, facilitando assim o seu uso e a leitura do código, já que seu aspecto principal pode carregar "características".

5. Testes

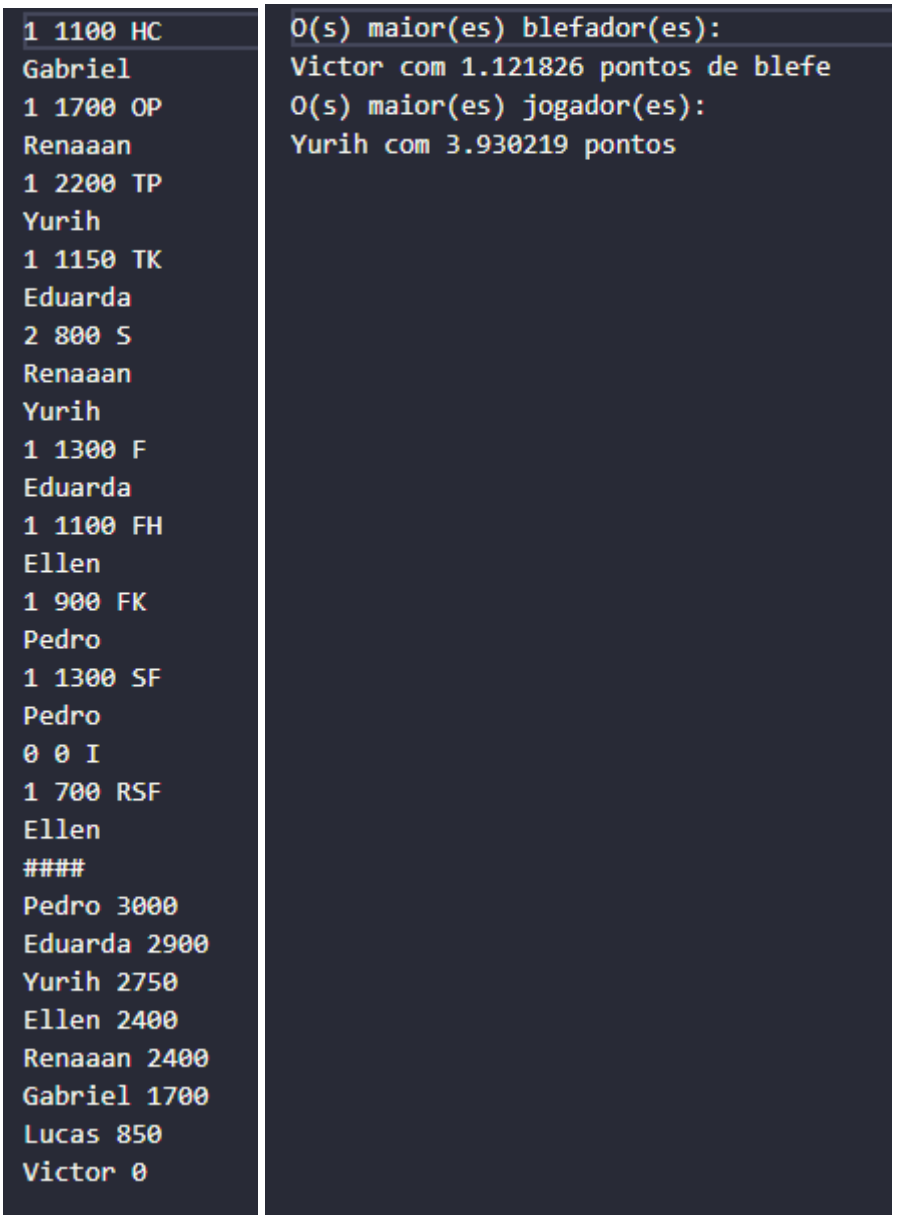
Para compreender o funcionamento do programa é preciso realizar testes com diferentes entradas para poder obter diferentes saídas e estudar o seu comportamento.

O primeiro caso estudado foi o exemplo de entrada da documentação: tem-se que as saídas estavam dentro do esperado, com os valores buscados, além de também fornecer os maiores blefadores e jogadores de acordo com as regras explicadas anteriormente.



Para uma partida com uma entrada maior, mais jogadores, mais rodadas, mãos variadas, o jogo continuou funcionando normalmente e apresentando os valores de saída esperado, além de realizar corretamente os desempates seguindo os critérios de cartas mais fortes e, também, considerar mais de um vencedor em ordem alfabética em casos que eram necessários. Além disso é interessante notar como nos casos em que a entrada possui mais rodadas o algoritmo para definir quem blefa mais e quem é o melhor jogador se torna mais preciso, pois existem mais dados para avaliar os jogadores. Nesse caso em específico, o jogador que mais blefou foi o que terminou com menor montante

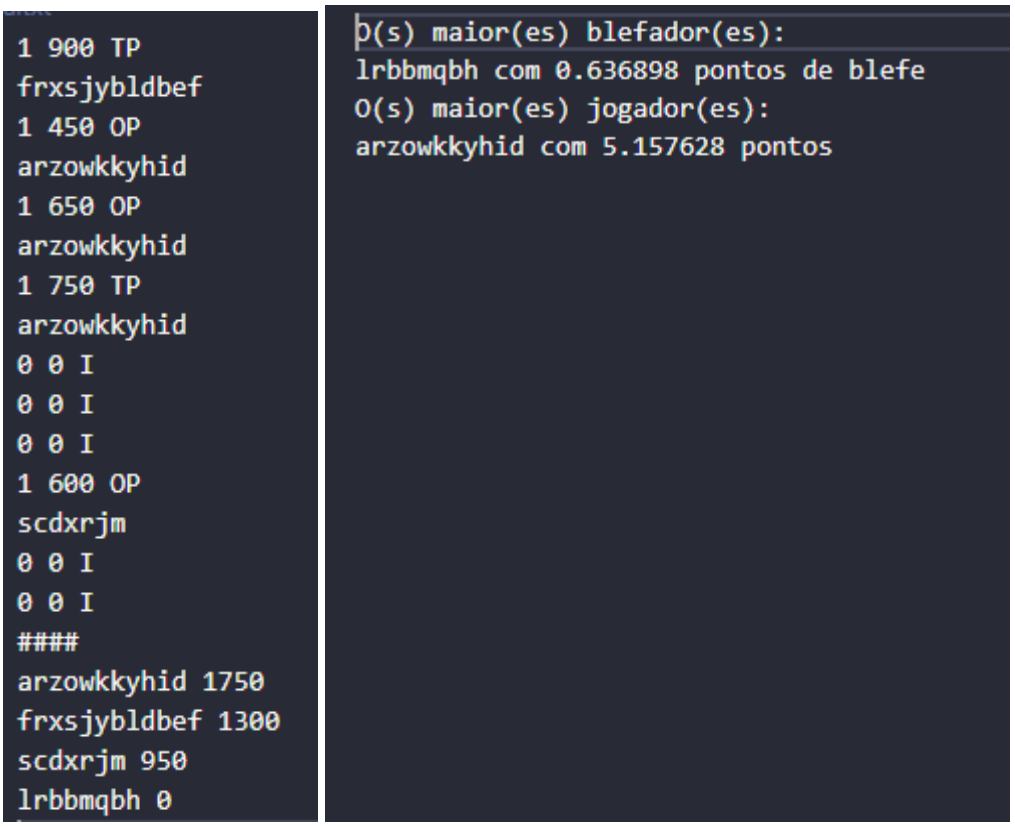
e o melhor jogador terminou com o terceiro maior montante.



```
1 1100 HC
Gabriel
1 1700 OP
Renaaan
1 2200 TP
Yurih
1 1150 TK
Eduarda
2 800 S
Renaaan
Yurih
1 1300 F
Eduarda
1 1100 FH
Ellen
1 900 FK
Pedro
1 1300 SF
Pedro
0 0 I
1 700 RSF
Ellen
####
Pedro 3000
Eduarda 2900
Yurih 2750
Ellen 2400
Renaaan 2400
Gabriel 1700
Lucas 850
Victor 0
```

```
O(s) maior(es) blefador(es):
Victor com 1.121826 pontos de blefe
O(s) maior(es) jogador(es):
Yurih com 3.930219 pontos
```

Por fim, pegando outro exemplo com mais rodadas, gerado pelo gerador fornecido, tem-se os seguintes resultados. Nota-se que normalmente, o jogador com menor montante costuma a ser também o jogador que mais blefou, visto que muitas vezes a mão deste jogador era ruim, porém isso não necessariamente o torna o pior jogador, podendo apenas ser o azar da partida. Também vale notar os testes de sanidade funcionando, para os casos em que o jogador não possuía dinheiro para o pingo mais a aposta na rodada era invalidada. Outro aspecto importante é notar que a soma dos montantes sempre será o dinheiro inicial vezes o número de jogadores total, já que o dinheiro não é diminuído nem aumentado durante a partida.



6. Análise experimental

Desempenho Computacional

Para a avaliação do do algoritmo, o principal objetivo foi modificar o valor de entrada para observar como isso afetaria o desempenho do programa, seu tempo de execução e saber como isso poderia ser otimizado ou deveria ser interpretado. Para isso foram testados valores com 10, 100, 1000 e 10000. Após testar para 10000 entradas, é possível notar que o programa já levaria muito tempo para ser executado com mais uma casa decimal, portanto as execuções foram interrompidas até esse valor. É possível notar que algumas modificações poderiam fazer o programa ser executado com entradas ainda maiores, porém busca-se a otimização para o caso de jogadas reais de poker, portanto o programa tende a funcionar melhor para entradas menores.

gráfico de número de rodadas x tempo de execução em segundos

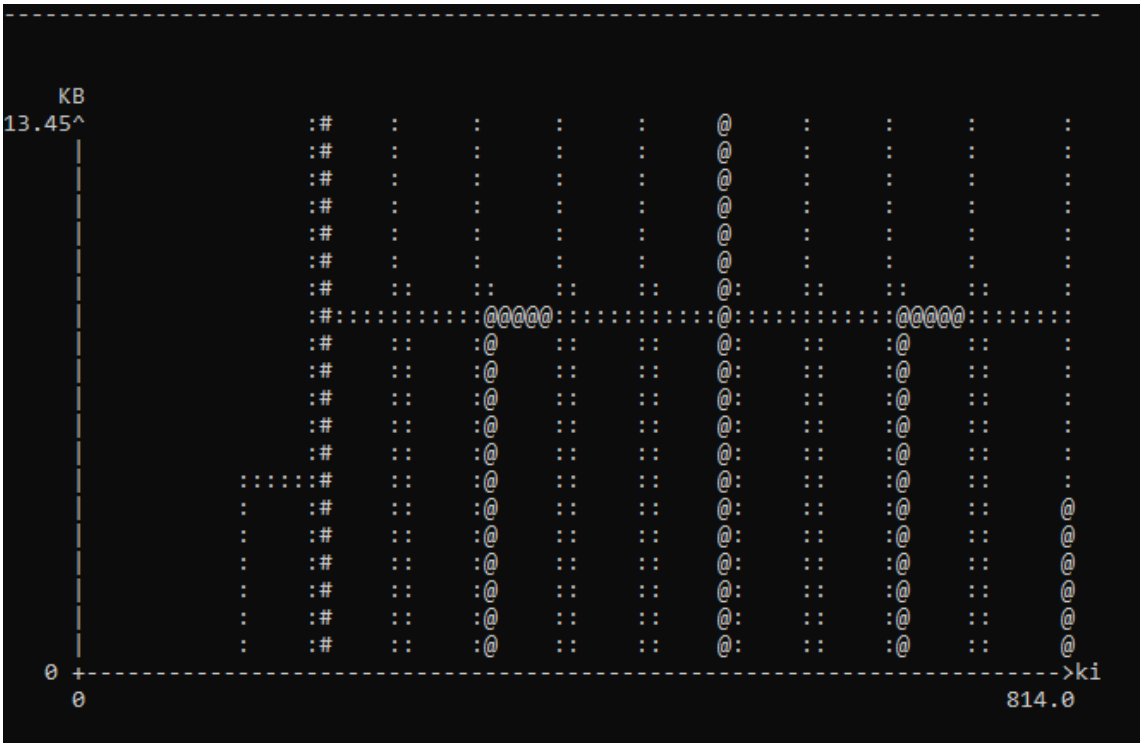
n rodadas	tempo(s)
10	0.039
100	0.169
1000	1.289
10000	13.797

dados para realizar o gráfico

Por meio da análise do gráfico é possível notar que o crescimento do tempo de execução é proporcional às rodadas adicionadas à partida. Como esperado pelas análises de complexidade anteriormente, o número de rodadas afeta diretamente o tempo de execução do programa, pois é o valor que pode tender a infinito e realizar a execução de todas as outras funções diversas vezes (10 mil como no pior caso apresentado). Apesar dos valores de limite máximo citados serem relativamente pequenos, podemos notar que

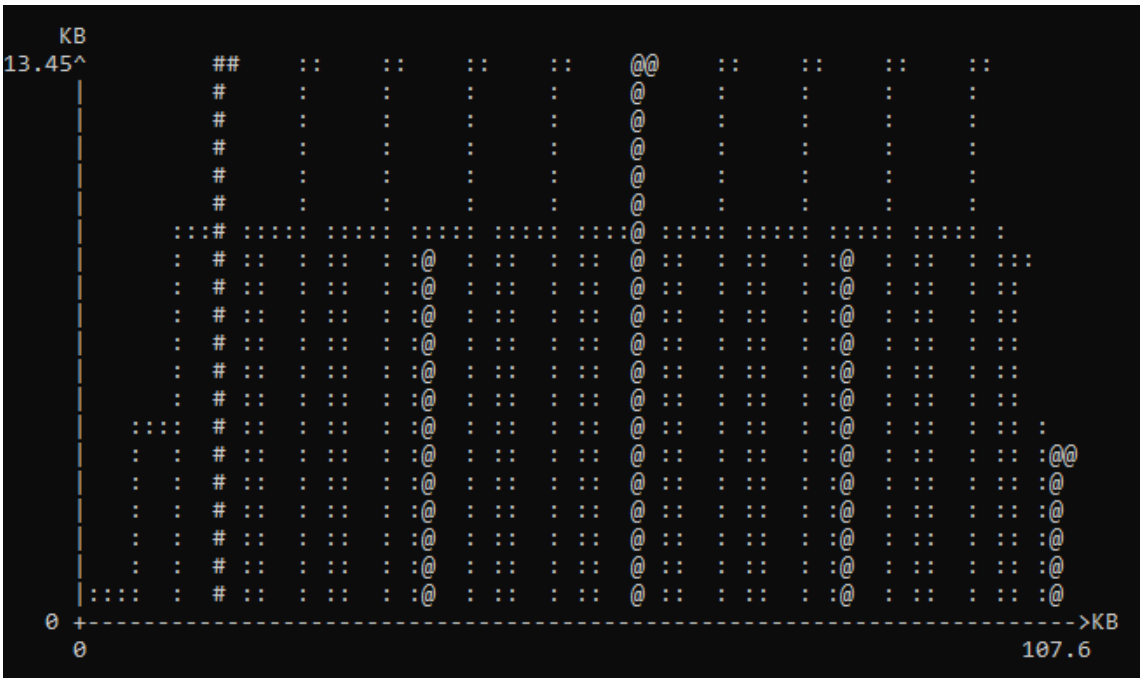
tiveram grande impacto no caso de números altos de rodadas, portanto, percebe-se que cada detalhe é importante para conseguir uma implementação otimizada do programa e que mesmo os limites máximos pequenos, podem influenciar de forma direta no desempenho, não podendo ser negligenciados.

Análise de Padrão de Acesso à Memória e Localidade de Referência



tempo em instruções executadas x memória gasta

Por meio do gráfico de acesso gerado pela ferramenta valgrind, é possível perceber o comportamento do programa no acesso de memória de acordo com as instruções executadas, ou seja, a memória consumida à medida que as instruções aumentam, também aumentam, pois no começo, por exemplo, as instruções são apenas de leitura, à medida que os dados são lidos, o processamento começa a ser realizado e a memória passa a ser utilizada de forma mais intensa, como é mostrado no gráfico. Além disso, existem diferenças proporcionais ao número de jogadores na rodada, porém como o gráfico representa uma entrada linear, em que sempre há o mesmo número de jogadores para todas rodadas, o gráfico se mantém simétrico em relação ao uso de memória. Isso também indica que o programa não apresenta variações drásticas ao longo do processamento dos dados, visto que não há grandes disparidades de memória utilizadas ao longo do seu tempo de execução, o que mostra boa localidade de referência e constância nos acessos aos jogadores e cartas.



unidade de tempo em bytes x memória gasta

Já nesse gráfico que muda a perspectiva, pois modifica a unidade de tempo, também conseguimos perceber a simetria e localidade de referência no gráfico. Para as diferentes funções executadas é possível observar os diferentes padrões que o gráfico adota, porém eles se repetem para cada uma das dez rodadas, além disso no início do programa reparamos o comportamento diferente representando a leitura dos dados e no final o comportamento diferente indicando a escrita dos dados representando todos os jogadores e seus montantes.

Vale ressaltar que ambos os gráficos vieram da mesma entrada, que será salva como ‘entradaAnalise.txt’ na pasta results, ela também foi utilizada na análise de desempenho, visto que os jogadores sempre saem com o mesmo dinheiro (para não cair no teste de sanidade), sendo assim pode-se multiplicar os casos para obter entradas com muitas rodadas.

Após analisar a localidade e referência, percebe-se como a estruturação do programa é importante para que ele tenha um bom desempenho e se mantenha funcional mesmo com variação nas entradas.

7. Conclusão

O objetivo do projeto era realizar um programa que pudesse ler as entradas de um jogo de poker e interpretá-las para, por fim, poder transcrever os resultados com vencedores e montantes somados para arquivos de texto. Também fazia parte do projeto, desenvolver códigos que permitissem a análise dos resultados e do programa, de forma a compreender quais são os fatores que impactam o tempo de execução. Vale ressaltar que

para realizar isso, era preciso utilizar boas técnicas de programação, como a programação defensiva, para realizar o código limpo e de fácil leitura.

Portanto, com a realização do trabalho, foi possível aprender e aprimorar métodos de realização de um programa para o mundo real, dentre esse aprendizados podem ser listados:

1. Como analisar e transformar os dados de entrada do jogo na saída correta, compreendendo casos específicos e desenvolvendo uma lógica para identificar mãos e compará-las ao longo da partida.(entendendo, principalmente, a complexidade do programa).
2. Importância de fazer análise de desempenho e localidade no programa, auxiliando dessa forma a compreender como otimizá-lo. O que era esperado foi compreendido, que se trata da relevância que essas análises têm para identificar como o programa pode ser melhorado.
3. O porquê de utilizar programação defensiva e bons modos de programação, que permitem que o desenvolvimento ocorra de forma mais rápida, segura e eficiente, visto que os erros são inevitáveis no processo.

8. Bibliografia

https://efdeportes.com/efd211/as-habilidades-essenciais-para-jogadores-de-poker.htm#:~:text=Concentração%3A%20A%20mente%20de%20um,preciso%20esperar%20o%20momento%20certo

https://www.oddsshark.com/poker/games/five-card-draw#:~:text=Five%20Card%20Draw%20Strategy,three%20of%20a%20kind%2C%20respectively.

http://www.math.hawaii.edu/~ramsey/Probability/PokerHands.html

https://www.ehow.com.br/vantagens-desvantagens-algoritmos-ordenacao-info_16277/

https://valgrind.org/docs/manual/ms-manual.html#ms-manual.theoutputgraph

9. Instruções para compilação e execução

O programa está estruturado dentro da pasta TP, basta apenas adicionar arquivo de entrada com nome “entrada.txt” na pasta origem e dar make para pleno funcionamento do programa. Os arquivos de entrada na parte de teste foram salvos na pasta results.

Belo Horizonte - 13 de junho de 2022