Trabalho Prático 1

Poker Face

Lucas Rocha Laredo

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

lucaslaredo@ufmg.br

1. Introdução:

A proposta deste trabalho prático foi a implementação de um programa capaz de receber entradas equivalentes a um jogo de Poker, com uma, ou múltiplas rodadas, e retornar uma saída equivalente ao número de vencedores de cada rodada, o total ganho por cada um, a combinação vitoriosa e o nome de cada vencedor, além do total de dinheiro disponível para cada jogador ao final da partida. A fim de solucionar o problema proposto neste trabalho, o projeto é capaz de lidar com arquivos, lendo-os, gerando-os e manipulando-os.

2. Implementação:

O programa foi desenvolvido na linguagem C++, utilizando o compilador G++, da GNU Compiler Collection.

2.1. Estrutura de Dados e Classes:

A implementação do programa teve como base a estrutura de dados Lista, tendo como base o princípio da Programação Orientada a Objetos, possibilitado pela Linguagem de Programação C++. Foram implementados quatro tipos de dados abstratos (TAD): Carta, Jogador, Lista e Jogo, entre eles, a Estrutura de Dados ‘Lista’.

Em primeiro lugar, o TAD Carta representa a abstração das cartas de cada jogador. Para compor-lo, temos os seguintes métodos e atributos:

```
class Carta {
    public:
        int Numero;
        char Naipe;
        Carta();
};
```

O TAD Jogador é um pouco mais complexo, e representa os jogadores participantes do Jogo (outra Classe dentro do programa). Observa-se que um vetor de instâncias da classe anterior (Carta) é um componente da classe jogador, como percebe-se adiante pela lista de métodos e atributos:

```
class Jogador {
    public:
        string Nome;
        int Dinheiro;
        int Aposta;
        Carta Cartas[5];
        int RSF;
        int SF;
        int FK;
        int FH;
        int F;
        int S;
        int TK;
        int TP;
        int OP;
        int HC;
        int combNumber;
        int maisAltaFK;
        int maisAltaTK;
        int trincaFH, duplaFH;
        int maiorPar, menorPar;
        Jogador();
        void testaPares();
        void testaIguais();
        void testaFH();
        void testaStraight();
        void testaFlush();
        void testaSF();
        void testaRSF();
};
```

O próximo TAD é responsável por agrupar os jogadores dentro do jogo. Ou seja, dentro do Jogo existem múltiplos jogadores, e esses devem pertencer à uma lista (essa explicação estará mais completa adiante), e essa lista é implementada através do Tipo de Dados Abstrato Lista.

Antes de listar os métodos e atributos do tipo lista, é fundamental citar que, a partir do princípio da herança, a Classe ListaArranjo foi utilizada. Isso significa que ListaArranjo é uma classe que herda métodos da classe "mãe" Lista. Além disso, a Classe TipoItem também foi utilizada para dar suporte ao TAD Lista. Essa é utilizada para instanciar os itens da lista, ou seja, os jogadores, facilitando todo o processo - além de reforçar o princípio do encapsulamento. A seguir, estão os métodos do TAD Lista:

```
class TipoItem
{
    public:
        TipoItem();
        TipoItem(TipoChave c);
        void SetChave(TipoChave c);
        TipoChave GetChave();
        void Imprime();
    private:
        TipoChave chave;
        // outros membros
};
```

```
class Lista {
    public:
        Lista() {tamanho = 0;};
        int GetTamanho() {return tamanho;};
        bool Vazia() {return tamanho == 0;};
        virtual TipoItem GetItem(int pos) = 0;
        virtual void SetItem(TipoItem item, int pos) = 0;
        virtual void InsereInicio(TipoItem item) = 0;
        virtual void InsereFinal(TipoItem item) = 0;
        virtual void InserePosicao(TipoItem item, int pos) = 0;
        virtual TipoItem RemoveInicio() = 0;
        virtual TipoItem RemoveFinal() = 0;
        virtual TipoItem RemovePosicao(int pos) = 0;
        virtual void Imprime() = 0;
        virtual void Limpa() = 0;
    protected:
        int tamanho;
};
```

```
class ListaArranjo : public Lista
{
    public:
        ListaArranjo() : Lista() {};
        TipoItem GetItem(int pos);
        void SetItem(TipoItem item, int pos);
        void InsereInicio(TipoItem item);
        void InsereFinal(TipoItem item);
        void InserePosicao(TipoItem item, int pos);
        TipoItem RemoveInicio();
        TipoItem RemoveFinal();
        TipoItem RemovePosicao(int pos);
        void Imprime();
        void Limpa();
    private:
        static const int MAXTAM = 100;
        TipoItem itens[MAXTAM];
};
```

A escolha de uma Lista foi feita a partir da necessidade de agrupar os jogadores participantes da partida - e de cada rodada, independentemente - em um único conjunto. Partindo disso, e da necessidade de uma implementação simples, mas eficiente, a lista se tornou uma opção viável, por ser mais complexa que um simples array, mas não tão complexa como uma Árvore Binária. Logo, a Lista, dentro desse programa, é responsável por armazenar os jogadores, e os seus diversos métodos (listados anteriormente, e explicitados adiante) tratam dos procedimentos ao longo do programa. Contudo, é importante citar que em termos de eficiência computacional, a Lista Arranjo não é a mais eficiente, dado que um tamanho fixo de memória é alocado. Ou seja, caso nem todas as posições alocadas da lista sejam utilizadas, teremos um desperdício de memória durante a execução do programa.

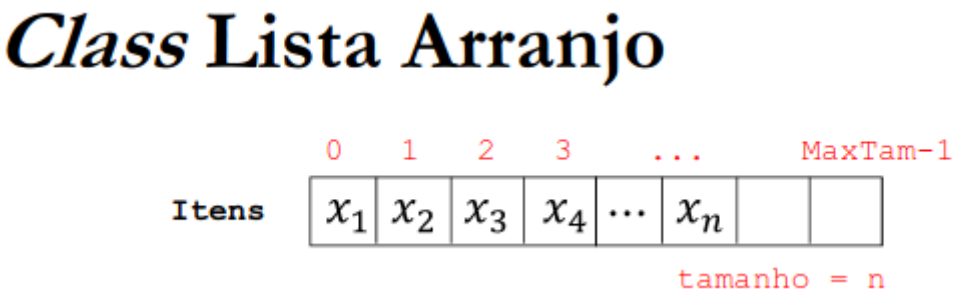


Figura 1. Diagrama da Lista Arranjo

Entretanto, essa escolha foi pensada tendo como base não apenas a eficiência computacional do programa, mas principalmente a sua simplicidade. Isso significa que uma Lista Arranjo possui uma implementação mais simples do que uma Lista Encadeada, que, por mais que seja mais eficiente, possui uma implementação infinitamente mais complexa, com ponteiros apontando para o início, e para o nó seguinte (Figura 2).

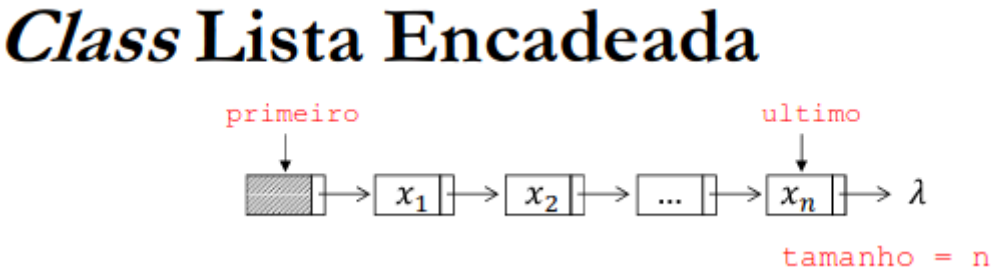


Figura 2. Diagrama Lista Encadeada

Por fim, o TAD Jogo foi implementado, reunindo todos os Tipos Abstratos citados anteriormente. Esse é responsável pelos atributos e métodos fundamentais para composição da partida de Poker:

```
class Jogo {
    public:
        int quantiaRodada;
        int numeroRodadas;
        Jogo(){quantiaRodada = 0; numeroRodadas = 0;};
        void Bolha(Carta *v, int n);
        ListaArranjo iniciaRodada(ListaArranjo listaJogadores, string nomeArquivo);
        void leArquivo(string nomeArquivo, ListaArranjo listaJogadores);
        ListaArranjo testaVitoria(ListaArranjo listaJogadores, int poteRodada, ListaArranjo listaJogo);

};
```

A partir desses TADs implementados, foram definidos os métodos e procedimentos que englobam todas as especificações exigidas pelo enunciado.

2.2. Métodos e Procedimentos utilizados e Funcionamento:

Os principais métodos e procedimentos utilizados na implementação deste programa estão distribuídos ao longo das classes. Essa é uma estratégia utilizada para deixar o programa com uma implementação mais organizada e eficiente.

Começamos com a leitura de arquivos. Essa leitura é feita por meio do método *leArquivo* (`void leArquivo(string nomeArquivo, ListaArranjo listaJogadores);`). Esse método é fundamental, pois ele é responsável por preencher a lista que armazena os jogadores participantes da partida. A partir dessa abordagem, é importante mencionar os métodos de preenchimento da ListaArranjo (Classe filha do TAD Lista). Antes disso, reforço que as classes 'Lista', 'ListaArranjo' e 'TipoItem' possuem métodos do tipo *get* e *set* - *ex:* `void SetChave(TipoChave c);` - e existem para manter o encapsulamento dentro dessas classes. Retornando a ListaArranjo em si, além dos métodos citados anteriormente, existem aqueles responsáveis pela inserção de novos '*TiposItens*' - leia-se Jogador - em diferentes posições da lista, assim como pela remoção desses '*TipoItens*'. Esses métodos são *InsereInicio, InsereFinal, InserePosicao (ex:* `void InserePosicao(TipoItem item, int pos);` e para remoção, *RemoveInicio, RemoveFinal* e *RemovePosicao (ex:* `TipoItem RemovePosicao(int pos);`).

Em seguida, a partir da lista de participantes já preenchida, o próximo passo é iniciar as rodadas. Intuitivamente, o método responsável por essa ação é *iniciaRodada* (`ListaArranjo iniciaRodada(ListaArranjo listaJogadores, string nomeArquivo);`). Esse método é, talvez, o mais complexo de todo o programa. Ele realiza novamente a leitura do arquivo, mas dessa vez, o arquivo completo. O que isso significa? No método anterior (*leArquivo),* que chama o método atual, apenas a primeira rodada é lida, uma vez que todos os participantes da partida estão nessa rodada obrigatoriamente. Para inserir os jogadores de cada rodada em uma nova lista, *jogadoresRodada*, que será tratada futuramente por outros métodos, as cartas dos jogadores são ordenadas de forma crescente, utilizando o método Bolha, em relação aos seus números

(de 1 até 13), e, em seguida, todas as combinações são testadas. Aqui abre-se um novo parênteses, para explicar o funcionamento dos métodos responsáveis pela definição da combinação da mão de cada jogador. Na verdade, esses procedimentos estão dentro do TAD Jogador, e são os seguintes: *testaPares, testaFH, testaStraight, testaFlush, testaSF* e *testaRSF.* Esses cinco métodos são testados para cada jogador a ser inserido na rodada, e definem qual combinação eles possuem. Essa definição é a partir da mudança de valor no atributo correspondente à combinação, ou seja, caso um jogador tenha um Full House, por exemplo, o seu atributo FH terá valor 1, e os demais, valor 0. Após testar todas as combinações, inserimos o jogador ao final da lista de jogadores da rodada. Para finalizar o método da rodada, temos dois métodos essenciais chamados: *testaVitoria* e *atualizaDinheiro.* Mais breve, o método *atualizaDinheiro* é mais intuitivo, e é responsável por alterar o dinheiro que cada jogador da partida possui ao final de cada rodada.

O método *testaVitoria (ListaArranjo testaVitoria(ListaArranjo listaJogadores, int poteRodada, ListaArranjo listaJogo);)* também é essencial, ao passo que a sua função faz justamente o que está em sua definição: define os jogadores de cada rodada. Esse teste preenche em um vetor de 10 posições (número de combinações), na posição correspondente ao "peso" de cada combinação, o valor 1. Caso mais de um jogador possua a mesma posição com o valor 1, o método auxiliar *testaEmpate* é chamado, e cada um dos possíveis casos de desempate são testados. Caso nenhum deles seja satisfeito, mais de um jogador é campeão daquela rodada, dividindo o prêmio pelo número de vencedores.

Por fim, ainda dentro do método *testaVitoria*, um método auxiliar *imprimeCombinacao* escreve no arquivo de saída - *saida.txt* - o número de vencedores da rodada, o total de dinheiro recebido por cada vencedor e a combinação vencedora, assim como os nomes dos jogadores vencedores daquela rodada. Retornando para o método *leArquivo,* ele também é responsável pela impressão dentro do arquivo de saída. No caso, ele imprime o nome dos participantes do jogo, seguido pela quantia de dinheiro restante ao final da partida.

Sendo assim, esses são os principais métodos e procedimentos que compõem o programa 'poker.out'. Existem alguns métodos auxiliares que possuem uma função muito intuitiva partindo de seus nomes, logo, não é tão essencial explicitá-los. É o caso dos métodos *preencheDinheiro* e *contaPalavras,* que, respectivamente, atualiza o dinheiro de cada participante da rodada, e conta o número de palavras do nome de cada jogador, fundamental para permitir nomes compostos no programa.

2.2. Análise de desempenho:

Os métodos implementados possuem complexidade não muito interessante do ponto de vista computacional, como será abordado adiante. Isso significa que em termos de desempenho, a performance do programa pode ser bastante prejudicada. Ou seja, dependendo do número de rodadas, e do número de participantes, o desempenho do programa é afetado.

3. Análise de Complexidade:

Em primeiro lugar, vamos começar com a análise de complexidade do leitura de arquivos.

O nome do arquivo de entrada é definido por padrão como "entrada.txt", e a primeira leitura é feita no método *leArquivo*. Serão feitas duas leituras, com complexidades diferentes. A primeira é feita apenas para a primeira rodada do jogo, apenas para preencher a lista de jogadores da partida. Nesse caso, o custo é O(n), uma vez que caminhamos apenas as n posições dos 'n' participantes da partida. A segunda leitura é mais complexa, uma vez que teremos mais de uma rodada sendo lida. Com isso temos dois loops aninhados, um para o número de rodadas, e outro para preencher a lista com cada jogador dentro daquela rodada. Ou seja, a complexidade se torna $O(n^2)$. Lembrando que, para ambas leituras, o melhor caso é O(1), se houver apenas um participante. Como isso não é possível, pois trata-se de um jogo de poker, esse melhor caso nunca ocorrerá. Na prática, a segunda leitura de arquivos possui uma complexidade um pouco maior, pois outros loops estão aninhados dentro desse método, através de outros métodos e procedimentos, mas isso será tratado adiante. O método de ordenação das cartas dos jogadores, presentes em ambas leituras de arquivos, possui complexidade $O(n^2)$ no pior caso, e O(1) no melhor caso, mas como são sempre 5 cartas, o melhor caso não é nunca atingido. Além disso, para contabilizar o número de palavras no nome de cada jogador utilizamos o método *contaPalavras*, com complexidade O(n). Para finalizar essa discussão em torno da leitura de arquivos, é importante citar que adicionar um novo item ao final da lista de jogadores possui custo O(1), reforçando o que foi dito anteriormente: A escolha da Lista Arranjo proporciona simplicidade ao código, além de, em termos de complexidade, oferecer alguns ganhos.

Em seguida, temos os métodos de testes. Dentro do problema proposto, é fundamental realizar vários testes, contudo, a otimização desses não é trivial. Sendo assim, começamos pelos métodos que definem a combinação dentro da mão do jogador. Os métodos *testaSF* e

testaFH possuem complexidade equivalente à O(1), já que cada um desses métodos é composto apenas por uma comparação (*if*) e atribuições. Já os demais, todos são mais complexos, e todos possuem complexidade O(n), uma vez que há uma iteração dentro de todos as cartas do jogador, testando os casos de combinação. Portanto, como passamos por todos esses métodos para cada jogador, temos:

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

Continuando com os testes, chegamos aos mais complexos: *testaVitoria* e *testaEmpate.* Como o método *testaVitoria* chama o método *testaEmpate* caso exista mais de um possível ganhador dentro da rodada (mais de um jogador com a mesma combinação), começamos pela análise do método *testaEmpate*. A composição desse é dada, basicamente, por uma sequência de comparações, do tipo *switch case.* Contudo, essas comparações, de complexidade O(1), devem ser feitas para cada jogador. Logo, a complexidade do método em si, passa a ser O(n), pois percorremos os n possíveis ganhadores. Já o método *testaVitória* também é composto por múltiplas comparações, entretanto, também existe um laço percorrendo os n jogadores da rodada, preenchendo suas combinações. Nesse caso, o método *preencheCombinacoes,* que define qual é a combinação do jogador, possui complexidade $O(n^2)$, e está dentro de *testaVitoria*. Ou seja, nesse caso, temos:

$$O(max(n, n^2)) + O(n) = O(n^2)$$

Como complexidade para testar qual o vencedor de cada rodada.

Por fim, é necessário atualizar o dinheiro de cada participante ao final de cada rodada, isso é feito por meio do método *atualizaDinheiro*. Esse método precisa percorrer todos os jogadores da partida, e os jogadores da rodada, comparando seus nomes, ao passo que, caso tenham nomes iguais, o dinheiro do jogador da partida será atualizado em relação ao que foi apostado na rodada. O mesmo é feito com a vitória, em *atualizaDinheiroVitoria,* em que o dinheiro é atualizado, mas agora com o valor ganho, em caso de vitória.

Desse modo, fica evidente que nenhuma operação de custo $O(n^3)$ é realizada. Isso é positivo para o programa, entretanto, a complexidade do programa não é das mais simples. Ou seja, isso indica que para entradas maiores, com mais rodadas e mais participantes, o tempo de execução do programa pode ser afetado, como citado anteriormente.

3.2. Espaço:

Cada jogador dentro do programa ocupa uma posição em uma Lista simples. Entretanto, como a escolha do programa foi feita com foco na simplicidade, a alocação de espaço não é das mais eficientes. Isso significa que um tamanho fixo de posições na memória será alocado independente do número de jogadores.

Em um cenário ideal, em um jogo com *n* jogadores, *O(n)* seria a complexidade do programa. Ou seja, uma complexidade linear. De fato, essa complexidade linear se mantém, em uma implementação de Lista Arranjo, mas esse n deveria depender do número de jogadores, não de um valor estático. Na prática, é alocado um vetor, de tamanho *MAXTAM = 1000* posições, que armazena os *n* jogadores. Independentemente do número de participantes, esse valor é alocado. É fundamental frisar que essa é uma implementação delicada, pois há um número máximo de participantes suportados, no caso, 1000. Caso esse valor seja ultrapassado, o programa não irá funcionar.

4. Estratégias de Robustez:

Todo programa mais robusto exige mecanismos para evitar bugs. Desse modo, para a depuração foi utilizado o gdb. Além disso, condicionais foram inseridos ao longo do programa. Elas evitam uma série de possíveis incompatibilidades. Por exemplo, para evitar que um jogador entre em uma rodada apostando menos que o '*pingo',* foi inserida uma condicional que garante que o jogador deposite pelo menos o mínimo. Caso não seja satisfeita essa condição, a aposta é desconsiderada. Além disso, é garantido que o jogador tenha o dinheiro para a aposta. Caso ele aposte um valor mais alto do que possui, a sua aposta também será desconsiderada. Além disso, dentro das listas, existem diversos pontos de garantia, através de *exit()*, para não possibilitar o acesso à posições não existentes dentro do vetor. Por exemplo, caso uma posição não preenchida do vetor, ou seja, uma posição que não possui um *TipoItem (Jogador),* tente ser acessada, o exit() será ativo, junto com a mensagem de erro. O mesmo é válido para tentar remover, ou adicionar um item em uma posição inválida. Isso tudo está presente para seguir a documentação exigida para o programa, assim como o seu bom funcionamento. É fundamental que a lógica seja obedecida dentro do programa, para evitar falhas e bugs.

É exigido também que o arquivo de entrada tenha nome "entrada.txt", caso contrário, a leitura do arquivo não será realizada, e uma mensagem de erro será ativada. Esse é um controle importante, pois garante o nome do programa, como pedido nas instruções. Além

disso, o arquivo de saída sempre terá o mesmo nome, "*saida.txt"*, outra exigência da documentação do programa.

5. Testes:

Uma série de testes foram realizados, o objetivo principal era testar se o programa continua estável, e com bons resultados com variações na entrada. No caso, foram tomados grupos menores, com menos rodadas, para avaliar os requisitos do programa, exigidos no enunciado, e maiores, com mais rodadas e participantes, com o objetivo de testar a estabilidade do programa, assim como o desempenho computacional e o tempo de execução.

Os testes foram realizados com o auxílio do *gerador de carga* disponibilizado no *moodle* da plataforma. No caso, esse algoritmo foi utilizado para tomar entradas mais complexas, com mais participantes e rodadas. A seguir, estão os comandos utilizados:

./geracarga -o entrada.txt -r 40 -j 12 -v 2

(entrada com 40 rodadas, média de 12 jogadores por rodada, com a variância de 2 jogadores)

***.**/geracarga -o entrada.txt -r 140 -j 12 -v 2*

(entrada com 140 rodadas, média de 12 jogadores por rodada, com a variância de 2 jogadores)

***.**/geracarga -o entrada.txt -r 240 -j 12 -v 2*

(entrada com 240 rodadas, média de 12 jogadores por rodada, com a variância de 2 jogadores)

Esses testes auxiliaram na determinação do custo computacional do programa, assim como o seu tempo de execução.

Os testes menores foram realizados com base no exemplo disponibilizado no *moodle*. Nesse caso, um número menor, e mais concentrado de rodadas foram testadas, verificando se a determinação das combinações, vencedores, valor recebido pelos vencedores e os vencedores estavam corretos, bem como o valor de cada participante ao final da partida.

Quanto maior o desafio, mais necessário é um controle em relação à sua estabilidade e segurança de suas respostas. Por esse motivo, foram realizados múltiplos testes, possibilitando assegurar a qualidade do algoritmo.

5.1. Tempo de Execução:

A análise do Tempo de Execução do algoritmo implementado teve como base a variação no número de rodadas. No caso, as rodadas variaram de 200 até 900, de 200 em 200 - *tirando das 800 rodadas para 900 rodadas, em que a variação foi de 100 rodadas* - e a conclusão já era esperada:

Como já foi citado anteriormente, o número de rodadas junto ao número de jogadores da partida - e de cada rodada - é o que mais afeta o programa, tanto em desempenho quanto em tempo de execução. No caso, como o número de participantes será inserido na lista, quanto mais participantes houver, maior o número de inserções - e instanciação de jogadores, que é bastante custoso, como já foi abordado - e o mesmo é válido para o número de rodadas: quanto maior, mais esse processo será realizado. Na figura a seguir, está explícito a variação do tempo de execução a partir das variações:

Figura 3. Variação no tempo de execução

Logo, ficou evidente que o tempo de execução é relativo à entrada. Entradas mais carregadas e complexas acarretam em um aumento no tempo de execução do programa.

5.2. Verificação de integridade:

Os testes de verificação de integridade do programa seguiram a entrada padrão disponibilizada no enunciado do problema:

3 1000
5 50
Giovanni 100 6O 3P 10E 11O 1O
John 200 3P 4E 3E 13C 13O
Thiago 100 12O 7P 12C 1O 13C
Gisele 300 12E 10C 11C 9C 13E
Wagner 50 5P 12P 5E 2E 1P
2 50
Wagner 200 2P 13E 9E 12C 2O
Gisele 350 11P 9P 2E 6E 4P
3 100
Thiago 250 1O 4P 1E 3O 8O
Gisele 100 9C 8C 8C 2C 6C
Giovanni 150 4P 12P 8E 12E 2P

No caso, essas entradas permitiram o teste das condições impostas pelo programa, a fim de garantir a saída adequada. Essa garantia foi assegurada por meio de condicionais, como já citado anteriormente. Fica evidente a importância dessa verificação de integridade, e ela é expressa através da garantia na saída correta dentro do programa, minimizando erros e bugs.

6. Análise Experimental:

Agora que os resultados já foram apresentados, faremos a análise. Para facilitar a compreensão, iremos separá-la em subseções.

6.1. Desempenho Computacional:

O desempenho computacional é bastante afetado pela forma que o programa é implementado. Funções otimizadas, ou seja, com complexidades preferenciais, como $O(\log n)$ oferecem um melhor desempenho computacional ao programa. Contudo, como citado anteriormente, muitas funções dentro do programa possuem uma complexidade de ordem $O(n^2)$, o que significa que, computacionalmente, dependendo do número de operações a serem realizadas, o programa pode ser bastante custoso em termos de processamento.

A métrica utilizada levou em consideração as análises de complexidades das funções citadas nessa documentação e para análise, foram utilizados arquivos de textos de entradas

geradas aleatoriamente, através do *Gerador de Carga* disponibilizado no *Moodle* da disciplina.

O algoritmo leva em consideração basicamente uma coisa: O arquivo de entrada. Entretanto, o volume de dados dentro desse arquivo é o que irá afetar o desempenho computacional do programa. Portanto, é fundamental enfatizar: As entradas foram geradas aleatoriamente, de modo que as entradas utilizadas não afetassem nas operações realizadas, ou seja, evitando que houvesse algum tipo de manipulação. Contudo, é fato que, as operações podem ser mais simples, ou mais complexas, dependendo das entradas. Por exemplo, uma entrada com 4 rodadas terá um custo computacional muito menor do que uma entrada com 40 e o mesmo é válido para o número de jogadores dentro das rodadas - *note que há um limite entre 2 a 12 jogadores médios por rodada*. Ainda que geradas aleatoriamente, as entradas do programa afetam seu desempenho.

A análise do gprof demonstrou o esperado: Dentre todas as funções implementadas,a mais custosa é também a mais complexa: A leitura de Arquivos.

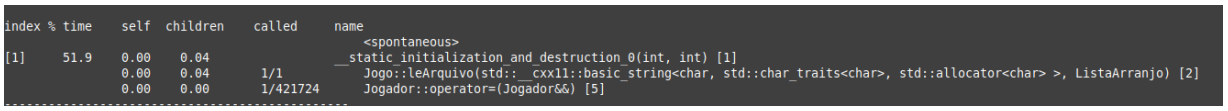


Figura 3. Custo computacional da função *leArquivo*

Na verdade, esse custo ***apenas*** para a leitura de arquivos não é esperado, uma vez que esse tende a ser um processo não tão custoso. Contudo, nesse cenário, a função *leArquivo* possui chamadas para outras também muito complexas - *como a análise de cada rodada, em que há uma nova leitura de arquivos* - portanto, isso explica o alto custo computacional oferecido.

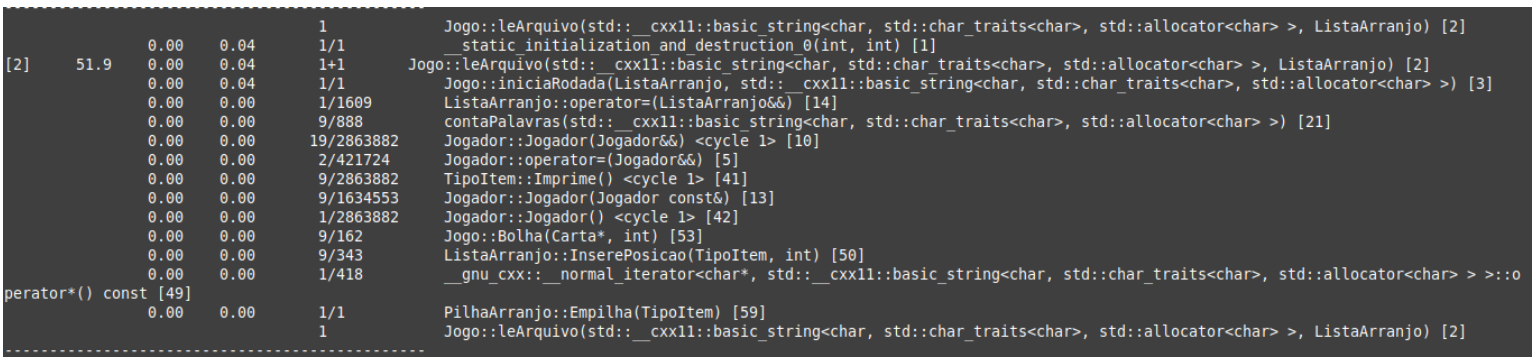
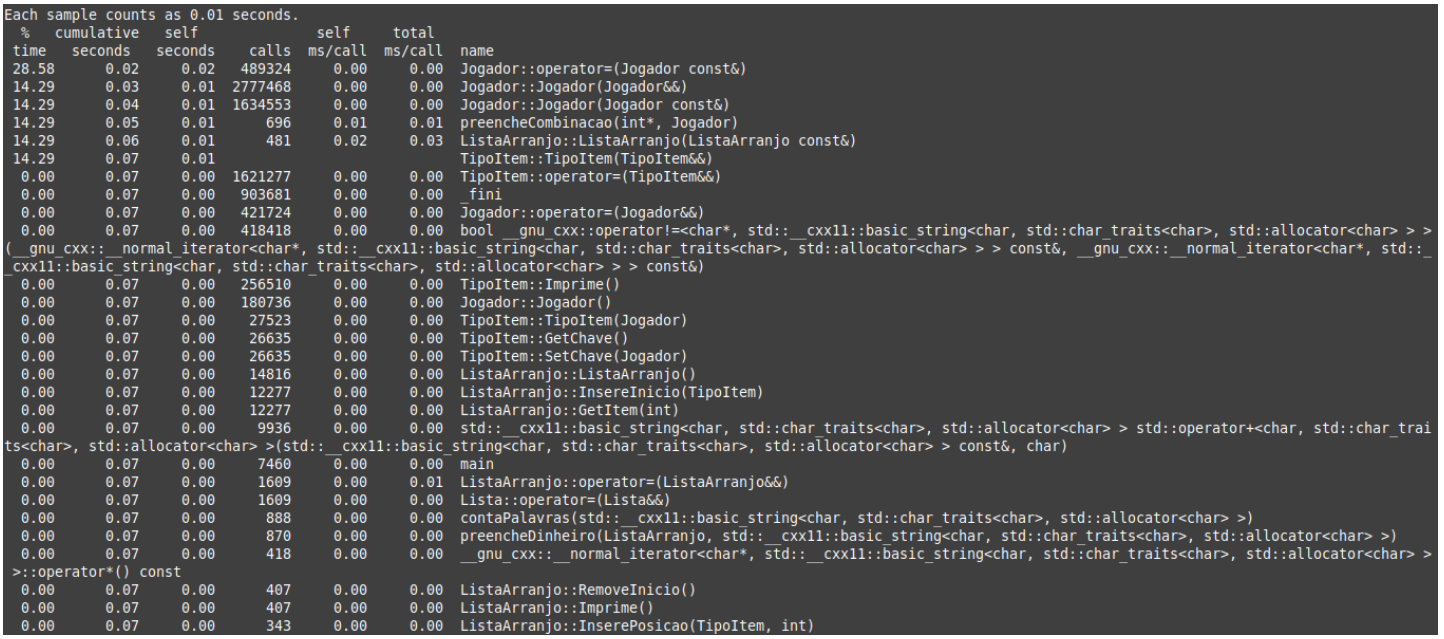


Figura 4. Detalhamento da função *leArquivo*

Na *figura 4* percebe-se que o que foi dito anteriormente: O alto número de funções dentro da leitura de arquivos é o que promove o alto custo computacional da função.

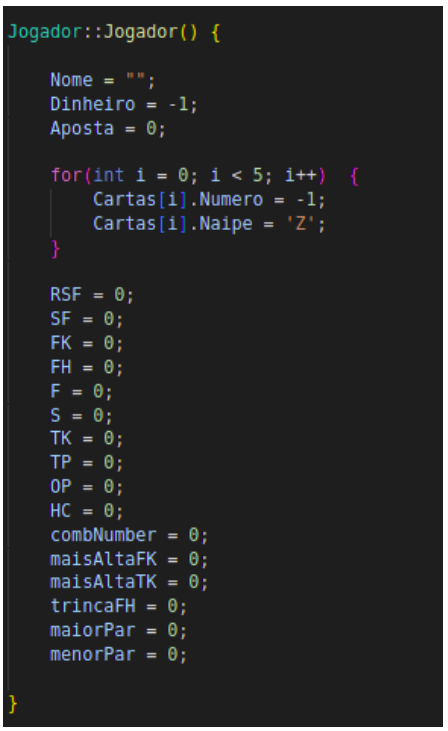
Aqui é importante fazer um parênteses para a análise dos construtores de algumas classes, no caso, de Jogador e Lista.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 28.58      0.02     0.02   489324     0.00     0.00  Jogador::operator=(Jogador const&)
 14.29      0.03     0.01  2777468     0.00     0.00  Jogador::Jogador(Jogador&&)
 14.29      0.04     0.01  1634553     0.00     0.00  Jogador::Jogador(Jogador const&)
 14.29      0.05     0.01      696     0.01     0.01  preencheCombinacao(int*, Jogador)
 14.29      0.06     0.01      481     0.02     0.03  ListaArranjo::ListaArranjo(ListaArranjo const&)
 14.29      0.07     0.01                             TipoItem::TipoItem(TipoItem&&)
  0.00      0.07     0.00  1621277     0.00     0.00  TipoItem::operator=(TipoItem&&)
  0.00      0.07     0.00   903681     0.00     0.00  _fini
  0.00      0.07     0.00   421724     0.00     0.00  Jogador::operator=(Jogador&&)
  0.00      0.07     0.00   418418     0.00     0.00  bool __gnu_cxx::operator!=<char*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >
(__gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > const&, __gnu_cxx::__normal_iterator<char*, std::_
_cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > const&)
  0.00      0.07     0.00   256510     0.00     0.00  TipoItem::Imprime()
  0.00      0.07     0.00   180736     0.00     0.00  Jogador::Jogador()
  0.00      0.07     0.00    27523     0.00     0.00  TipoItem::TipoItem(Jogador)
  0.00      0.07     0.00    26635     0.00     0.00  TipoItem::GetChave()
  0.00      0.07     0.00    26635     0.00     0.00  TipoItem::SetChave(Jogador)
  0.00      0.07     0.00    14816     0.00     0.00  ListaArranjo::ListaArranjo()
  0.00      0.07     0.00    12277     0.00     0.00  ListaArranjo::InsereInicio(TipoItem)
  0.00      0.07     0.00    12277     0.00     0.00  ListaArranjo::GetItem(int)
  0.00      0.07     0.00     9936     0.00     0.00  std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > std::operator+<char, std::char_trai
ts<char>, std::allocator<char> >(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&, char)
  0.00      0.07     0.00     7460     0.00     0.00  main
  0.00      0.07     0.00     1609     0.00     0.01  ListaArranjo::operator=(ListaArranjo&&)
  0.00      0.07     0.00     1609     0.00     0.00  Lista::operator=(Lista&&)
  0.00      0.07     0.00      888     0.00     0.00  contaPalavras(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
  0.00      0.07     0.00      870     0.00     0.00  preencheDinheiro(ListaArranjo, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
  0.00      0.07     0.00      418     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
>::operator*() const
  0.00      0.07     0.00      407     0.00     0.00  ListaArranjo::RemoveInicio()
  0.00      0.07     0.00      407     0.00     0.00  ListaArranjo::Imprime()
  0.00      0.07     0.00      343     0.00     0.00  ListaArranjo::InserePosicao(TipoItem, int)
```

Figura 5.

Percebe-se, pela *figura 5, o enorme* custo computacional do construtor de *Jogador*. Isso já era esperado, uma vez que dentro dessa função, existem laços e uma série de atribuições. Além disso, o que explica o alto custo dessa função, é o fato dela ser chamada em múltiplos momentos, na criação de jogadores auxiliares, na criação dos jogadores participantes da rodada e do jogo, etc. Ou seja, além de ser uma função cara, como percebe-se na figura a seguir, a sua recorrência no código contribui também para o seu alto ‘preço’.



```
Jogador::Jogador() {

    Nome = "";
    Dinheiro = -1;
    Aposta = 0;

    for(int i = 0; i < 5; i++)  {
        Cartas[i].Numero = -1;
        Cartas[i].Naipe = 'Z';
    }

    RSF = 0;
    SF = 0;
    FK = 0;
    FH = 0;
    F = 0;
    S = 0;
    TK = 0;
    TP = 0;
    OP = 0;
    HC = 0;
    combNumber = 0;
    maisAltaFK = 0;
    maisAltaTK = 0;
    trincaFH = 0;
    maiorPar = 0;
    menorPar = 0;

}
```

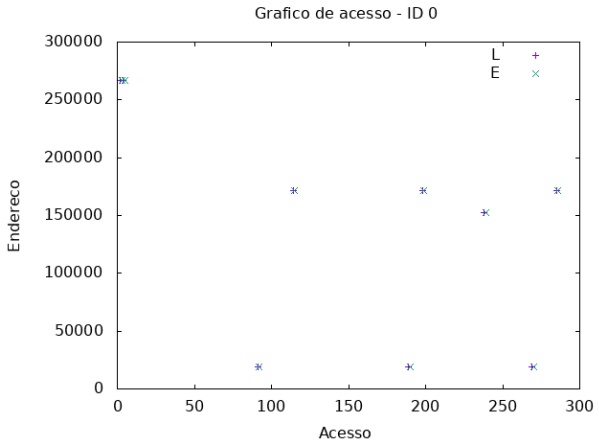
Figura 6. Construtor da Classe Jogador

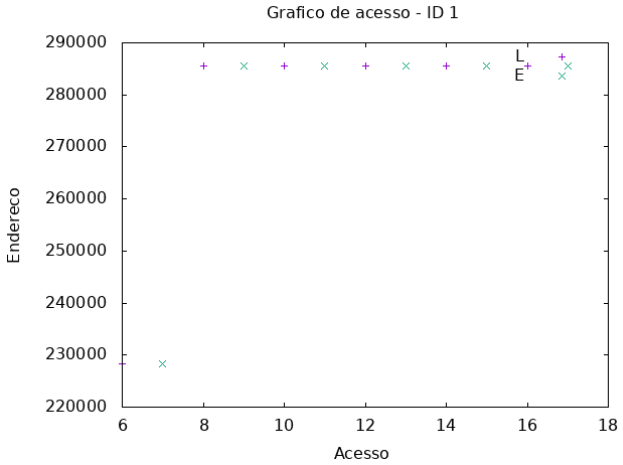
A análise do alto custo computacional do construtor da Classe Lista é também intuitivo: Como as listas são compostas por jogadores - são os *tipoItem* das listas - em múltiplos momentos é necessário instanciar jogadores dentro das listas. Logo, como visto anteriormente, o alto custo dessa instanciação contribui para o aumento do custo oferecido pelo método construtor das listas.

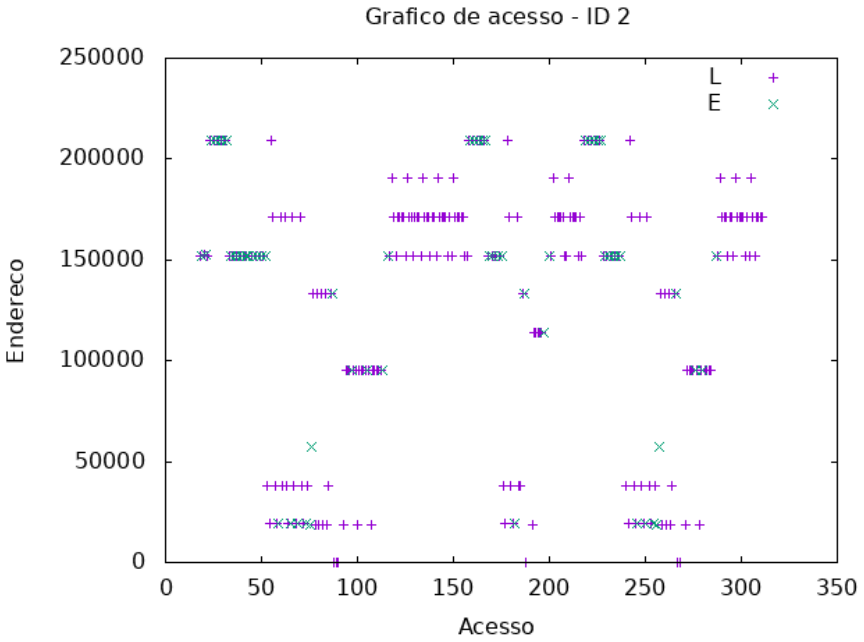
Portanto, os resultados obtidos condizem com a análise de complexidade abordada nessa documentação. Como dito anteriormente, algumas funções não possuem uma complexidade pouco custosa computacionalmente, entretanto, os algoritmos utilizados buscam otimizar o desempenho, junto a simplicidade. A melhora desse programa, em termos de desempenho computacional poderia vir principalmente de dois fatores: Um algoritmo de ordenação menos complexo, ou seja, substituir o método *Bolha* por um método "*mais eficiente"*, como o *quicksort*. Além disso, substituir uma Lista Estática (Arranjo), pelo uso de ponteiros, como é o caso de uma Lista Encadeada, por exemplo. Em ambos os casos, o desempenho computacional poderia ser otimizado, mas não necessariamente haveriam ganhos em todos os âmbitos. Por exemplo, o código se tornaria consideravelmente mais complexo.

6.2. Padrão de acesso à Memória e Localidade de Referência:

Para a análise de acesso à Memória e localidade de referência foi utilizada a entrada padrão, disponibilizada no enunciado deste TP. São 5 rodadas, com um total de 5 participantes. Essa escolha busca encontrar um intervalo entre um programa de tamanho razoável, oferecendo desempenho suficiente para análise de localidade.





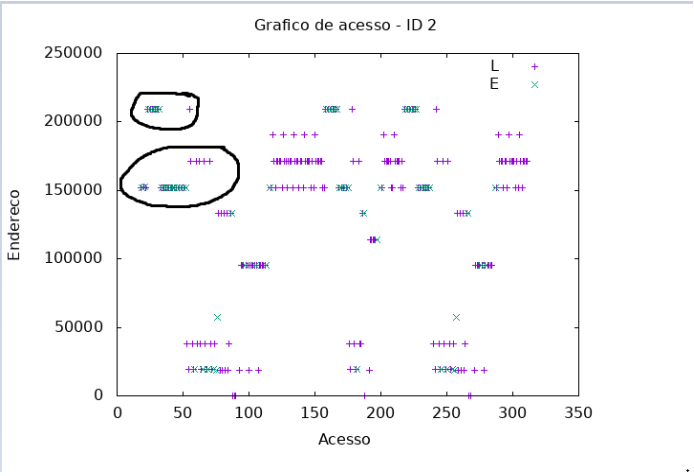


O mapa de acesso de ID 0 demonstra a primeira sequência de alocações de memória, ou seja, é a primeira alocação de Lista Arranjo dos jogadores participantes. Esses acessos estão bastantes dispersos em memória, ocorrendo a leitura e escrita praticamente simultaneamente. Essa dispersão definitivamente não é positiva em termos de localidade de referência.

O mapa de acesso de ID1 é bastante diferente do anterior, nesse caso, trata-se do momento de leitura do arquivo de entrada. Sendo assim, temos inicialmente a leitura das

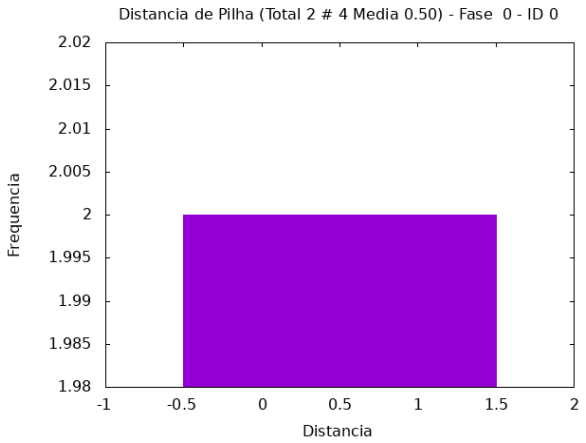
linhas, e logo em sequência, a escrita dos dados lidos de cada linha individualmente. Isso ocorre pela escolha do método de leitura: Cada linha é lida por vez, portanto, temos uma certa cadeia de eventos. Percebe-se que a localidade de referência nesse mapa é bem melhor, uma vez que os dados são escritos, ou lidos, em endereços próximos.

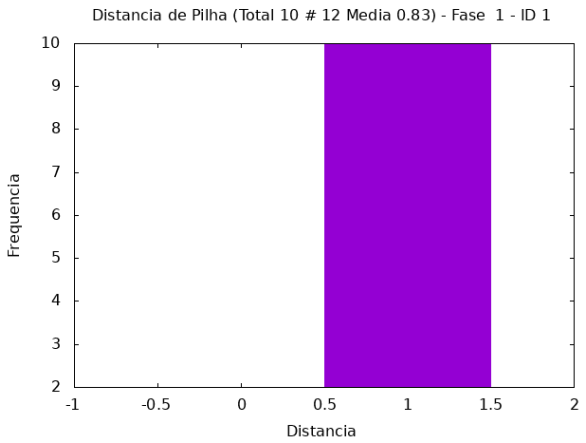
Por fim, temos o gráfico de acesso de ID 2, ou seja, momento de ***CADA RODADA DA PARTIDA,*** e a conclusão é um gráfico um pouco mais robusto, que devemos analisar com mais atenção. Em primeiro lugar, temos o processo de leitura do arquivo, novamente, mas agora feito rodada por rodada. Sendo assim, temos a leitura de cada um dos jogadores, e a escrita na lista de jogadores da rodada. Esse processo pode ser percebido pelos momentos a seguir do gráfico:

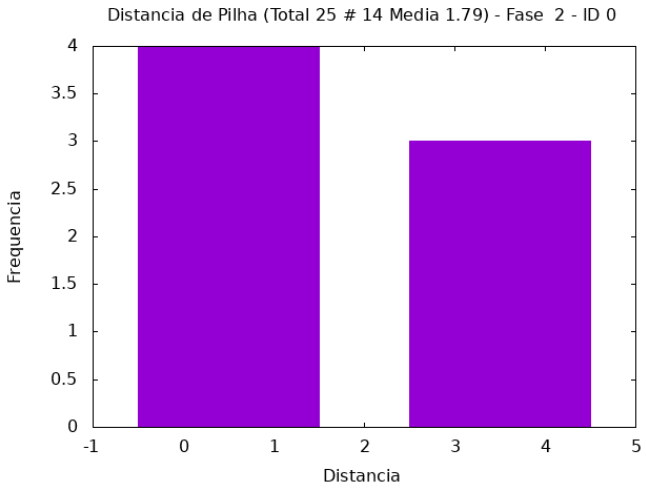


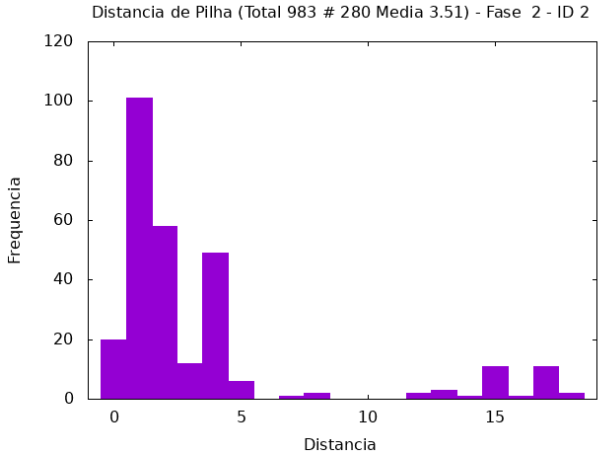
A sequência de eventos envolve o processo de verificação de vencedores, e, no caso de empate, a verificação de empate. No gráfico, isso corresponde aos acessos sequentes aos apresentados anteriormente. Percebe-se que os acessos de leitura estão bastante concentrados, entretanto, em memória, estão bastante dispersos. Isso significa que as leituras são realizadas em sequência, contudo, em posições muito distintas da memória. De certa forma, há o lado positivo: a leitura é feita em posições similares da memória, e o lado negativo: Essa leitura está bastante dispersa ao longo da memória.

Agora, para uma análise mais ampla, visando a quantificação dos acessos, faremos uma análise das distâncias dessa mesma operação:







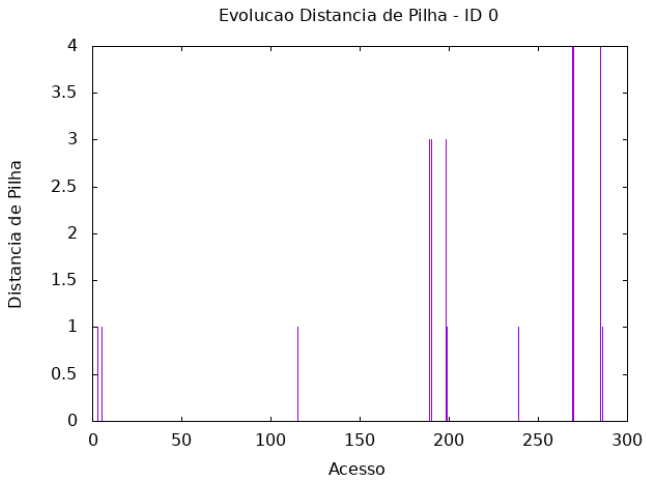


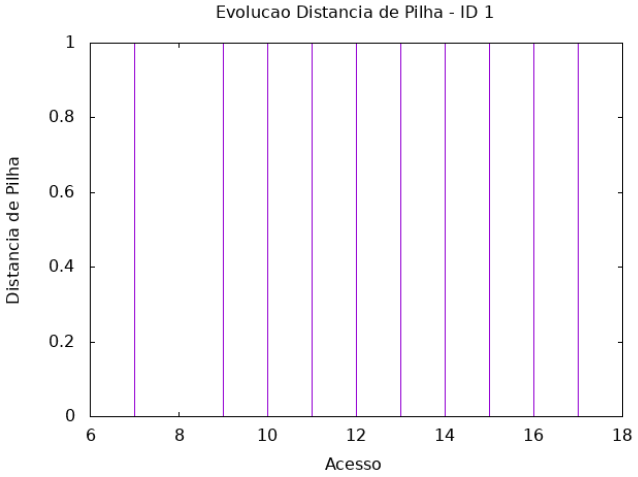
Novamente, esses gráficos expressam as 3 fases determinadas ao longo do processo: Alocação da lista inicial de jogadores da partida, leitura do arquivo, e divisão das rodadas. O primeiro gráfico, representa exatamente a primeira fase. Percebe-se que ele é bastante simples, e compacto: A sua frequência indica o número de listas alocadas: A de jogadores, e a de vencedores. Além disso, a distância de pilha demonstra o tamanho inicial dessas listas. Nesse caso, a distância de pilha é ***extremamente baixa*.**

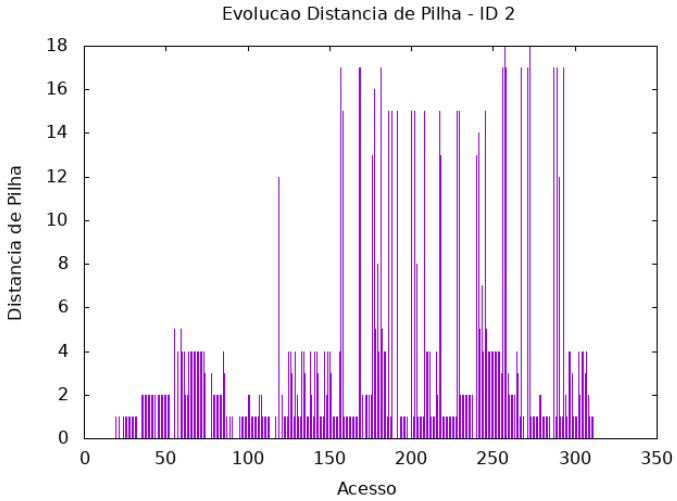
O gráfico seguinte representa o momento de leitura de arquivos. Novamente, ele possui uma distância de pilha bastante similar ao anterior, entretanto, a frequência é bastante distinta, ou seja, a distância de pilha é ainda ***bastante baixa***, contudo, houve um crescimento de 500% em relação à fase anterior. O fato é que a leitura é realizada novamente, para uma lista distinta. Uma vez que não foram utilizados ponteiros, temos duas listas de jogadores: Uma no main, outra na função de leitura de arquivos. Do ponto de vista computacional, isso é um desperdício, uma vez que é possível a economia de memória, entretanto, do ponto de vista da simplicidade no código, houve um ganho.

Os gráficos seguintes representam a leitura de cada rodada do arquivo de entrada. Nota-se que esse é o momento mais complexo do ponto de vista computacional. É aqui que a maior parte das condicionais e loops do programa são acionados. Logo, era esperado que a distância de pilha fosse um pouco distinta nesse momento. Percebe-se que a fase mais custosa para a memória possui distância de pilha igual à 983. Apesar de parecer muito, em relação ao numeroso conjunto de funções, isso é positivo, uma vez que torna o algoritmo mais dinâmico, oferecendo menos custos computacionais. Na fase 0, cada elemento é acessado individualmente, são exatamente os 25 acessos iniciais. Em seguida, durante a inicialização, toda a lista de jogadores da rodada é lida, oferecendo uma maior distância de pilha. Durante a fase seguinte, novamente, cada elemento é acessado individualmente, e em seguida há o preenchimento da lista, o que é representado pela maior distância de pilha.

Por fim, apenas citamos os gráficos de evolução de pilha, que confirmam o que foi dito anteriormente:







7. Conclusão:

O programa implementado envolveu o desenvolvimento de um algoritmo capaz de receber entradas equivalentes a um jogo de Poker e retornar uma saída, informando os vencedores de cada rodada, junto com a quantia de dinheiro restante para cada participante. Este trabalho definitivamente instigou o desenvolvimento de diversas técnicas que envolvem a programação. Esse desenvolvimento extrapolou apenas o desenvolvimento em si, mas também atingiu níveis como abstração, percepção da realidade, pesquisa, solução de problemas e o planejamento de software.

Sem dúvida, o criar um programa com um certo nível de complexidade como esse, partindo do zero, exige um certo planejamento, ao passo que seja possível uma modularização de código mais eficiente, permitindo atingir melhores resultados mais a frente, tanto em relação ao aproveitamento das Classes e Objetos, quanto aos resultados finais em si. Nesse sentido, a abstração também é desenvolvida, uma vez que é importante modularizar o código, a medida que, a implementação de alguns métodos e procedimentos não fiquem evidentes à um possível usuário final, mas apenas interesse o seu uso - como ocorre nos métodos e procedimentos da Lista Arranjo implementada. Entretanto, em um sentido mais amplo, a abstração é fundamental para conseguir capturar as nuances de um jogo de poker da realidade e transformá-las em Classes, TADs, métodos e atributos dentro do código.

Além disso, ainda na implementação, esse trabalho proporcionou o desenvolvimento do uso de Estruturas de Dados de forma mais concreta, ou seja, exigindo a implementação dessas estruturas. No caso, a Estrutura Utilizada foi a Lista, entretanto, a escolha desse tipo exigiu o conhecimento de todos os demais. Portanto, os ganhos durante o desenvolvimento são incontáveis.

Por fim, houveram ganhos '*extras'* ao longo do desenvolvimento. Em um projeto amplo é esperado o surgimento de *bugs* e erros sucessivos. A fim de contornar esses problemas, é importante a pesquisa e a depuração. Logo, esse programa ainda proporcionou o desenvolvimento da capacidade de encontrar os problemas e solucioná-los, o que é fenomenal, dado que essa é uma competência fundamental dentro da Ciência da Computação.

Sendo assim, a implementação do TP1 foi fundamental para o meu crescimento, tanto profissional quanto pedagógico. Lidar com novas ferramentas ofereceram uma oportunidade para aumentar meu conhecimento, e esse aprendizado me instigou a correr atrás de novos instrumentos, o que aumentou o meu repertório. Portanto, a implementação do TP1 ofereceu grande margem de aprendizado, em múltiplas competências, como citado anteriormente.

8. Bibliografia:

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Java e C++:˜ Capítulo 3: Estruturas de Dados Básicas. Editora Cengage.

9. Instruções para compilação e execução:

A compilação do programa é "automática", via Makefile, implementado para esse programa. O comando "*make all"* deve ser realizado via terminal na pasta TP1. Esse comando irá compilar o programa, gerando o executável e todos os arquivos **.out* necessários para o funcionamento. Caso seja necessário "*reiniciar"* a compilação, o comando "*make clean"* apaga todos os **.o* e **.out*, permitindo uma recompilação.

Após compilar, o programa implementado terá o seu '*executável'* disponível na pasta bin. Esse executável terá nome *poker.out* e **nessa pasta deve estar contido o arquivo de entrada** que com **NOME** *"entrada.txt"*.

ATENÇÃO: *Caso nenhum arquivo "entrada.txt" esteja presente na pasta 'bin', a execução do programa será interrompida, e uma mensagem de erro irá indicar o problema.*

A execução do programa é bastante simples, e a saída será sempre padrão, no arquivo "*saida.txt",* também presente na pasta bin. No caso, não existem entradas via linha de comando, a própria execução do programa via terminal - através do comando *./tp.out* - já gera a saída esperada em um novo arquivo.