Algoritmos e Estruturas de Dados III

Trabalho Prático I

BigNum

Isabella Vieira Ferreira
Mônica Neli de Resende

Março/2012

Sumário

1 Introdução

Uma solução para as limitações dos tipos inteiros, que usualmente estão presentes em algumas linguagens de programação, é criar um tipo abstrato de dados capaz de lidar com números de tamanho arbitrário, ou seja, limitado somente pela disponibilidade de memória.

Este trabalho tem como intuito criar uma biblioteca chamada *BigNum* que permita a representação de números inteiros, tão extensos quanto for desejado, em conjunto com as operações aritméticas sobre esses números.

Este documento está estruturado da seguinte forma: na seção 3 apresentaremos o formato de entrada e saída de dados. Na seção 4, será descrito o tipo abstrato de dados e a descrição das soluções apresentadas. Na seção 5 há uma análise da complexidade das rotinas e na seção 6 os testes e a análise dos resultados obtidos.

2 Proposta de trabalho

Para criar a biblioteca *BigNum* é necessário criar um TAD com sete operações principais: *criar, destruir, somar, subtrair, multiplicar, dividir, imprimir*. O TAD será implementado utilizando alocação dinâmica de memória.

Para testar a biblioteca, é implementada uma rotina que calcula a combinação de dois números, onde tais números são provenientes de um arquivo de entrada contendo uma ou mais linhas. O resultado das linhas pares são somados e subtrai-se a soma dos resultados das linhas ímpares. O resultado final é armazenado em um arquivo de saída.

3 Formato de Entrada/Saída

A entrada de dados é feita através de um arquivo de texto denominado *"entrada.in"*. Cada linha do arquivo deve conter dois números ("n"e"m") separados por espaço, sendo estes os parâmetros para o cálculo da combinação. Vale ressaltar que o número "n"deve ser maior que "m"e os valores "0 0"indicam o final de arquivo.

```
100 6
20 5
18 6
0 0
```

Figura 1: Exemplo do arquivo "entrada.in"

A saída é dada também em um arquivo de texto denominado *"saida.out"* e contém uma única linha com o resultado da subtração entre a soma das combinações das linhas pares e a soma das combinações das linhas ímpares.

```
1192055460
```

Figura 2: Exemplo do arquivo "saida.out"

4 Solução apresentada

4.1 Tipo Abstrato de Dados

Para a implementação do tipo abstrato de dados foi utilizada uma lista linear, onde cada casa numérica do número é armazenada em uma célula. De acordo com Ziviani (2011) listas lineares são estruturas flexíveis e por serem alocadas dinamicamente favorecem a manipulação de quantidades imprevisíveis de dados, pois crescem ou diminuem de tamanho durante a execução do programa de acordo com a demanda.

Optamos por utilizar uma lista linear duplamente encadeada com descritor:

- pelo fato das operações de soma, subtração e multiplicação iniciarem a partir dos algarismos menos significativos, retornando o resultado invertido, e a divisão começar pelo algarismo mais significativo, inserindo o resultado na ordem correta na lista;
- necessidade do tamanho da lista para comparar os números de forma mais rápida e com um menor custo.

Dessa forma, a lista duplamente encadeada permite que caminhemos pela lista nos dois sentidos, começando do primeiro ou do último a um custo constante. O descritor *tamanho* é um recurso importante ao comparamos dois números, uma vez que quando esses números forem diferentes temos o resultado também a um custo constante O(1).

```
typedef struct celula *Apontador;

typedef struct celula {
    int chave;
    Apontador prev,prox;
} celula;

typedef struct {
    Apontador primeiro, ultimo;
    int tam;
} lista;
```

Figura 3: Estrutura de dados utilizada

4.2 Descrição das soluções

A solução do problema foi dividida em treze funções, sendo sete funções principais, ou seja, as operações do TAD e seis funções extras utilizadas pela operações. Apresentaremos a seguir uma breve descrição sobre o que cada uma delas realiza.

1. **void criar(lista *num)**
   A função *"criar"* é responsável por criar a célula cabeça, que é um modo de simplificar as operações sobre a lista. Por conveniência, armazenamos uma chave nessa célula a fim de a utilizarmos em algumas comparações. É também nessa função que inicializamos o descritor tamanho.

2. **void inserir(int n, lista *num)**
   A função *"inserir"* é responsável por criar uma nova célula, armazenar a chave e incrementar o tamanho da lista.

3. **void destruir (lista *num)**
   A função *"destruir"* percorre a lista liberando a memória alocada para cada célula.

4. **void somar (lista *num1, lista *num2, lista *resultado)**
   A função *"somar"* inicia com o algarismo menos significativo de ambos os números, ou seja, o último número da lista. Tal procedimento é repetido até que algum dos números acabe ou ambos atinjam a célula cabeça. Se acontecer de um número acabar antes que o outro o restante das chaves são inseridos no resultado.

5. **void subtrair (lista *minuendo, lista *subtraendo, lista *diferenca)**
   Primeiramente é feita uma comparação sobre os operandos através da função *"valida_lista"* para verificar se o minuendo é maior que o subtraendo. Caso seja, os ponteiros apontam para os algarismos menos significativos de ambas as listas. Caso contrário, ou seja, o minuendo é menor que o subtraendo, apenas invertemos os ponteiros (o minuendo passa a ser o subtraendo e o subtraendo passa a ser o minuendo) e colocamos os números mais significativos dos operandos como negativos, para que o resultado tenha o sinal correto. A subtração é repetida até que algum dos números acabe ou ambos atinjam a célula cabeça. Se acontecer de um número acabar antes que o outro o restante das chaves são inseridos no resultado.

6. **void multiplicar (lista *multiplicando, lista *multiplicador, lista *produto)**
   Primeiramente avaliamos se o multiplicando é menor que o multiplicador, caso seja apenas invertemos os ponteiros (o multiplicando passa a ser o multiplicador e este passa a ser o multiplicando). Para cada chave do multiplicador, a lista *"multiplicando"* é percorrida até atingir a célula cabeça. Vale ressaltar que para a multiplicação utilizamos um ponteiro para fazer a rotina de shift. Na primeira vez que a lista *"multiplicando"* é percorrida, os números são apenas inseridos, e na segunda vez o ponteiro *"shift"* passa a apontar para a próxima célula da lista e o resultado da multiplicação é somado com o valor já existente.

7. **void divisao (lista *dividendo, lista *divisor, lista *quociente)**
   O primeiro passo é verificar se o dividendo é menor ou igual ao divisor, pois nesses casos o resultado será respectivamente 0 ou 1. Caso contrário, efetua-se a divisão iniciando o procedimento a partir da inserção de um número maior ou igual ao divisor em uma lista *"auxiliar"*. O número encontrado é dividido pelo divisor e o processo de divisão é repetido até encontrar o final da lista *"dividendo"*.

8. **void fatorial (int n, lista *result)**
   O fatorial é calculado a partir do valor lido no arquivo de entrada, "n"ou "m", fazendo sucessivas multiplicações pelo seu antecessor até o valor um. Vale ressaltar que os números são inseridos em listas, dessa forma é realizada uma multiplicação de listas.

9. **void combinacao (int n, int m, lista *resultado_fatorial)**
   Utilizamos a função *"fatorial"* para calcular o fatorial de "n", "m"e "n - m"e obtemos o resultado da combinação de acordo com a fórmula:

$$\mathrm{C(n,m)} = \frac{n!}{(n-m)!m!}$$

10. **void imprimir (lista resultado_final, FILE *saida)**
    A função *"imprimir"* grava no arquivo de saída o resultado final, ou seja, a subtração da soma das combinações das linhas pares e a soma das combinações das linha ímpares.

11. **int valida_lista (lista *num1, lista *num2)**
    A função *"valida_lista"* é responsável por verificar se o *"num1"* é maior, menor ou igual ao *"num2"*. Primeiramente comparando os tamanhos das listas, pois se os tamanhos forem diferentes a verificação tem custo

constante. Caso sejam de tamanhos iguais inicia-se a verificação a partir do número mais significativo até encontrar a primeira chave que seja maior ou menor, retornando um e zero respectivamente. Caso contrário a função retorna dois, informando que os números são iguais.

12. **void desinverte (lista *num1, lista *resultado_desinvertido)**
A função *"desinverte"* é responsável por copiar o número de uma lista para outra, pois algumas operações precisam ser realizadas iniciando do último algarismo até o primeiro, ou seja, o resultado é colocado na lista de trás para frente. Com isso, a função *"desinverte"* tem como propósito colocar o último número da lista "resultado"na primeira célula de outra lista, desinvertendo o número.

13. **int main(int argc, char *argv[])**
A função principal contém a leitura do arquivo de entrada e é responsável por separar as linhas pares das ímpares. A cada cálculo da combinação o resultado obtido é somado com o resultado anterior correspondente a linha lida. Por fim, chamamos a função *"imprimir"* para gravar o resultado final no arquivo de saída.

5 Análise da Complexidade das Rotinas

Nesta seção faremos uma análise da complexidade das funções da biblioteca *BigNum*. Cada função será analisada individualmente e em seguida calcularemos a complexidade total a partir da análise da função principal. Ressaltamos que comparações e atribuições possuem complexidade constante O(1), *"n"* representa o tamanho da lista e as complexidades serão calculadas para o pior caso.

- *Criar*
Esta função é responsável por criar a lista. Temos apenas atribuições, logo a complexidade é O(1) . 6, onde 6 é o número de operações básicas da função, ou seja, O(1), segundo as regras de operações com a notação *O*.
Complexidade: O(1).

- *Inserir*
Esta função cria uma nova célula na lista a medida que novos números são inseridos. Como possui apenas operações básicas e o uso de listas duplamente encadeada com descritor reduz o custo de complexidade para chegar ao último (basta utilizar o descritor *"ultimo"*), temos que a complexidade é *O(1) . 7*, onde 7 é o número de operações básicas da função, logo O(1).
Complexidade: O(1).

- *Destruir*
Para destruir uma lista completa, é preciso percorrer a lista toda desalocando cada célula. Como dentro do anel temos apenas operações básicas então a complexidade da função destruir é O(n) . O(1) = O(n), segundo as regras de operações com a notação *O*.
Complexidade: O(n).

- *Desinverte*
Esta função percorre a lista até o final e insere cada valor lido em outra lista. Como a função inserir tem complexidade O(1) temos O(n) . O(1) = O(n).
Complexidade: O(n).

- *Valida_lista*
A função possui duas comparações iniciais, ou seja, com complexidade O(1), que podem terminar a função.

Porém caso não sejam satisfeitas, a lista é precorrida até que encontre um número diferente entre as duas listas. No pior caso, teremos que comparar toda a lista resultando assim uma complexidade O(n).
Complexidade: O(n).

- *Somar*
  Nesta função as duas listas são percorridas até o fim e para cada elemento da lista é realizada uma comparação, temos então O(n) . O(1) = O(n). Ao término do anel são feitas comparações verificando se uma das listas não foi percorrida totalmente. Na primeira comparação temos um anel que continuará percorrendo a lista restante fazendo uma comparação para cada elemento da lista, o que resulta em O(1) + O(n) . O(1) = O(n) onde *"n"* representa o número de células restantes. Na segunda comparação temos a chamada da função inserir - complexidade O(1) - portanto O(1) + O(1) = O(1). A função destruir - complexidade O(n) - é chamada duas vezes, logo 2 . O(n) = O(n). Como resultado final, temos: O(n) + O(1) + O(n) = O(n) para todos os casos, considerando que *"n"* é o tamanho da maior lista.
  Complexidade: O(n).

- *Subtrair*
  Nesta função há uma comparação em que é chamada a função validalista - complexidade O(n). Logo após há um anel percorrendo as duas listas fazendo várias comparações e em todos os casos a função inserir é chamada -complexidade O(1) - portanto O(n) . O(1) = O(n). Logo após o término do anel, temos uma condição verificando se ainda resta células na primeira lista (a maior sempre será a primeira lista), em caso positivo temos outro anel para terminar de percorrer a lista contendo algumas comparações. Como na função somar esse anel completa o anterior a fim de percorrer a maior lista, ou seja, complexidade O(n) onde *"n"* é o tamanho da maior lista. A função destruir - complexidade O(n) é chamada duas vezes, 2 . O(n) = O(n). Logo, temos que a complexidade total da função subtrair é O(n) + O(n) = O(n), lembrando que *"n"* é o tamanho da maior lista.
  Complexidade: O(n).

- *Multiplicar*
  Nesta função temos dois anéis aninhados que para cada célula da lista *"multiplicador"* toda a lista *"multiplicando"* é percorrida. Dentro desse anel duplo temos operacões básicas e a função inserir - complexidade O(1), ou seja, O(1) + O(1) = O(1). Logo, como a complexidade do primeiro e segundo anel é O(n) cada, temos então que a complexidade total da função multiplicar é O(n) . O(n) . O(1) = O(n^2).
  Complexidade: O(n^2).

- *Divisão*
  Nesta função temos um anel percorrendo o dividendo, porém as operações nele contidas não são executadas para cada chave, uma vez que deve-se pegar um número de chaves suficientes de modo que seja possível efetuar a divisão. Portanto a complexidade da *"divisão"* depende dos tamanhos das listas *"dividendo"* e *"divisor"*.

  Dentro do anel temos a chamada das funções destruir, subtrair e desinverte, todas de complexidade O(n), como também as funções inserir e criar, de complexidade O(1). A função multiplicar de complexidade O(n^2) é executada dentro de outro anel, porém neste caso a complexidade depende apenas do tamanho da lista *"divisor"*, pois o tamanho da lista *"multiplicando"* é um, ou seja, complexidade O(m), sendo "m"o tamanho do divisor. Esse anel é executado no máximo nove vezes, resultando na complexidade 9 . O(m). Logo, a complexidade da função é O(n) . O(m) = O(n . m), onde "n"é o tamanho do dividendo. Portanto a complexidade total da função divisao é O(n.m)
  Complexidade: O(n.m).

- *Imprimir*
  A lista é percorrida totalmente e os valores são gravados em um arquivo de saída. Portanto sua complexidade é O(n).

Complexidade: O(n).

- *Fatorial*
  Além de comparações e atribuições iniciais temos um anel variando de $(n-1)$ a 1. Dentro desse anel são chamadas as funções criar e destruir, complexidades O(1), a função destruir, complexidade O(n), e a função multiplicar de complexidade O(n^2). Logo, temos que a complexidade total da função fatorial é O(n) + O(1) + $O(n^2)$ = O(n^2).
  Complexidade: O(n^2).

- *Combinação*
  A função combinação efetua as chamadas às funções: criar, de complexidade O(1), às funções subtração, destruir e desinverte, de complexidades O(n) e as funções fatorial e multiplicar, de complexidades O(n^2), e divisão de complexidade O(n.m). O que resulta em uma complexidade total de O(1) + O(n^2) + O(n) + O($n.m$) = O(n^2).
  Complexidade: O(n^2).

- *Função principal*
  A função principal é responsável pela leitura do arquivo de entrada (n vezes). Para cada linha temos as chamadas às funções criar (O(1)), somar (O(n)), destruir (O(n)) e combinação ($O(n^2)$), . Ao final da soma o resultado é transferido para outra lista (complexidade O(n)). Por fim chama-se a função subtrair de complexidade O(n) e grava-se o resultado final no arquivo de saída, a partir da função imprimir (complexidade O(n)). Logo a complexidade total da função principal é O(1) + O(n) + O(n^2) = O(n^2).
  Complexidade: O(n^2).

6 Testes e Resultados

Nesta seção iremos apresentar a análise dos resultados obtidos em relação ao tempo de usuário, de sistema e de relógio. Vale ressaltar que as funções *criar*, *inserir* não foram analisadas por possuirem complexidade constante.

Os testes foram realizados a partir da quantidade de algarismos dos parâmetros das funções. Definimos como padrão para os testes as seguintes quantidades: 10, 100, 1.000, 10.000, 100.000, 1.000.000, 10.000.00. Esses algarismos foram gerados aleatoriamente e inseridos em uma lista linear, sendo que as duas listas possuem tamanhos iguais. Foram feitos dez testes para cada quantidade de algarismos, tendo como resultado a média final. Para realizar os mesmos foram utilizadas as funções *getrusage* e *gettimeofday*.

Para tal comparamos:

- Tempo de usuário: tempo de interação com o usuário, leitura, escrita e fatores externos;
- Tempo de sistema: tempo de processamento;
- Tempo total de execução: é a soma dos tempos de usuário e de sistema;
- Tempo de real de execução: é o tempo de relógio.

Como já foi mostrado, a função *"somar"* tem como complexidade O(n), onde "n"é o tamanho da maior lista. De acordo com o gráfico abaixo observamos que está de acordo com a complexidade calculada. Com entrada de pequenos algarismos o tempo não é significativo, a partir de números com 2.000.000 de algarismos os tempos crescem linearmente. O tempo de usuário e o tempo total de execução são bem próximos, já o tempo total de sistema é o menor, o que revela que os processos não são feitos continuamente.

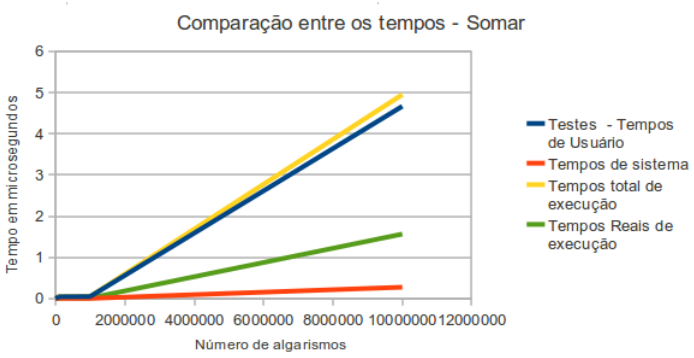

Figura 4: Testes e tempos - “Função Somar”

Na função *“subtrair”* de complexidade O(n) observamos que, como na função *“somar”*, o crescimento é linear, porém a partir de pequenas entradas. Dessa forma, o comportamento do gráfico confirma a complexidade calculada. O tempo de usuário e o tempo real de execução também são bem próximos, assim como o tempo total de execução e o tempo de sistema são menores em relação ao tamanho da entrada.

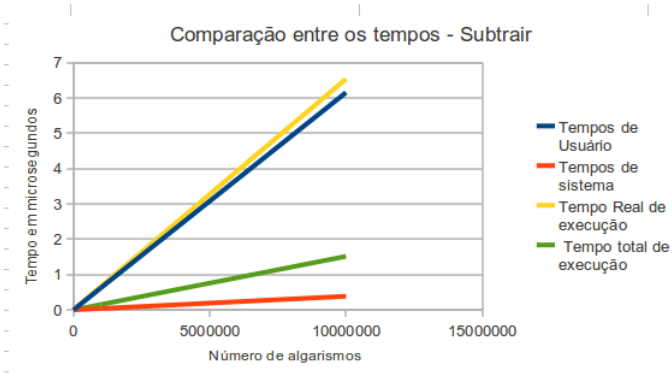

Figura 5: Testes e tempos - “Função Subtrair”

Na função *“multiplicar”* percebemos que o comportamento do gráfico tem um crescimento muito mais rápido que as funções anteriores. Isso acontece pois a função está diretamente dependendente do número de algarismos do multiplicador e do multiplicando. Logo, quanto maior o tamanho desses números, mais operações serão realizadas sobre eles. Com isso a complexidade O(n^2) condiz com o que o gráfico apresenta. Os tempos de usuário, tempo total e real de execução são muito próximos em todos os tamanhos de entrada.

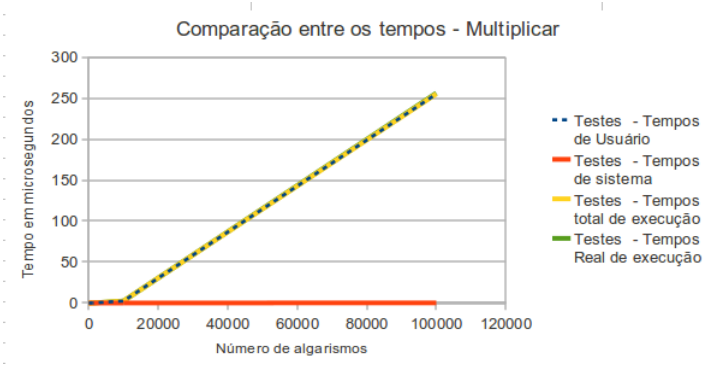

Figura 6: Testes e tempos - "Função Multiplicar"

Na divisão o crescimento do gráfico é alto mesmo para entradas pequenas, pois os testes que fizemos analisam o divisor e dividendo com mesma quantidade de algarismos.

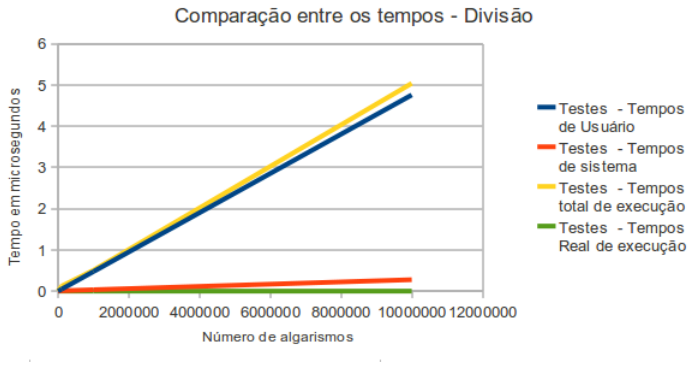

Figura 7: Testes e tempos - "Função Divisão"

Na função fatorial o tempo total de execução é o maior devido ao tempo de usuário ser muito grande. O comportamento do gráfico nos mostra que mesmo para tamanhos de entrada pequenos o tempo inicial é alto, revelando um crescimento rápido a medida que o número de algarismos cresce, confimando sua complexidade sendo O(n^2).

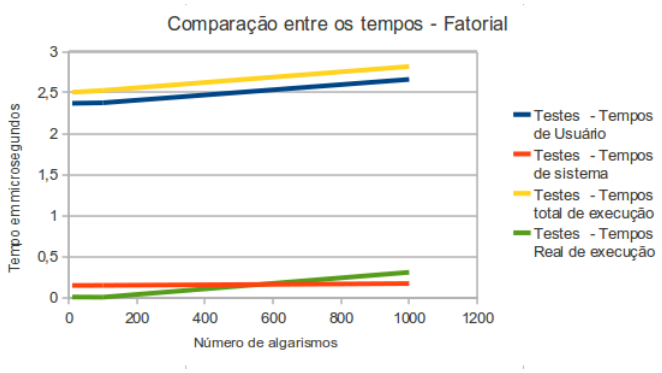

Figura 8: Testes e tempos -"Função Fatorial"

Na função *"imprimir"* temos que o tempo real de execução é mínimo quando o tamanho de entradas é pequeno, mas cresce a partir de 1.000.000 de algarismos, assim como o tempo total de execução. Como o crescimento do gráfico é lento, confirma-se a complexidade calculada de O(n).

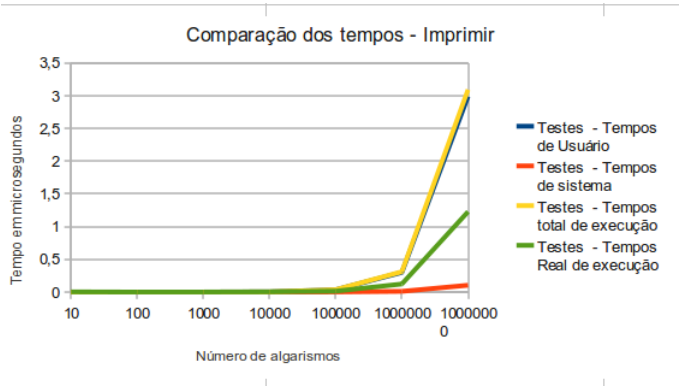

Figura 9: Testes e tempos - "Função Imprimir"

A função *"combinação"* depende principalmente dos números contidos no arquivo, uma vez que se esses números forem muito grandes ou muito pequenos o tempo é alterado significativamente.

7 Modularização

O algoritmo está modularizado da seguinte forma:

Modularização	
Arquivo	**Descrição**
big_num.c	Contém as principais operações do *BigNum*: - Criar - Destruir - Somar - Subtrair - Multiplicar - Dividir - Imprimir
big_num.h	Este arquivo contém o cabeçalho de cada função implementada no arquivo *big_num.c*
funcoes.c	Contém as funções: - Desinverte - Combinação - Fatorial
funcoes.h	Este arquivo contém o cabeçalho de cada função implementada no arquivo *funcoes.c*
funcoes_extras.c	Este arquivo contém as funções: - Inserir - Valida_lista
funcoes_extras.h	Este arquivo contém o cabeçalho de cada função implementada no arquivo *funcoes_extras.c*, bem como a estrutura de dados utilizada.
tp1.c	Contém a função principal

Tabela 1: Modularização do problema

8 Conclusão

O planejamento e desenvolvimento deste trabalho permitiu uma compreensão a respeito da necessidade do estudo de técnicas de projeto de algoritmos para se obter um melhor custo, assim como fazer uma análise detalhada da complexidade da solução desenvolvida. Fomos capazes de aperfeiçoar e aprofundar os conhecimentos sobre a linguagem C.

Por meio dos resultados obtidos observamos que o tamanho da entrada está diretamente ligado ao tempo de execução do algoritmo, e a complexidade está fortemente ligada às complexidades das rotinas das quais ela depende.

9 Referências Bibliográficas

[1] Ziviani, Nívio. Projetos de Algoritmos com implementações em Pascal e C, 2011.

[2] Kernighan, Brian W., Ritchie, Dennis M. C - A linguagem de programação padrão ANSI, 1989.

10 Apêndice

Para juntar os módulos do projeto e compilá-los foi desenvolvido um Makefile.
Para compilar o trabalho é preciso entrar na pasta /TP1/codigo/ e digitar o seguinte comando no terminal:

make

Após a compilação, para executar o programa é preciso digitar no terminal:

./tp1 entrada.in

Para fazer uma limpeza do projeto compilado digite:

make clean