Trabalho Prático 1
Poker Face

Vitor Emanuel Ferreira Vital
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

vitorvital@dcc.ufmg.br

Introdução

O problema proposto foi implementar um programa em C/C++ utilizando tipos abstratos de dados, alocação dinâmica de variáveis, processos simples de ordenação, dentre outros métodos, para um jogo de pôquer.

Para um conjunto de jogadores, suas apostas e suas cartas, deveríamos classificá-los de acordo com sua pontuação, cedendo o(s) ganhador(es) e o respectivo valor ganho, além disso, ao final do jogo, devemos elencar os jogadores por suas quantias finais. A classificação, vitória e quantia ganha respeitam as regras de um pôquer convencional.

A partir do programa resultante, devemos fazer uma análise da complexidade de tempo e de espaço dos procedimentos implementados.

Método

A implementação da resolução do problema baseou-se na utilização de classes, nesse sentido, existem 4 relevantes tipos de dados representados para resolução do problema, sendo elas Carta, Jogadores, Rodada e Partida. Cada jogo possui uma partida com n rodadas, cada rodada é composta de m jogadores possuindo cada um deles 5 cartas distintas.

Apesar de ter um custo maior que a implementação estática, para fins didáticos, foram utilizadas listas encadeadas para representarmos rodadas, jogadores e cartas.

O comportamento das rodadas é semelhante a fila, ou seja, uma nova é inserida posteriormente a última rodada jogada. Para haver tal inserção foi criada uma classe auxiliar nomeada ListaRodadas, no qual as rodadas são ordenadas, havendo a inserção de uma nova rodada se as especificações dela forem válidas, ou seja, se o valor do pingo, apostas e jogadores participantes forem válidos.

Concomitante as rodadas, há a criação de uma lista de jogadores a partir de um TAD nomeado ListaJogadores, nesse caso, os jogadores são inseridos de maneira ordenada por sua pontuação (foram atribuídos valores de 100 em 100 para cada classificação possível, sendo *One Pair* a pontuação 100 e *Royal Straight Flush* 1000) , para isso suas cartas são analisadas e obtém-se a pontuação do jogador em posse delas. Os jogadores só são adicionados a uma nova rodada caso tenham participado de rodadas anteriores.

Em casos de empate de pontuação realizam-se os testes para cada caso, ou seja, teste da maior quádrupla, tripla, dupla, maior carta, etc. Entretanto, em caso de empates em todos esses casos os jogadores são inseridos um posterior ao outro sem distinção, adicionando mais de um jogador aos vencedores.

Por haver uma ordenação prévia, como citado, foi utilizado o Método Inserção, portanto, a lista de jogadores é percorrida continuamente até obter-se a posição na qual o jogador deve ser inserido.

Da mesma forma, houve a criação de uma lista de cartas a partir de uma classe chamada ListaCartas, nessa, assim como a ListaJogadores, há a utilização do Método Inserção, nesse caso, apenas o numeral é considerado para ordenação, ou seja, cartas com mesmo valor são colocadas em seguida. Tal implementação permite o reconhecimento de forma mais simples da classificação das cartas.

É importante salientar que, a classe rodada possui uma referência à ListaRodadas, cada Rodada possui uma referência à ListaJogadores (Jogadores participantes da rodada) e cada Jogador possui uma referência a ListaCartas (Cartas utilizadas em determinada rodada).

Análise de complexidade

A partir de cada entrada faremos a inserção das rodadas que terá um custo O(1), pois trata-se da inserção em uma fila dinâmica com referência ao último elemento, que no caso é a última rodada jogada. Entretanto, cada jogada faz referência a um número n de jogadores, que são os participantes daquela jogada. Um jogador é ordenado de maneira decrescente de acordo com sua classificação, portanto, seu custo de inserção é variável possuindo um melhor caso quando possuir pontuação maior que demais e o pior caso quando tiver a menor pontuação, por isso, sua inserção é de O(n).

Assim com as rodadas, os jogadores também referenciam outra classe, sendo essas a Cartas, nesse caso, se número de cartas de um jogador tendesse a infinito teríamos um complexidade assintótica O(m) para inserção de novas cartas no pior caso, visto que são ordenadas de forma crescente, ou seja, possui custo variável, dessa forma, teria custo O(1) caso fosse menos que as outras. Entretanto, cada jogador tem 5 cartas, como a entrada é pequena podemos considerar que o custo de ordenação de cartas é O(1) e não impacta significativamente o custo de tempo e espaço do programa.

Outra operação relevante é a que calcula a classificação de cada jogador. Nesse caso, é necessário realizar, no pior caso, 10 testes, referentes às possíveis classificações do pôquer. Nesse caso, temos para cada teste por volta de 5 comparações referentes ao número de cartas, podendo haver até 25 comparações, nesse sentido, se considerarmos que cada teste realiza 25 comparações teremos, em 10 testes, 250 comparações. Se na pior situação temos que realizar todas elas para todos os jogadores teremos para cada rodada 250*n comparações, que é o mesmo que uma complexidade assintótica O(n). Portanto o custo das operações mais relevantes do programa são na teoria polinomiais, logo, ao realizarmos as análises experimentais espera-se obter um custo linear.

Estratégias de robustez

Os mecanismos de programação defensiva e tolerância a falhas são de extrema importância na realização de programas, isso porque evita que o usuário conceda entradas incorretas, utilize dessas falhas para danificar de alguma forma o programa, ou obtenha resultados inesperados na execução das funções do programa.

Nessa etapa foram utilizadas as funções __erroassert e __avisoassert para informar ao usuário problemas durante a execução do programa ou por introdução de valores incorretos.

Nome do arquivo de entrada

Ao inicializar o programa é necessário passar o arquivo no qual os dados utilizados pelo programa estão, dessa forma, é necessário verificar se existe a passagem do nome do arquivo para sua futura abertura.

Abertura dos arquivos de entrada e saída

Na implementação desse programa foi solicitado pegar os dados da partida de um arquivo de entrada, além de gravar os resultados obtidos em um arquivo de saída, ou seja, a utilização de arquivos é de extrema importância no decorrer do programa. Entretanto, podem haver falhas na

abertura desses, o que impossibilita a execução do programa. Portanto, é de suma importância verificar se os arquivos foram abertos corretamente para utilização no programa.

Inicializações

Durante a obtenção dos dados dos jogos algumas condições devem ser satisfeitas, dentre elas o valor inicial dos jogadores deve ser um número positivo, para que exista um jogos; o número de rodadas deve ser um valor não negativo; o número de jogadores deve ser um valor não negativo; o pingo deve respeitar a condição de ser no mínimo 50; o nome e as cartas devem ser válidos, evitando problemas durante o jogo.

Adição de novos membros

Na fase de adição de novos membros, sendo ele Rodadas, Jogadores e Cartas, é importante que a nova entidade seja inicializada corretamente, para isso, utilizou-se de avisos, para que o valor adicionado seja um valor válido e caso exista algum problema a rodada seja desconsiderada.

Acesso aos membros

No decorrer do programa diversas vezes acessam-se pontos específicos das listas, seja para adicionar novos membros, ou mesmo fazer alguma comparação, nesse sentido é importante verificar se os valores retornados são válidos evitando problemas futuros.

Saldo do jogador

A cada rodada os jogadores devem realizar algum pagamento, seja colocando o pingo ou apostando, entretanto pode ocorrer do jogador não possuir o valor para realização de tais ações, o que não é fator relevante para acabar uma partida, porém é necessário invalidar a rodada.

Nesse caso, para que o usuário saiba o porquê de uma rodada ter sido invalidada foram colocados avisos quando ocorrer tal situação.

Apostas inválidas

Da mesma forma que ocorre com o saldo do jogador, as apostas também devem ser válidas, ou seja, possuírem valores positivos e múltiplos de 50, dessa forma, apostas incorretas invalidam a rodada, porém não invalidam o jogo. Portanto, é importante que haja o alerta ao usuário sobre o ocorrido, evitando saídas inesperadas.

Adição e remoção do dinheiro

A fim de evitar utilizações indevidas das funções de adição e remoção do dinheiro, são válidos apenas valores não negativos adicionando-o ou removendo do montante total do jogador.

Análise experimental

Grande parte do custo de memória do programa é relativo à inserção dos elementos trabalhados, por isso faremos uma análise dessas inserções.

Inserções

As inserções das Cartas e dos Jogadores, por serem organizados em uma lista dinâmica de forma ordenada, possuem melhor e pior caso. De início, o melhor caso tanto para inserção das cartas quanto dos jogadores é O(1), nesse cenário, a carta é a menor, sendo inserida no início e em seu pior caso inserida no final, com custo O(n). Por haver um número de cartas igual a 5 o custo da inserção é relativamente baixo, entretanto estamos avaliando o algoritmo em geral, por isso consideramos que pode-se inserir n cartas e, nesse cenário, é relevante saber onde haverá tal inserção. Já no caso dos jogadores, o melhor caso é quando o jogador tem a melhor classificação, sendo inserido no início.

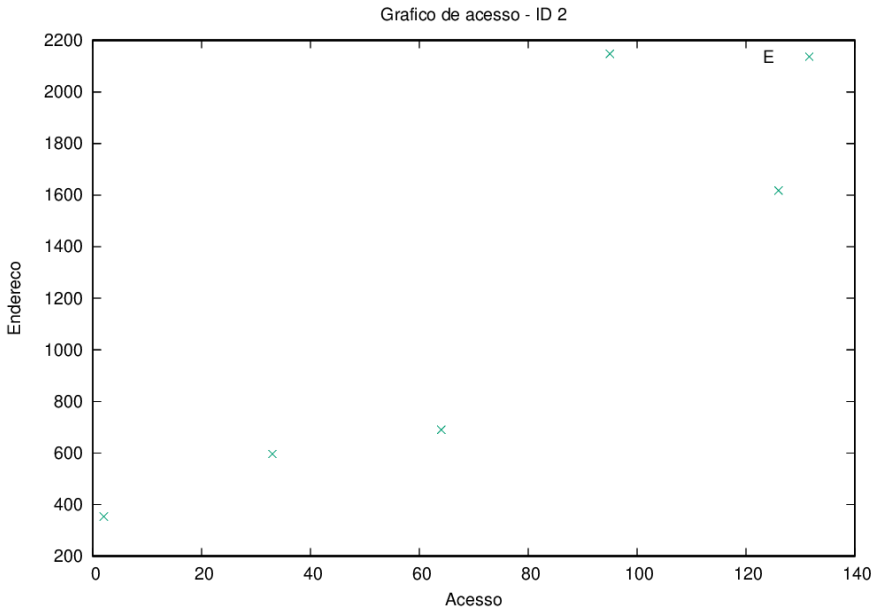
Porém, outros fatores influenciam a inserção do jogador, o que impede de possuir uma análise simples de sua inserção. Para sabermos a classificação do jogador é necessário percorrer as cartas dele. Novamente iremos considerar um número n de cartas. Para o melhor cenário percorremos a carta dele apenas uma vez e descobrimos que ele possui a melhor classificação, em pôquer a melhor classificação é o *Royal Straight Flush*, possuindo a ordem de complexidade de tempo igual a O(n).

Entretanto, no pior caso, ele além de verificar qual classificação ele possui e no pior caso ele será considerado termo-a-termo dez vezes, ele terá de comparar sua pontuação com todos os jogadores da rodada e inserido ao final, de forma simplificada, para m jogadores, haverá um custo O(m*n).

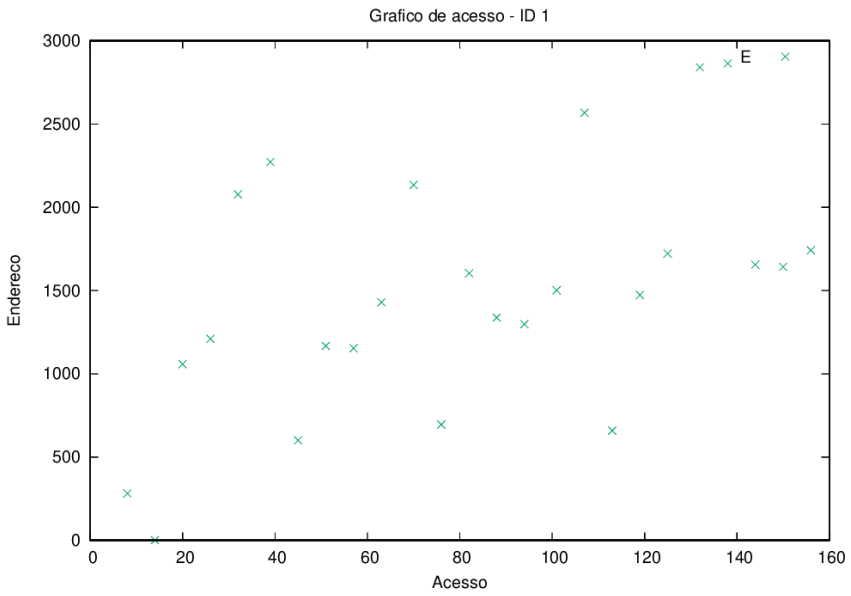
Por fim, a inserção de uma rodada haverá apenas o custo O(n), visto que ela sempre é inserida no final, semelhante a uma fila.

Pode ser visualizada na tabela abaixo os relativos mapas de acesso das inserções das Rodadas, Jogadores e Cartas. O endereço deles não possuem um padrão que permite uma visualização fácil de como é realizada, isso ocorre, pois ao inserir rodadas consecutivas, por exemplo inserimos entre elas m jogadores e n*m cartas, o que impede uma sequência uniforme entre elas.

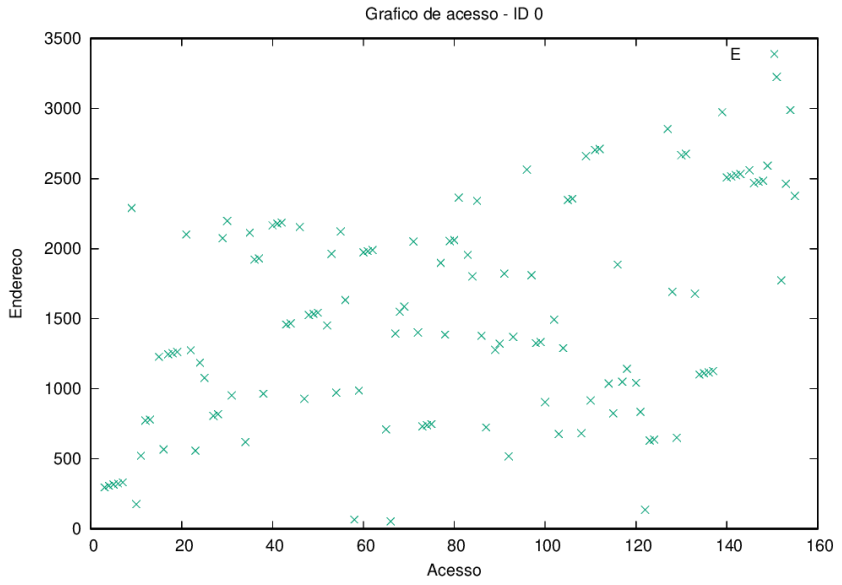
Inserção de Rodadas - (5 rodadas)



Inserção de Jogadores - (5 jogadores por rodada = 25)

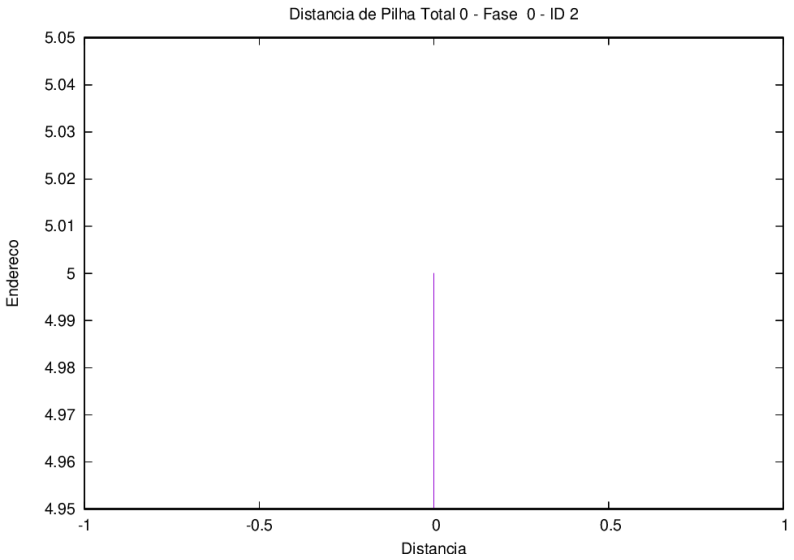


Inserção de Cartas (5 cartas por jogador por rodada = 125 cartas)



A partir do gráfico de cartas podemos ver que ocorrem 155 acessos, o que condiz com os acessos esperados, 125 cartas + 25 jogadores + 5 rodadas.

Já ao tentarmos realizar o histograma das inserções não obteremos sucesso, isso porque tanto a ordem quanto o local de acesso é aleatória, dependendo de como o usuário insere os dados. Por tal fato, não é possível haver uma inicialização prévia impedindo que haja um mapeamento da localização e posterior acesso. Obtemos assim o seguinte gráfico para todos os membros inseridos.



Complexidade de tempo

Como citado, grande parte do custo de tempo provém das inserções e chamadas de funções que garantem tal processo. Porém além disso, temos outras funções provindas de bibliotecas auxiliares que também gastam tempo considerável de execução.

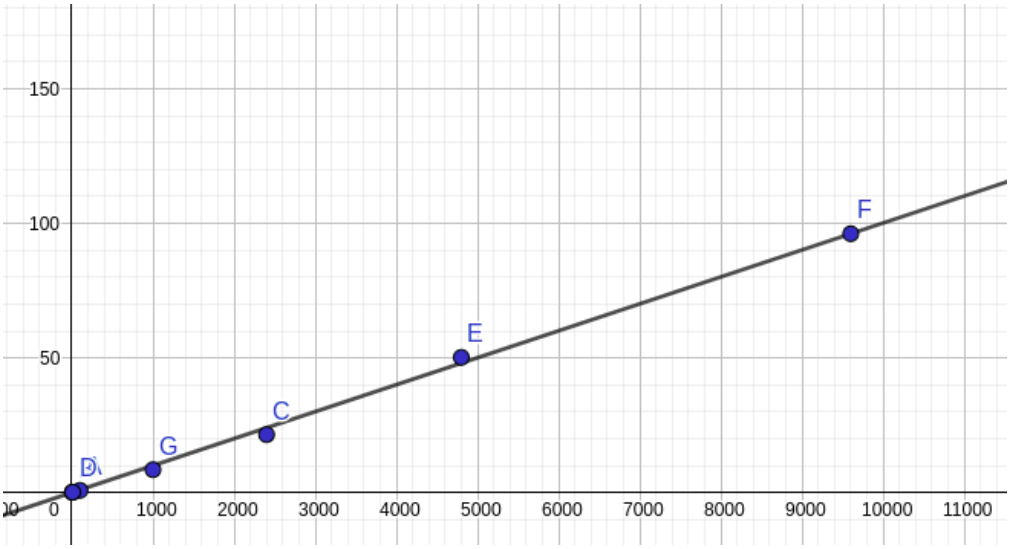
Podemos ver o tempo relativo para entradas com 9600, 4800, 2400, 1000, 100, 10 e 5 rodadas, todas elas possuem 5 jogadores como entrada e todos eles possuem as mesmas cartas em todas as rodadas. Podemos ver que, o programa chama funções distintas para um número de entradas

distintas, adicionando ainda mais ao tempo de execução do programa. Por exemplo, ao comparar 5 e 10 rodadas obtemos os seguintes resultados, respectivamente:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 14.29      0.01     0.01                             std::vector<char, std::allocator<char> >::end() const
 14.29      0.02     0.01                             std::_Vector_base<char, std::allocator<char> >::_M_allocate(unsigned long)
 14.29      0.03     0.01                             std::_Rb_tree<long, std::pair<long const, long>, std::_Select1st<std::pair<lo
 14.29      0.04     0.01                             char const* std::__miter_base<char const*>(char const*)
 14.29      0.05     0.01                             bool std::binary_search<__gnu_cxx::__normal_iterator<char const*, std::vector<
 14.29      0.06     0.01                             char* std::__uninitialized_copy_a<char const*, char*, char>(char const*, char
  7.14      0.07     0.01                             std::_Vector_base<char, std::allocator<char> >::_Vector_impl::_Vector_impl(st
  7.14      0.07     0.01                             std::remove_reference<std::__detail::_BracketMatcher<std::__cxx11::regex_trai
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 15.39      0.02     0.02                             std::__detail::_BracketMatcher<std::__cxx11::regex_traits<char>, false, false>
  7.69      0.03     0.01                             __gnu_cxx::__ops::_Iter_equals_val<std::__cxx11::basic_string<char, std::char
  7.69      0.04     0.01                             __gnu_cxx::__normal_iterator<char const*, std::vector<char, std::allocator<ch
  7.69      0.05     0.01                             bool __gnu_cxx::operator!=<std::pair<char, char> const*, std::vector<std::pai
  7.69      0.06     0.01                             std::vector<char, std::allocator<char> >::size() const
  7.69      0.07     0.01                             std::_Vector_base<std::__cxx11::basic_string<char, std::char_traits<char>, st
  7.69      0.08     0.01                             std::_Function_base::_Function_base()
  7.69      0.09     0.01                             std::allocator_traits<std::allocator<char> >::deallocate(std::allocator<char>&
  7.69      0.10     0.01                             void std::vector<std::pair<std::__cxx11::basic_string<char, std::char_traits<c
  7.69      0.11     0.01                             std::vector<char, std::allocator<char> >::data()
  7.69      0.12     0.01                             void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<c
  3.85      0.13     0.01                             std::deque<long, std::allocator<long> >::back()
  3.85      0.13     0.01                             std::deque<long, std::allocator<long> >::pop_back()
```

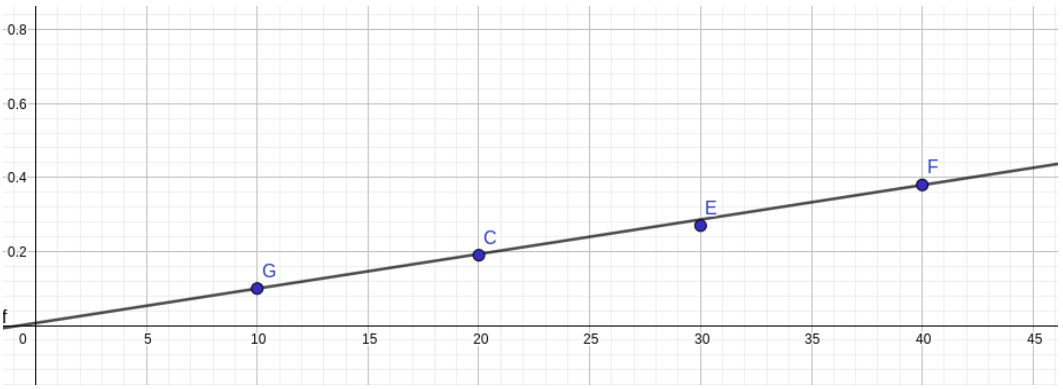
Podemos ver que o número de funções chamadas são bem maiores e que o custo de tempo dobrou para um número maior de rodadas. Porém como as entradas são pequenas o custo pode ser significativamente alterado por fatores externos. Para isso, foi realizado um teste com com diversas rodadas para vermos o comportamento de tempo para elas. Podemos ver abaixo o gráfico custo de tempo em segundos x tamanho da entrada, lembrando que todas as rodadas possuem 5 jogadores com as mesmas cartas e realizam as mesmas apostas e pingo.



Com base nos dados obtidos e nos gráficos a variação de rodadas, para uma mesma entrada de jogadores e cartas possui um custo O(n).

Entretanto, se alterarmos o número de jogadores por partida, para isso os jogadores podem possuir a mesma carta em mãos, mesmo que não seja algo válido em uma partida de pôquer, porém com o objetivo de fazer análise de complexidade, permitiremos durante o programa.

Nesse cenário, fizemos o teste para 5 rodadas com 10, 20, 30 e 40 jogadores distintos, com isso obtivemos o seguinte gráfico tamanho da entrada x custo de tempo em segundos:



Com base nos dados obtidos podemos perceber que tanto a adição de jogadores quanto as rodadas possuem um custo de tempo O(n).

Conclusão

Após a implementação do programa, pode-se notar que fomos propostos a cumprir duas etapas no trabalho. A primeira tinha como enfoque a implementação de um programa em linguagem C/C++. Já a segunda era utilizar programas para avaliação do desempenho e corretude do código, com base na análise do desempenho de tempo, do uso da memória e do funcionamento do algoritmo para diferentes entradas.

Com base nisso, conforme citado obteve-se um custo de tempo e memória linear, ou seja, se dobrarmos a entrada o tempo e memória gastos pelo programa também dobrava, comportamento esperado conforme a análise feita anteriormente.

A grande dificuldade encontrada na implementação foi na utilização conjunta de classes e de listas, além da falta de especificação inicial sobre o trabalho, gerando diversas ambiguidades.

Porém, com o desenvolver do código alguns fatores relevantes ficaram claros, além de conseguirmos aprimorar e encontrar problemas de soluções mais criativas para diferentes tipos de problemas.

Por fim, esse trabalho, deixou mais evidente a importância da utilização das ferramentas de análise de desempenho e memória de programas. Tal prática possibilitou fixar mais sobre utilização de alocações dinâmicas, utilizações de TAD's, listas encadeadas e estratégias de robustez. Além disso, permitiu diferenciar ainda mais a perspectiva de apenas programar para resolver problemas, para programar com o objetivo de encontrar a solução mais eficaz para os problemas, entendendo a implementação e seu comportamento na memória e em custo de tempo.

Bibliografia

www.stackoverflow.com

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Pascal e C

Cormen, T. Leiserson, C, Rivest, R., Stein, C.(2009) Introduction to Algorithms

Aulas e materiais disponibilizados no moodle UFMG.

Instruções para compilação e execução

Para compilar o programa basta que, na pasta TP, seja digitado make, assim será gerado na pasta ./bin o arquivo executável do programa. Com isso em mãos, basta que executem o arquivo junto ao nome do arquivo de entrada em formato .txt. Por exemplo, “./bin/run.out entrada.txt”.

Com isso será gerado o arquivo com a saída do programa “saida.txt” na pasta TP com os resultados obtidos de acordo com a entrada, junto a ele obteremos o arquivo “saidap”, esse arquivo contém os dados para a análise do analisamemlog.

Após tal execução é possível obter a saída do gprof utilizando a linha de comando “gprof ./bin/run.out gmon.out > saidagprof.txt”.