SCC-210
Algoritmos Avançados

Capítulo 5
Aritmética & Álgebra

Adaptado por João Luís G. Rosa

Organização

- Aritmética Computacional
- Números Inteiros Gigantes (*Big Numbers*)
- Bases Numéricas e Conversões
- Números Reais
- Álgebra (Polinômios)
- Logaritmos
- Bibliotecas

Aritmética Computacional

- Nos computadores modernos, as operações aritméticas elementares de **adição**, **subtração**, **multiplicação** e **divisão** são mapeadas de forma praticamente direta para instruções em nível de hardware.
- Muitas outras operações podem precisar ser realizadas via software como base em equivalências matemáticas ou aproximações numéricas. Algumas formas podem ser muito mais rápidas e/ou precisas que outras.
- Além disso, existem aplicações que demandam a manipulação de números que superam de longe a capacidade de representação convencional da máquina.

Aritmética Computacional

- Por exemplo, uma máquina de 32 bits representa e manipula um tipo inteiro primitivo **`long long`** na faixa $\pm 2^{63}$ (aproximadamente):

`-9.223.372.036.854.775.808` a **`+9.223.372.036.854.775.807`**

- Isso é mais que suficiente para a grande maioria das aplicações modernas, mas não todas. Exemplos:
  - Pesquisa matemática
  - Física/Astronomia
  - Criptologia/Criptografia
  - ...
- Em criptografia, por exemplo, muitas vezes é preciso manipular números inteiros com centenas de dígitos.

Aritmética Computacional

- Para problemas da maratona é imprescindível conhecer os limites de cada tipo de dados. Para o gnu c/c++ em uma máquina 32 bits:

Tipo	Bits	Faixa	Precisão
char, signed char	8	-128 .. 127	2
unsigned char	8	0 .. 255	2
short, signed short	16	-32.768 .. 32.767	4
unsigned short	16	0 .. 65.535	4
int, signed int	32	-2×10^9 .. 2×10^9	9
unsigned int	32	0 .. 4×10^9	9
long long	64	-9×10^{18} .. 9×10^{18}	18
unsigned long long	64	0 .. 18×10^{18}	19

Hashmat the brave warrior (UVa 10055)

Hashmat is a brave warrior who with his group of young soldiers moves from one place to another to fight against his opponents. Before fighting he just calculates one thing, the difference between his soldier number and the opponent's soldier number. From this difference he decides whether to fight or not. Hashmat's soldier number is never greater than his opponent.

Input

The input contains two integer numbers in every line. These two numbers in each line denotes the number of soldiers in Hashmat's army and his opponent's army or vice versa. The input numbers are not greater than 2^32. Input is terminated by End of File.

Output

For each line of input, print the difference of number of soldiers between Hashmat's army and his opponent's army. Each output should be in separate line.

Hashmat the brave warrior (UVa 10055)

Sample Input:

10 12
10 14
100 200

Sample Output:

2
4
100

Inteiros Gigantes (*Big Numbers*)

- Representações via concatenação de dígitos:
  - Arranjos de Dígitos:
    - Elemento inicial representa o dígito menos significativo.
    - Mantém-se um contador com o número de dígitos significativos.
    - Contador está relacionado ao índice da célula do último dígito.
    - Vantagens:
      - Simples.
      - Eficiente: 100.000 dígitos $\Rightarrow$ Arranjo de 100.000 *chars* $\Rightarrow$ 100Kb !!!
    - Requerimento:
      - Limitante superior não conservativo para o no. máximo de dígitos.
  - Lista Encadeada de Dígitos (Estrutura Dinâmica):
    - Mais complexa, mas necessária quando não se dispõe de um limitante superior não conservativo para o número de dígitos.

Inteiros Gigantes (*Big Numbers*)

- Representação via Arranjo de Dígitos:

```
#include <stdio.h>

#define MAXDIGITS   100      /* maximum length bignum */
#define PLUS        1        /* positive sign bit */
#define MINUS       -1       /* negative sign bit */

typedef struct {
        char digits[MAXDIGITS]; /* represent the number */
        int signbit;            /* 1 if positive, -1 if negative */
        int lastdigit;          /* index of high-order digit */
} bignum;
```


Inteiros Gigantes (*Big Numbers*)

- Representação via Arranjo de Dígitos:

```
print_bignum(bignum *n)
{
    int i;

    if (n->signbit == MINUS) printf("- ");
    for (i=n->lastdigit; i>=0; i--)
        printf("%c",'0'+ n->digits[i]);

    printf("\n");
}
```

* Nota:

- Os dígitos não correspondem aos caracteres 0 a 9 (ASCII 48 a 57), mas aos caracteres cujos códigos ASCII são 0 a 9.
- Isso permite operar com os dígitos (**`char`**) como se fossem inteiros (**`int`**).

Inteiros Gigantes (*Big Numbers*)

- Adição:

```
add_bignum(bignum *a, bignum *b, bignum *c)
{
    int carry;
    int i;

    initialize_bignum(c);

    if (a->signbit == b->signbit) c->signbit = a->signbit;
    else {
        if (a->signbit == MINUS) {
            a->signbit = PLUS;
            subtract_bignum(b,a,c);
            a->signbit = MINUS;
        } else {
            b->signbit = PLUS;
            subtract_bignum(a,b,c);
            b->signbit = MINUS;
        }
        return;
    }
    // continua...
```

(–|a|) + (–|b|) = –(|a| + |b|)

|a| + |b| = +(|a| + |b|)

(–|a|) + |b| = |b| – |a|

|a| + (–|b|) = |a| – |b|

Inteiros Gigantes (*Big Numbers*)

- Adição:

c->lastdigit = max(a->lastdigit,b->lastdigit)
ou c->lastdigit = max(a->lastdigit,b->lastdigit)+1
Na dúvida utiliza-se o maior + a função **zero_justify(c)**

```
    // ...continua...
    c->lastdigit = max(a->lastdigit,b->lastdigit)+1;
    carry = 0;

    for (i=0; i<=(c->lastdigit); i++) {
        c->digits[i] = (char) (carry+a->digits[i]+b->digits[i]) % 10;
        carry = (carry + a->digits[i] + b->digits[i]) / 10;
    }
    zero_justify(c);
}
```

divisão inteira

```
1 1 1
  134
  968
 1102
```

```
zero_justify(bignum *n) {
    while ((n->lastdigit > 0) && (n->digits[ n->lastdigit ] == 0))
        n->lastdigit --;
        if ((n->lastdigit == 0) && (n->digits[0] == 0))
            n->signbit = PLUS;  /* hack to avoid -0 */
}
```

Inteiros Gigantes (*Big Numbers*)

- Comparação:

```
compare_bignum(bignum *a, bignum *b)
{
    int i;              /* counter */

    if ((a->signbit == MINUS) && (b->signbit == PLUS)) return(PLUS);
    if ((a->signbit == PLUS) && (b->signbit == MINUS)) return(MINUS);

    if (b->lastdigit > a->lastdigit) return (PLUS * a->signbit);
    if (a->lastdigit > b->lastdigit) return (MINUS * a->signbit);

    for (i = a->lastdigit; i>=0; i--) {
        if (a->digits[i] > b->digits[i]) return(MINUS * a->signbit);
        if (b->digits[i] > a->digits[i]) return(PLUS * a->signbit);
    }

    return(0);
}
```

  - Retorna: +1 se a < b , –1 se a > b , 0 se a=b

Inteiros Gigantes (*Big Numbers*)

- Subtração:

a<0 & b<0:

–|a| – (–|b|) =

–|a| + |b|

a>0 & b<0:

|a| – (–|b|) =

|a| + |b|

a<0 & b>0:

–|a| – |b| =

–|a| + (–|b|)

```
subtract_bignum(bignum *a, bignum *b, bignum *c) {
    int borrow;         /* has anything been borrowed? */
    int v;              /* placeholder digit */
    int i;              /* counter */

    initialize_bignum(c);

    if ((a->signbit == MINUS) || (b->signbit == MINUS)) {
       b->signbit = -1 * b->signbit;
       add_bignum(a,b,c);
       b->signbit = -1 * b->signbit;
       return;
       }

    if (compare_bignum(a,b) == PLUS) {
        subtract_bignum(b,a,c);
        c->signbit = MINUS;
        return;
    }
```

a>0 & b>0 & a<b:

|a| – |b| =

– (|b| – |a|)

Inteiros Gigantes (*Big Numbers*)

- Subtração:

```
 1 1
 104   → a
 28    → b
 76
```

```
    // ...continuação
    c->lastdigit = max(a->lastdigit,b->lastdigit);
    borrow = 0;

    for (i=0; i<=(c->lastdigit); i++) {
        v = (a->digits[i] - borrow - b->digits[i]);
        borrow = 0;
        if (v < 0) {
            v = v + 10;
            borrow = 1;
        }
        c->digits[i] = (char) v;
    }
    zero_justify(c);
}
```


Inteiros Gigantes (*Big Numbers*)

- Multiplicação:

```
 1 2
 3 4
 1 5 7   → a
   3 6   → b
 9 4 2   → 6 × a = a + a + a + a + a + a
 4 7 1 0 → (3 × a) × 10 = 10 × a + 10 × a + 10 × a
 5 6 5 2 → (a + a + a + a + a + a) + (10×a + 10×a + 10×a)
```

```
bignum c;
bignum tmp;
initialize_bignum(c);
for (j=1; j<=b.digits[0]; j++){
    add_bignum(&c,&a,&tmp);
    c = tmp;}
```

```
digit_shift(bignum *n, int d) {      /* multiply n by 10^d */
    int i;                           /* counter */
    for (i=n->lastdigit; i>=0; i--)
        n->digits[i+d] = n->digits[i];
    for (i=0; i<d; i++) n->digits[i] = 0;
    n->lastdigit = n->lastdigit + d;
}
```

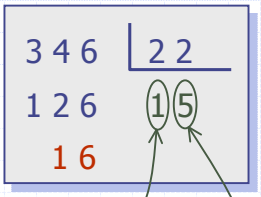
Inteiros Gigantes (*Big Numbers*)

- Multiplicação:

```
multiply_bignum(bignum *a, bignum *b, bignum *c)
{
    bignum row;          /* represent shifted row */
    bignum tmp;          /* placeholder bignum */
    int i,j;             /* counters */

    initialize_bignum(c);

    row = *a;

    for (i=0; i<=b->lastdigit; i++) {
        for (j=1; j<=b->digits[i]; j++) {
            add_bignum(c,&row,&tmp);
            *c = tmp;
        }
        digit_shift(&row,1);
    }

    c->signbit = a->signbit * b->signbit;

    zero_justify(c);
}
```


Inteiros Gigantes (*Big Numbers*)

- Divisão (Resto Desprezado):
  - 346 ÷ 22
    - 3 ≥ 22 ? (F)
    - 34 ≥ 22 ? (**V**)
      - 34 – 22 = 12 (1x)
      - 12 ≥ 22 (F)
      - 126 ≥ 22 (**V**)
        - 126 – 22 = 104 (1x)
        - 104 – 22 = 82 (2x)
        - 82 – 22 = 60 (3x)
        - 60 – 22 = 38 (4x)
        - 38 – 22 = 16 (5x)
        - 16 ≥ 22 (F)



Inteiros Gigantes (*Big Numbers*)

```
divide_bignum(bignum *a, bignum *b, bignum *c)
{
        bignum row;
        bignum tmp;
        int asign, bsign;
        int i,j;

    initialize_bignum(c);

    c->signbit = a->signbit * b->signbit;

    asign = a->signbit;
    bsign = b->signbit;

    a->signbit = PLUS;
    b->signbit = PLUS;

    initialize_bignum(&row);
    initialize_bignum(&tmp);

    // continua...
```

```
    // ... continuação
    c->lastdigit = a->lastdigit;
    for (i=a->lastdigit; i>=0; i--) {
        digit_shift(&row,1);
        row.digits[0] = a->digits[i];
        c->digits[i] = 0;
        while (compare_bignum(&row,b) != PLUS){
            c->digits[i] ++;
            subtract_bignum(&row,b,&tmp);
            row = tmp;
        }
    }
    zero_justify(c);
    a->signbit = asign;
    b->signbit = bsign;
}
```

Divisão Inteira

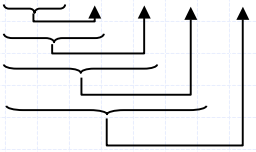
Inteiros Gigantes (*Big Numbers*)

- Potenciação:
  - $a^n = a \times a \times \ldots \times a \rightarrow (n-1)$ multiplicações

    ou
  - $a^n = a^{n \div 2} \times a^{n \div 2} \times a^{(n \bmod 2)} \rightarrow O(\log n)$ multiplicações
  - Exemplo:

$a^{16} = a^8 \times a^8 = (a^4 \times a^4) \times a^8 = (a^2 \times a^2) \times a^4 \times a^8 = a \times a \times a^2 \times a^4 \times a^8$

$a^{33} = a^{16} \times a^{16} \times a = a \times a \times a^2 \times a^4 \times a^8 \times a^{16} \times a$



Inteiros Gigantes (*Big Numbers*)

- Potenciação (Solução Recursiva):

```
bignum exp_bignum(bignum *a, int n) /* Somente para n >= 1 */
{
        int aux;
        bignum tmp, tmp2;
        aux = (int) floor((double) n/2);
        if (aux>0){
            tmp2 = exp_bignum(a,aux);
            multiply_bignum(&tmp2, &tmp2, &tmp);
            if (n%2!=0){
                        multiply_bignum(&tmp,a,&tmp2);
                        tmp = tmp2;
            }
            return(tmp);
        }
        else if (n==1) { tmp = *a; return(tmp); }
}
```


Inteiros Gigantes (*Big Numbers*)

- Conversão e Inicialização:

```
int_to_bignum(int s, bignum *n) {
    int i;              /* counter */
    int t;              /* int to work with */
    if (s >= 0) n->signbit = PLUS; else n->signbit = MINUS;
    for (i=0; i<MAXDIGITS; i++) n->digits[i] = (char) 0;
    n->lastdigit = -1;
    t = abs(s);
    while (t > 0) {
        n->lastdigit ++;
        n->digits[ n->lastdigit ] = (t % 10);
        t = t / 10;
    }
    if (s == 0) n->lastdigit = 0;
}

initialize_bignum(bignum *n) {
    int_to_bignum(0,n);
}
```

500! (UVa 623)

In these days you can more and more often happen to see programs which perform some useful calculations being executed rather then trivial screen savers. Some of them check the system message queue and in case of finding it empty (for examples somebody is editing a file and stays idle for some time) execute its own algorithm.

As an examples we can give programs which calculate primary numbers.

One can also imagine a program which calculates a factorial of given numbers. In this case it is the time complexity of order $O(n)$ which makes troubles, but the memory requirements. Considering the fact that 500! gives 1135-digit number no standard, neither integer nor floating, data type is applicable here.

500! (UVa 623)

Your task is to write a programs which calculates a factorial of a given number.

Assumptions: Value of a number n which factorial should be calculated of does not exceed 1000 (although 500! is the name of the problem, 500! is a small limit).

Input

Any number of lines, each containing value n for which you should provide value of $n!$

Output

2 lines for each input case. First should contain value n followed by character `!'. The second should contain calculated value $n!$.

500! (UVa 623)

Sample Input

10
30
50
100

Sample Output

10!
3628800
30!
265252859812191058636308480000000
50!
30414093201713378043612608166064768844377641568960512000000000000
100!
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000

Ones (UVa 10127)

Popularity: A, Sucess rate: high, Level: 2

Given any integer $0 \le n \le 10000$ not divisible by 2 or 5, some multiple of n is a number which in decimal notation is a sequence of 1's. How many digits are in the smallest such multiple of n?

Input

A file of integers at one integer per line.

Output

Each output line gives the smallest integer $x > 0$ such that $p = \sum_{i=0}^{x-1} 1 \times 10^i$, where a is the corresponding input integer, $p = a \times b$, and b is an integer greater than zero.

Ones (UVa 10127)

Sample Input

3
7
9901

Sample Output

3
6
12

Bases Numéricas & Conversão

- A **base** de um sistema de numeração diz respeito a quantos diferentes símbolos são utilizados naquele sistema.
- Bases numéricas mais usuais:
  - Decimal: Utiliza dez dígitos (0 a 9). A razão histórica para a adoção dessa base-10 é que aprendemos a contar com os dedos das mãos*.
  - Binária: Utiliza dois dígitos (0 e 1). A razão evidente para a adoção dessa base-2 em sistemas digitais é o seu mapeamento natural com os estados ligado/desligado da lógica dos circuitos.
  - Hexadecimal: Utiliza 16 símbolos (dígitos 0 a 9 + letras A a F). Simplifica a representação de números binários pela conversão direta de grupos de 4 bits em um dígito hexadecimal.
  - Octal: 8 dígitos (0 a 7). Representa nos. binários por grupos de 3 bits.

* Nota: Os Maias utilizavam um sistema com base 20, supostamente porque também utilizavam os pés para contar.

Bases Numéricas & Conversão

- Conversão de número decimal *N* em qualquer outra base *b*:
  - Divisão inteira sucessiva de *N* por *b*.
  - Dígitos menos significativos do resultado são dados pelas sobras.
  - Sobras são facilmente calculadas utilizando o operador módulo (%):
  - Exemplos:

93 / 2	=	46	resto **1**
46 / 2	=	23	resto **0**
23 / 2	=	11	resto **1**
11 / 2	=	5	resto **1**
5 / 2	=	2	resto **1**
2 / 2	=	1	resto **0**
1 / 2	=	0	resto **1**

01011101

93 / 16	=	5	resto **D**
5 / 16	=	0	resto **5**

5D

Bases Numéricas & Conversão

- Conversão Inversa:
  - Sejam d_0 e d_n os dígitos menos e mais significativos, respectivamente, de *N* na base *b* (*Nb*). Então:

$$N_{10} = d_0 \times b^0 + d_1 \times b^1 + \ldots + d_{n-1} \times b^{n-1}$$

  - Exemplos:
    - $\mathbf{01011101_2} = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + \ldots + 1 \times 2^6 + 0 \times 2^7 = \mathbf{93_{10}}$
    - $\mathbf{5D_{16}} = D \times 16^0 + 5 \times 16^1 = 13 \times 1 + 5 \times 16 = \mathbf{93_{10}}$

Números Reais

- Trabalhar com números reais em computadores é desafiador
- Representação é feita em notação científica:
  - *Mantissa* $\times$ Base$^{(Expoente)}$
- Problemas:
  - Aritmética de ponto flutuante possui precisão limitada:
    - Notação científica pode passar a ilusão de capacidade gigantesca
    - Fato é que a mantissa e expoente possuem no. finito de bits
  - Não existe padrão único (embora exista padrão usual – e.g. IEEE):
    - Para representação (e.g. em base 10 ou base 2).
    - Para operações (e.g. arredondamentos): podem variar com a arquitetura do computador, com a linguagem de programação e com o compilador!

Números Reais

- Matemática *versus* Computação
  - Continuidade dos reais ($a<b \Rightarrow \exists c: a<c<b$):
    - Matemática (Verdadeiro para $\forall$ a, b $\in \Re$)
    - Computação (Não necessariamente verdadeiro)
  - Propriedades Algébricas
    - Ex. Associatividade da Adição: (a + b) + c = a + (b + c)
    - Matemática (Verdadeiro para $\forall$ a, b, c $\in \Re$)
    - Computação (Não necessariamente – Erros de arredondamento)
    - Maiores problemas ocorrem na manipulação de números com magnitudes muito distintas (muito grandes e muito pequenos).
    - Implica em outro grande problema: Teste de Igualdade.

Números Reais

- Teste de Igualdade e Precisão:
  - Testes de igualdade exatos:
    - Verificar se dois números reais são binariamente idênticos
    - Verificar se um número real é idêntico a zero
    - São usualmente inadequados devido à presença de erros de arredondamento nos bits menos significativos da mantissa.
  - Testes de igualdade aproximados:
    - Verificar se um número real está dentro de um intervalo de tolerância $[-\varepsilon,+\varepsilon]$ ao redor da referência de comparação.
    - Esse é o tipo de teste recomendado.

Números Reais

- Truncamento *versus* Arredondamento:
  - **Truncamento**:
    - Desprezar os *n* dígitos menos significativos do número.
    - Exemplo: funções "floor", que desprezam parte fracionária.
    - Para desprezar apenas após do *k*-ésimo dígito decimal:
      - $trunc(A, k) = floor(A \times 10^k) / 10^k$
      - Exemplo: $trunc(10.2345 , 2) = floor(10.2345 \times 10^2) / 10^2 = 10.23$
  - **Arredondamento**:
    - Aproximar os *n* dígitos menos significativos do número.
    - Para aproximar os dígitos após do *k*-ésimo dígito decimal:
      - $round(A, k) = floor(A \times 10^k + 1/2) / 10^k$
      - Ex.: $round(10.2345, 2) = floor(10.2345 \times 10^2 + 0.5) / 10^2 = 10.23$
      - Ex.: $round(10.2355, 2) = floor(1024.05) / 10^2 = 10.24$

Números Reais

- Tipos de Números:
  - **Inteiros**:
    - São os números contáveis, cujos subconjuntos incluem os números Naturais e os Positivos (em notação não universal).
  - **Racionais**:
    - Números que podem ser expressos como a razão de dois inteiros.
    - No. *c* é racional se e somente se $c = a/b$, onde *a* e *b* são inteiros.
    - Obviamente, qualquer no. *c* inteiro é racional pois $c = c/1$.
  - **Irracionais**:
    - Números que não podem ser expressos como a razão de dois inteiros.
    - É possível demonstrar que não existem dois inteiros *a* e *b* cuja razão resulte em números como $\pi = 3.1415...$, $e = 2.7182...$, entre outros.

Números Reais

- Representação de Números Irracionais:
  - Pré-definição aproximada com *n* dígitos:
    - Exemplo: $\pi \cong 3.1415926$
    - ~10 dígitos é suficiente para a maioria das aplicações
  - Aproximação numérica:
    - Números irracionais que representam o valor $f(x_0)$ de uma função analítica $f(x)$ em um dado ponto x_0 podem ser aproximados via série de Taylor
    - Exemplos incluem: $e = 2.7182...$, sqrt(2) = 1.41421...

Números Reais

- Série de Taylor:

$$f(x) = \sum_{m=0}^{\infty} \frac{f^m(x_0)}{m!}(x - x_0)^m$$

- Exemplo:

$$e^x = e^{x_0} + (x - x_0)\left[\frac{d(e^x)}{dx}\right]_{x=x_0} + \frac{1}{2}(x - x_0)^2\left[\frac{d^2(e^x)}{d^2x}\right]_{x=x_0} + \ldots$$

  - Dado que $d(e^x)/dx = e^x$, tem-se tomando $x_0 = 0$ que:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} \cdots$$

$$e = 1 + 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \frac{1}{120} + \frac{1}{720} + \cdots = 2.7181 + \cdots$$

Álgebra - Polinômios

- Polinômio é uma função do tipo: $p(x) = c_0 + c_1x + \ldots + c_nx^n$
- "n" é o *grau* do polinômio, isto é:
  - O maior expoente j de um termo x^j com coeficiente c_j não nulo
- Cálculo de $p(x)$:
  - Literal: $O(n + n\text{-}1 + \ldots + 2) \equiv \mathbf{O(n^2)}$ multiplicações.
  - Duas multiplicações por termo, a partir do termo de menor ordem:
    - Para $j = 2, \ldots, n$ multiplica-se ($x^{j-1} \times x$) e o resultado por c_j, i.e., $(x^j \times c_j)$
    - $\mathbf{O(2n)} \equiv O(n)$ multiplicações
  - Regra de Horner: $c_0 + c_1x + \ldots + c_nx^n = c_0 + x(\ldots + x(c_{n-1} + c_nx)\ldots)$
    - $O(n)$ multiplicações

Polinômios

- Adição:
  - $p(x) = c_0 + c_1x + \ldots + c_nx^n$ e $q(x) = d_0 + d_1x + \ldots + d_nx^n$
  - $p(x) + q(x) = (c_0 + d_0) + (c_1 + d_1)\,x + \ldots + (c_n + d_n)\,x^n$
  - OBS: Graus de *p* e *q* não precisam ser iguais!
- Subtração:
  - $p(x) - q(x) = (c_0 - d_0) + (c_1 - d_1)\,x + \ldots + (c_n - d_n)\,x^n$
- Multiplicação:

$$p(x)\times q(x) = \sum_{i=0}^{grau(p)} \sum_{j=0}^{grau(q)} \left(c_i c_j x^{i+j}\right)$$

  - Implementação literal é ***O(n²)*** assumindo $n = grau(p) = grau(q)$
  - Possível calcular em ***O(n log n)*** como convolução via **F.F.T. (*fast Fourier transform*)**

Polinômios

- Divisão:
  - Polinômios **não são fechados sob divisão**.
  - Isso significa que uma divisão polinomial não necessariamente resulta em um polinômio.
- Representação:
  - Usualmente por **vetor de coeficientes**.
  - No caso de polinômios multidimensionais:
    - Matriz ou arranjo multi-dimensional de coeficientes
    - Exemplo: *c[i][j]* representa o coeficiente c_{ij} associado ao termo $x^i \times y^j$
- Polinômios Esparsos:
  - Polinômios com grau "*n*" elevado e muitos coeficientes c_j nulos
  - Podem ser representados com **lista encadeada de coeficientes**
  - Elevado espargimento pode justificar essa representação alternativa

Polinômios

- Cálculo de Raízes Polinomiais:
  - Identificar um ou mais valores de x tal que $p(x) = v$
  - Se $p(x)$ tem grau $n = 1$ então $c_1 x + c_0 = v \rightarrow \boxed{x = (v - c_0) / c_1}$
  - Se $p(x)$ tem grau $n = 2$ então:

$$x = \frac{-c_1 \pm \sqrt{c_1^2 - 4c_2(c_0 - v)}}{2c_2}$$

  - Existem ainda fórmulas fechadas mais complexas para $n = 3$ e $n = 4$
  - A partir de $n = 3$, no entanto, tipicamente se utiliza métodos numéricos:
    - **Newton**
    - **Newton-Raphson**
    - **etc**

Logaritmos

- Um logaritmo nada mais é que uma função exponencial inversa:
  - Dizer que $b^x = y$ é equivalente a dizer que $x = \log_b(y)$
  - b é denominada *base* do logaritmo
- Três bases de particular importância por razões matemáticas e históricas são:
  - Base e
    - $\log_e(x) := ln(x) \rightarrow$ conhecido como log. *neperiano* ou *natural*
  - Base 2
  - Base 10 $\rightarrow$ *logaritmo comum*

Logaritmos

- A importância do logaritmo natural advém da sua relação inversa com a função exponencial:
  - $exp(ln(x)) = e^{ln(x)} = x$
- A importância do logaritmo em base 2 é dupla:
  - Sistema de numeração binário.
  - Análise de complexidade de algoritmos.
- O logaritmo em base 10 é menos usual atualmente:
  - Foi importante para o cálculo manual envolvendo grandes números, antes da invenção de calculadoras e computadores.
  - **Tábuas logarítmicas para multiplicações**

Logaritmos

- Logaritmos ainda são úteis para multiplicações, em particular aquelas envolvidas em potências:
  - $log_a(xy) = log_a(x) + log_a(y)$
  - Conseqüentemente: $log_a(n^b) = b \times log_a(n)$
  - Logo: $a^b = exp(ln(a^b)) = exp (b\ ln(a)) \quad \forall a, b \in \Re$
  - **Com exp e ln implementadas via Taylor calcula-se a^b !**
  - Podemos utilizar esse método para calcular a raiz quadrada:
    - De fato, $sqrt(x) = x^{1/2}$
    - Mas não se surpreenda se algum compilador produzir $e^{(0.5\ ln\ 4)} \neq 2$!
- Uma última propriedade muito útil: $$\log_a b = \frac{\log_c b}{\log_c a}$$

Algumas Bibliotecas

- **C/C++**:
  - <math.h> library
    - `double floor(double x);`
    - `double ceil(double x);`
    - `double fabs(double x);` // Equivalente para real de "abs()" para **int**
    - `double sqrt(double x);`
    - `double exp(double x);`
    - `double log(double x);` // Logaritmo Neperiano.
    - `double log10(double x);`
    - `double pow(double x,double y);`
    - etc
  - GNU g++ **`Integer`** class.
- **Java**:
  - **`java.lang.Math`** class
  - **`java.math.BigInteger`** class

Para a próxima aula

- 10579 – *Fibonacci Numbers* (Obrigatório)
- 11410 – *LAEncoding* (Opcional)
- 568 – *Just the Facts* (Opcional)

Fibonacci Numbers

- A Fibonacci sequence is calculated by adding the previous two members of the sequence, with the first two members being both 1.
  - $f(1) = 1, f(2) = 1, f(n > 2) = f(n - 1) + f(n - 2)$
- **Input and Output**
  - Your task is to take numbers as input (one per line), and print the corresponding Fibonacci number.
  - **Sample Input**
    - 3 100
  - **Sample Output**
    - 2 354224848179261915075
  - **Note:** No generated Fibonacci number in excess of 1000 digits will be in the test data, i.e. $f(20) = 6765$ has 4 digits.

Referências

- Batista, G. & Campello, R.
  - Slides disciplina *Algoritmos Avançados*, ICMC-USP, 2007.
- Skiena, S. S. & Revilla, M. A.
  - *Programming Challenges – The Programming Contest Training Manual*. Springer, 2003.