Algoritmos e Estruturas de Dados 3 (2022-1)
Trabalho Prático 1

realizado pelo aluno: Henrique Franco Dalgalarrondo
proposto pelo professor: Leonardo Chaves Dutra da Rocha

1 Introdução
A proposta do seguinte trabalho é a implementação de um tipo abstrato de dados: BigNum - número de precisão arbitrária. Esse tipo deve conseguir armazenar números tão grandes quanto a memória da máquina puder aguentar, ou seja, números muito grandes. A ideia é utilizar essa TAD para conseguir calcular combinações (análise combinatória) de números grandes, sendo necessário para esse cálculo o uso do fatorial. Assim o BigNum é um tipo que armazena o resultado dos fatoriais e também tem suas próprias operações (soma, subtração, multiplicação e divisão).

2- Estrutura dos Dados
Para armazenar o BigNum, foi utilizada uma lista encadeada, com nó cabeça. Cada ponteiro da lista guarda um algarismo de base 10. O nó cabeça recebe o nome de BigNum e armazena um inteiro que guarda o tamanho da lista, um inteiro que guarda o algarismo mais significativo (chamarei de primeiro algarismo), um ponteiro para a lista encadeada, e um inteiro que guarda o sinal do número. A lista armazenará os algarismos começando do menos significativo (chamarei de último algarismo) e indo em direção ao mais significativo, dessa forma a utilização dos algoritmos das operações torna-se mais rápida, pois os algoritmos começam a calcular pelos últimos algarismos, em contrapartida a impressão dos valores na ordem natural torna-se mais lenta, pois realiza-se o processo inverso.

```
typedef struct lista {
    struct lista* prox;
    int number;
} Algarismos;

typedef struct bignum {
    int tamanho;
    int primeiro; //cópia do algarismo mais significativo da lista
    Algarismos* ultimo; //ponteiro para o inicio da lista
    int sinal;
} BigNum;
```

2.1 - Criando um BigNum
Para criar um tipo BigNum, chama-se a função criar(“numero”), que recebe o valor do BigNum como uma string. Ela aloca dinamicamente cada caractere da string a uma posição da lista encadeada, e é claro, transforma o caractere no número que ele representa antes de salvá-lo. Essa função também recebe o sinal no número, sendo o sinal negativo o

caractere “-” e o sinal positivo a ausência de caractere que difere de um número. A lista também recebe o tamanho do número pelo método strlen().

3 - Operações

3.1 Soma

Foi implementado o algoritmo clássico da soma, que soma as casas menores primeiro e transporta o excedente (vai um) para a próxima casa. Para implementá-lo criei uma função mais básica que adiciona um novo elemento no final da lista encadeada. No início da função soma faço um estudo de sinal para definir o cálculo: se os sinais entre os dois números sendo somados são diferentes, chamo a função de subtração, se são iguais, começo a soma: somo os dois primeiros algarismos caso não estejam em posição nula (ponteiro para NULL), caso o número ultrapasse o valor 9, guardo o modulo dele na casa atual, e levo seu valor dividido por dez para próxima operação.

3.2 - Subtração

Também foi implementada a solução clássica da subtração, utilizando o algoritmo que empresta os valores das casas mais significativas caso necessário. A chamada da função faz a operação dessa forma: subtração(a,b) = a - b, subtração(b,a) = b - a. No seu início também tem uma análise de sinal, caso os sinais sejam iguais continua a subtração, caso sejam diferentes a soma é chamada (uma subtração de números com sinais opostos torna-se uma soma). Foi criada uma função para comparar o tamanho dos BigNums, analisando primeiro o tamanho deles, depois os valores mais significativos até os menos significativos. Se o primeiro BigNum for maior que o segundo, essa função retorna o valor 0, caso contrário retorna o valor 1 e caso sejam iguais retorna o valor 2. Sabendo disso, o script sempre realiza a subtração do maior valor pelo menor, e no final faz a análise de sinal para definir o resultado. A subtração também faz a conta começando do final de cada número, se o resultado de uma operação é menor que 0, é somado dez ao número atual e em seguida adiciona esse número ao BigNum novo, na próxima operação é subtraído mais 1 além da subtração entre os algarismos.

3.3 - Multiplicação

Foi utilizado um algoritmo com shifting para agilizar a multiplicação. Invés de somar o número n vezes, usa o shift, que multiplica um BigNum por dez. Utiliza-se um laço para somar o “BigNum B” vezes o algarismo do “BigNum A”, e para passar multiplicar o próximo algarismo de A, faz-se um shift de B. Ex: 20 * 35: passo1 - soma 20 + 20 + 20 + 20 + 20. passo2 - faz o shift de 20 = 200. passo3 - soma 200 + 200 + 200.

3.4 - Divisão

Também foi utilizado um esquema de shift para realizar a divisão. Procura-se o divisor que multiplicado por dez é mais próximo do dividendo, para isso faz-se o shift do divisor até antes de ultrapassar o dividendo. fez-se a subtração do dividendo pelo divisor “shiftado”, e guarda o a quantidade de vezes que foi feito o shift, esse é o quociente. A próxima etapa é procurar o número que vezes o divisor mais se aproxima do resto da última operação, e assim sucessivamente. Ex: 100/5 : passo1 - procura o shift do divisor = 50 (quociente = 10) passo2 - subtraí 100 - 50 = 50. passo3 = procura o próximo shift = 50 (quociente = 10). passo4 - subtraí 50 - 50 = 0 (termina o loop). passo5 - soma dos quocientes = 20.

4 Fatorial

Como o intuito é calcular combinações, é necessário criar uma função para calcular fatoriais e guardá-los em BigNums. Utilizou-se uma versão iterativa para encontrar o fatorial.
Pede-se na chamada da função um inteiro (o fatorial de interesse). A cada iteração do script faz-se a multiplicação do número da interação pelo fatorial atual. A operações são realizadas entre BigNums, então inicialmente cria-se três BigNums, um para guardar o valor da iteração, outro para guardar o valor do fatorial a cada iteração, e um com o valor fixo 1 para somar ao BigNum que conta o loop.

```
BigNum* fatorial(int numero){
    BigNum* fat = criar("1");
    BigNum* um = criar("1");
    BigNum* conta = criar("0");

    for(int i = 0; i < numero; i++){
        conta = somar(conta,um);
        fat = multiplicar(fat,conta);
    }
    destruir(um);
    destruir(conta);
   return fat;
}
```

5 - Combinação

Implementa-se a partir das operações já mencionadas a fórmula da combinação:

$$C_{n,p} = \frac{n!}{p!(n-p)!}$$

6 - Análise de Complexidade

Irei utilizar o modelo matemático que contabiliza o número de comparações realizadas no algoritmo.

6.1 - Adiciona

N = tamanho da lista
Realiza n+1 (mais um por conta do for) comparações para caminhar até o fim da lista e adicionar o novo int.
portanto tem o comportamento assintótico linear O(n).

6.2 - Somar

i = número da iteração do loop
n = tamanho max = tamanho do maior BigNum + 1
cada iteração do loop principal realiza 7 comparações até o penúltimo laço
também chama a função adiciona n - 1 vez (adiciona = i comparações por loop)
então fica 7n + somatório(i) = 7n + (n(n+1))/2
assintoticamente fica O(n^2)

6.3 - Comparar

pior caso - quando os números são iguais
n = tamanho dos números
4 comparações para o tamanho e elemento mais significativo
um loop dentro do outro para comparar cada elemento da lista:
(i+i-1)n = -1+n^2
assintoticamente = O(n^2)

6.4 - Subtrair

n = tamanho do maior BigNum
1 comparaç ao de sinal
1 chamada ca func comparar = n^2 -1
3 comparações antes de começar o loop
4 comparações dentro do loop, ou seja, 4n
n-1 vezes a func adiciona() = (n(n+1))/2
= n^2+4n+ 2 +(n^2+n)/2
complexidade assintótica = O(n^2)

Loop que elimina os zeros desnecessários: situação causal, irei adicionar n para essa função, poderia ser n+n-1+n-2, sendo o número de zeros que sobraram que definem a quantidade de somas. No geral não tem muitos zeros para eliminar, o que não altera a complexidade assintótica.

6.4 - Multiplicação

n = tamanho de a
m = tamanho de b
m *(algarismo de b + somar)
-> m * (algarismo de b + 7n + (n(n+1))/2) + m //algarismo de b = j, j < 10
-> m*(j+(n^2+n)/2) + m
->(mn^2 +mn + 9)/2 + m
pior caso, complexidade assintótica = O(n^3)

6.5 - divisão

pior caso = subtrai 9 vezes o b do a, ex: 999 / 1
n*9
sendo n = número de casas que o a é maior que o b
toda vez tem que comparar usando a func. compara()

então -> n*m^2-1*9*g^2-1 //m = compara externo, g = compara interno
porém m e g são lineares, já que no pior caso os números têm tamanhos diferentes.
n(subtrair)*9
n(n^2+4n+ 2 +(n^2+n)/2)*9
complexidade assintótica = O(n^3)

6.6 - Imprimir
n = tamanho da lista
somatório n*(n-1)
((n+1)(n-1))/3
complexidade assintótica = O(n^2)

6.7 - Fatorial
n = índice do fatorial
n*multiplicar()
n*(O(n^3))
complexidade assintótica = O(n^4)

6.8 - Combinação
3*fatorial+multiplicacao+divisao = O(n^4)

Conclusão
A utilização do BigNum não é leve, porém não chega a ser exponencial. Há outras formas de implementação visando menor tempo, é interessante estudá-las. Poderia ter escolhido a implementação com lista duplamente encadeada, seria interessante comparar essas duas formas de realizar a TAD.