# Teoria dos Números

Laboratório de Programação Competitiva - 2020

Pedro Henrique Paiola (paiola@fc.unesp.br)
Giulia Moura Crusco (giulia@fc.unesp.br)
João Pedro Marin Comini (joaocomini@gmail.com)
**Unesp Bauru**

# BigInteger

- Certos problemas da Maratona de Programação recebem como entrada números inteiros que **extrapolam** o limite de variáveis do tipo **long long int**
- Tamanho de uma variável long long int: **8 bytes**
- Intervalo de números que podem ser armazenados em uma variável desse tipo:
    - -9.223.372.036.854.775.808 à 9.223.372.036.854.775.807
    - 0 à 18.446.744.073.709.551.615 (unsigned long long int)

# BigInteger

- Exemplo: [2667 - Jogo de Boca]
    - Entrada: N ($3 <= N <= 10^{100}$)

- E agora? O que fazer?

# BigInteger

- 1ª Opção: dependendo das operações necessárias de se fazer com o número, podemos ler o número como sendo uma **string** e trabalhar com essa string.

- Exemplos:
    - Operações simples com dígitos
    - Uso de Aritmética Modular

# BigInteger

- 2ª Opção: se precisarmos fazer operações com esse número como **soma, subtração, multiplicação e divisão**, o problema se torna mais complexo.

- Nesses casos, não recomendamos usar a linguagem C++. É possível trabalhar com BigInteger em C++ (a biblioteca do Thiago traz códigos para isso), porém a quantidade de código necessária é relativamente grande.
- Sugestões: **Java** ou **Python**

# BigInteger

- Em Python, não precisamos nos preocupar muito com o tamanho de um inteiro, a memória é alocada conforme o necessário para comportar o tamanho do número.

  Entrada e Saída em Python

  Python em Programação Competitiva

  Muita coisa sobre Python

# BigInteger

- Em Java podemos usar a classe [BigInteger] da biblioteca java.math

```
String Num;
BigInteger NumGrande;
Scanner S = new Scanner(System.in);
Num = S.nextLine();
NumGrande = new BigInteger(Num);
NumGrande = NumGrande.mod(new BigInteger("3"));
System.out.println(NumGrande);
```

# Teoria dos Números

- A Teoria dos Números é o ramo da matemática que se preocupa com as propriedades dos **números inteiros**.
- Existe uma coleção de algoritmos interessantes derivados de estudos da Teoria dos Números que solucionam problemas de forma inteligente e eficiente.
- Aqui faremos uma breve introdução à alguns tópicos relativos à Teoria dos Números.

# Números primos

- Diversos problemas envolvem o uso de números primos.
- Dessa forma, precisamos, inicialmente, de uma forma de testar se um número é primo ou não.

- Recordando: números primos são números naturais que têm apenas dois divisores: 1 e ele mesmo.

# Números primos

- Algoritmo ingênuo O(n)

```
bool ehPrimo(int n)
{
    for(int i = 2; i < n; i++)
        if (n % i == 0)
            return false;
    return true;
}
```

# Números primos

- Porém, na verdade só precisamos testar até √(n)
- Demonstração:

Suponha que não, nesse caso existe **n** tal que o menor fator primo **p** de **n** é maior que **√(n)**.

Se **p** divide **n**, então **n/p** também divide **n**, e **n/p** deve ser maior que **√(n)**.

Mas se **p** > **√(n)** e **n/p** > **√(n)**, então **p** * **(n/p)** > **n**, o que é um absurdo!

# Números primos

- Algoritmo O(√(n))

```
bool ehPrimo(int n)
{
    int raiz = sqrt(n);
    for(int i = 2; i <= raiz; i++)
        if (n % i == 0)
            return false;
    return true;
}
```

# Números primos

- Algoritmo O(√(n))

```
bool ehPrimo(int n)
{
    for(int i = 2; i * i <= n; i++)
        if (n % i == 0)
            return false;
    return true;
}
```

# Crivo de Eratóstenes

- O Crivo de Eratóstenes é um método de encontrar os números primos até um certo valor limite.
- Útil em casos que faremos vários testes de primalidade e na fatoração de números.
- Ideia geral: dado que um número **p** é primo, marcamos os múltiplos de **p** como não sendo números primos.

# Crivo de Eratóstenes

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |
| 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 |

Prime numbers

# Crivo de Eratóstenes

- Algoritmo:
  - Cria-se uma lista de 2 a MAX, marcando todos como primos
  - Para cada número i de 2 até raiz(MAX)
    - Se i está marcado como primo
      - Marcar todos os números múltiplos de i a partir de i*i como compostos (não primos)

# Crivo de Eratóstenes

- Algoritmo:
  - Cria-se uma lista de 2 a MAX, marcando todos como primos
  - Para cada número i de 2 até raiz(MAX)
    - Se i está marcado como primo
      - Marcar todos os números múltiplos de i a partir de i*i como compostos (não primos)

Por que posso marcar só a partir de i*i?

# Crivo de Eratóstenes

- Antes de **i * i** temos: i*2, i*3, i*4, ... i*(i-1). Ou ainda, **i*j | 2 < j < i**
- Seja **x = i * j**, x é **múltiplo de i** e também é **múltiplo de j**
- Todo j ou é primo, ou é múltiplo de um número primo menor que i, ou seja, um primo já “descoberto” pelo algoritmo
- Se j é primo
    - Todos os seus múltiplos foram marcados como não primo, inclusive i*j
- Se j é múltiplo de um primo p < i
    - Então ele já foi marcado como composto, por ser múltiplo de p, assim como todos os seus múltiplos
- Logo, todos os números **i*j | 2 < j < i** já foram marcados

# Crivo de Eratóstenes

```
bool ehPrimo[MAX+1];
vector<int> primos;

void crivo(int n){
    memset(ehPrimo, true, sizeof(ehPrimo);
    for(int p = 2; p * p <= n; p++){
        if (ehPrimo[p]){
            primos.push_back(p); //Lista incompleta, com primos até sqrt(n)
            for(int i = p*p; i <= n; i += p)
                ehPrimo[i] = false;
        }
    }
}
```

# “Look-up tables”

- Existem casos onde podemos gerar um **vetor ou matriz de consulta** manualmente (ou previamente por outro programa), e inseri-los prontos no nosso programa. Dessa forma, economiza-se o tempo de gerar tal vetor/matriz.
- Por exemplo, se para resolver um problema precisamos de todos os primos até N, podemos embutir um vetor de primos já dentro do código.

```
int primos[] = {2, 3, 5, 7, 11, 13, ... }
```

- Isso também pode ser gerado por um programa auxiliar

# “Look-up tables”

“The judge can’t look into your heart or your program to see your intentions: it only checks the results.” (Skiena & Revilla, 2003; p. 129)

# Fatoração

```
vector<int> fatorar(int n){
    vector<int> fatores;
    int i = 0;
    while(n > 1){
        while(n % primos[i] == 0){
            fatores.push_back(primos[i]);
            n /= primos[i];
        }
        i++;
    }
    return fatores;
}
```

# Máximo Divisor Comum

- Problema: encontrar o maior divisor comum de um par de números.
- Algoritmo de Euclides

- OBS: se mdc(x, y) = 1, então dizemos que x e y são coprimos ou primos entre si.

# Máximo Divisor Comum

O processo das divisões sucessivas

$$\begin{array}{c|c} a & b \\ \hline r_1 & q_1 \end{array} \qquad \begin{array}{c|c} b & r_1 \\ \hline r_2 & q_2 \end{array} \qquad \begin{array}{c|c} r_1 & r_2 \\ \hline r_3 & q_3 \end{array} \quad \cdots \quad \begin{array}{c|c} r_{n-2} & r_{n-1} \\ \hline r_n & q_n \end{array} \qquad \begin{array}{c|c} r_{n-1} & r_n \\ \hline 0 & q_{n+1} \end{array}$$

nos garante que:

$$mdc\,(a,b) = mdc(b, r_1) = mdc(r_1, r_2) = \cdots = mdc(r_{n-2}, r_{n-1}) = mdc(r_{n-1}, r_n) = mdc(r_n, 0) = r_n$$

Esse processo pode ser efetuado usando-se o seguinte dispositivo prático:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | $q_1$ | $q_2$ | $q_3$ | | | $q_n$ | $q_{n+1}$ |
| $a$ | $b$ | $r_1$ | $r_2$ | ... | $r_{n-2}$ | $r_{n-1}$ | $r_n$ |
| $r_1$ | $r_2$ | $r_3$ | | | $r_n$ | 0 | |

Observe que o ***mdc* (*a,b*)** é o **último resto não nulo** do processo das divisões sucessivas.

# Máximo Divisor Comum

```
int gcd(int a, int b){
    if (a == 0)
        return b;
    return gcd(b % a, a);
}
```

# Mínimo Múltiplo Comum

- Problema: encontrar o menor múltiplo comum entre um par de inteiros.
- Para encontrar o mmc(x, y), podemos calcular o mdc(x, y) e utilizar a seguinte fórmula:

  mmc(x, y) * mdc(x, y) = x * y

  Ou seja:

  mmc(x, y) = x * y / mdc(x, y)

# Mínimo Múltiplo Comum

```
int lcm(int a, int b){
    return a * (b / gcd(a, b));
}
```

# Equações diofantinas

- Podemos definir uma equação diofantina linear como uma equação da forma

$$a_1x_1 + a_2x_2 + \dots + a_nx_n = c$$

sendo $a_1, \dots, a_n$ coeficientes inteiros não nulos, $x_1, \dots, x_n$ as variáveis **inteiras** a serem determinadas e c uma constante inteira.

- Diversos problemas podem ser modelados usando equações diofantinas. Em especial, vamos nos preocupar com equações diofantinas de duas variáveis

$$ax + by = c$$

# Equações diofantinas

- Exemplo de Problema: **Fantastic Beasts** (Final da Maratona SBC de Programação - 2018)

- Resumindo: considere um **grafo direcionado** em que os vértices representam zoológicos, e cada zoológico aponta para apenas para um outro zoológico (grau de saída = 1). Temos animais espalhados por esses zoológicos, e **a cada unidade de tempo todos os animais avançam para o próximo** zoológico.
- Objetivo: determinar **onde e quando TODOS os animais se encontrarão** no mesmo zoológico ao mesmo tempo (se isso puder ocorrer em diversos momentos e locais, determinar o primeiro deles, o mais cedo possível)

# Equações diofantinas

- Supondo que já estamos em uma fase um pouco mais avançada no problema, onde conseguimos modelar para cada **zoológico z** uma **equação** que determina os **momentos** em que um **animal a** passa por lá (os animais vão acabar presos em ciclos).

$$t = t_0 + k.i$$

# Equações diofantinas

- Se quisermos saber quando que os animais $a_1$ e $a_2$ se encontram no zoológico z, temos:

  Para animal $a_1$: $t^1 = t^1_0 + k_1 i$
  Para animal $a_2$: $t^2 = t^2_0 + k_2 j$
  Como queremos saber o momento de encontro, $t^1 = t^2$

$$t^1_0 + k_1 i = t^2_0 + k_2 j$$

$$\mathbf{k_1 i - k_2 j = t^2_0 - t^1_0}$$

  Equação diofantina com $a = k_1$, $b = -k_2$, $c = t^2_0 - t^1_0$ e com variáveis i,j

# Equações diofantinas

- Proposição 1: **$ax + by = c$** admite solução see **$\gcd(a,b) \mid c$**
- =>
  - Sendo $(x_0, y_0)$ uma solução da equação
  - Seja $d = \gcd(a,b)$, então $d \mid a$ e $d \mid b$. Logo podemos reescrever $a = Ad$ e $b = Bd$

$$c = ax_0 + by_0 = (Ad)x_0 + (Bd)y_0 = d(Ax_0 + By_0)$$

Denotando $q = Ax_0 + By_0$

$$c = dq$$

Portanto, $d \mid c$

# Equações diofantinas

- Proposição 1: **$ax + by = c$** admite solução see **gcd(a,b) | c**
- <=
  - Seja $d = \gcd(a,b)$
  - Pelo Teorema de Bézout, existe solução $(x_0, y_0)$ para $ax + by = d$
  - Por hipótese, $d \mid c \Rightarrow \exists\, t \,/\, c = dt$

$$c = dt$$
$$c = (ax_0+by_0)t$$
$$c = a(x_0t) + b(y_0t)$$

Portanto, se $d \mid c$, então a equação $ax + by = c$ admite solução

# Equações diofantinas

- Como determinar uma solução?

1. Obter uma solução $(x_0, y_0)$ para $ax + by = gcd(a,b)$
2. Para $ax + by = c$:
   a. $t = c/d$ em que $d = gcd(a,b)$
   b. $x = x_0 t$
   c. $y = y_0 t$
3. Se uma equação diofantina tem uma solução, então ela tem infinita:

$$S = \left\{ \left( x + \frac{b}{d}k, y - \frac{a}{d}k \right), k \in \mathbb{Z} \right\}$$

# Equações diofantinas

- Solução para ax + by = gcd(a,b)

1) Caso base :

Se $a = 0 \rightarrow by = gcd(0, b)$

Pelo Algoritmo de Euclides: $gcd(0, b) = b$

Então $by = b \rightarrow y = 1$

$x$ pode assumir qualquer valor, como queremos uma solução qualquer, faremos $x = 0$

Solução base: $(0, 1)$

# Equações diofantinas

- Solução para ax + by = gcd(a,b)

2) Passo da indução :

$$ax + by = gcd(a, b)$$

Pelo Algoritmo de Euclides, sabemos que

$$gcd(a, b) = gcd(b\%a, a) = c,$$

logo:

$$ax + by = c = (b\%a)x_1 + ay_1 \ (*)$$

# Equações diofantinas

- Solução para ax + by = gcd(a,b)

Considerando o resultado da divisão inteira, podemos dizer que :

$$b = \frac{b}{a}a + b\%a$$

$$b\%a = b - \frac{b}{a}a$$

Substituindo em $(*)$

$$\left(b - \frac{b}{a}a\right)x_1 + ay_1 = c$$

$$bx_1 - \frac{b}{a}ax_1 + ay_1 = c$$

$$a\underbrace{\left(y_1 - \frac{b}{a}x_1\right)}_{\text{x}} + b\underbrace{x_1}_{\text{y}} = c$$

# Equações diofantinas

```
\\Implementação:
int gcd(int a, int b, int &x, int &y){
    if (b == 0){
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a/b);
    return d;
}
```

# Equações diofantinas

```
bool solve(int a, int b, int c, int &x0, int &y0, int &g) {
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) {
        return false;
    }

    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}
```

# Aritmética Modular

- Em vários problemas precisamos operar com os restos de divisões de inteiros.
- A aritmética modular permite fazer cálculos com restos de divisões de modo eficiente, e é especialmente útil quando estamos trabalhando com números grandes (BigInteger).
- Na verdade, a Aritmética Modular pode nos ajudar a evitar ter que trabalhar com números muito grandes.

# Aritmética Modular

- A aritmética modular se baseia nas seguintes propriedades:

```
(x + y) % n = ((x % n) + (y % n)) % n
(x - y) % n = ((x % n) - (y % n)) % n
(x * y) % n = ((x % n) * (y % n)) % n
(x ^ y) % n = ((x % n) ^ y) % n
```

# Aritmética Modular

- [UVa 374 - Big Mod]
    - `Calcule R := B`$^P$` mod M`
    - `0 <= B, P <= 2147483647 e 1 <= M <= 46340`

# Aritmética Modular

```
long long pow(long long x, long long y, long long mod) {
    if (y == 0)
        return 1;

    long long p = pow(x, y/2, mod);
    if (y % 2 == 0)
    {
        return (p * p) % mod;
    }
    else
    {
        return (((p * p) % mod) * (x % mod)) % mod;
    }
}
```

# Referências


Biblioteca de códigos de Thiago Alexandre Domingues de Souza.

Matemática Discreta e Suas Aplicações. Kenneth H. Rosen.

Programming Challenges: The Programming Contest Training Manual. Stevem S. Skiena e Miguel A. Revilla.

https://www.geeksforgeeks.org/sieve-of-eratosthenes/

http://www.lcad.icmc.usp.br/~jbatista/scc210/AulaTeoriadosNumeros1.pdf

http://www.lcad.icmc.usp.br/~jbatista/scc210/AulaTeoriadosNumeros2.pdf

https://www.ufsj.edu.br/portal2-repositorio/File/comat/tcc_Ricardo.pdf

https://cp-algorithms.com/algebra/linear-diophantine-equation.html

# Análise Combinatória

Laboratório de Programação Competitiva - 2020

Pedro Henrique Paiola (paiola@fc.unesp.br)
Giulia Moura Crusco (giulia@fc.unesp.br)
João Pedro Marin Comini (joaocomini@gmail.com)
**Unesp Bauru**

# Análise Combinatória

- Combinatória, o estudo dos arranjos dos objetos, é uma parte importante da matemática discreta. É o ramo da matemática que se dedica à contagem de elementos ou eventos discretos e de suas possíveis combinações.
- Diversos problemas de programação competitiva envolvem análise combinatória, muitas vezes associado à programação dinâmica.

# Análise Combinatória

- Diversos problemas de contagem e de combinação possuem **soluções fechadas**, ou seja, existem **fórmulas matemáticas** resultantes da análise combinatória que podem ser aplicadas.
- Este é um dos motivos da importância da análise combinatória para a **Computação**, pois permite substituir um algoritmo com complexidade alto (busca por *backtracking*, por exemplo), por uma única chamada a uma simples fórmula.

# Análise Combinatória

- Em Programação Competitiva, isto é particularmente importante, permitindo resolver problemas aparentemente complexos de forma bastante simples, e sem estourar o tempo limite.
- Em alguns casos, também é possível a obtenção de *look-up tables* para soluções *off-line*.

# Análise Combinatória

- Bases da contagem:
  - **Regra do Produto**: Suponha que um procedimento possa ser dividido em uma sequência de duas tarefas. Se houver **n** formas de fazer a primeira tarefa **e**, para cada uma dessas formas, há **m** formas de fazer a segunda, então há **n.m** formas de concluir o procedimento.

Exemplo: Quantidade de números de 3 dígitos que podem ser formados apenas com os algarismos 1, 2, 5 e 7.

Devemos preencher 3 dígitos escolhendo dentro de 4 algarismos: 4 * 4 * 4 = 64 possibilidades

Se não pudesse haver repetição de algarismos: 4 * 3 * 2 = 24 possibilidades

# Análise Combinatória

- Bases da contagem:
  - **Regra do Soma**: Se uma tarefa puder ser realizada em uma de **n** formas **ou** em uma das **m** formas, em que nenhuma das n formas seja igual a alguma das m formas, então há **n + m** formas de realizar a tarefa.
  - Caso mais geral: quando há intersecção entre os conjuntos de “formas”, devemos subtraí-la da soma:

$$|A_1 \cup A_2| = |A_1| + |A_2| - |A_1 \cap A_2|$$

# Análise Combinatória

- **Contagem de associações:**
    - Uma associação é um arranjo de **n** itens, onde cada item pode ser escolhido de uma lista de **m** valores, com repetição.
    - Por exemplo, quantas formas diferentes existem de se pintar 4 casas utilizando 3 cores.
    - Utilizando a regra do produto:

$$S(n, m) = m^n$$

- Existem S(4,3) = $3^4$ = 81 associações possíveis entre 4 casas e 3 cores

# Análise Combinatória

- **Contagem de associações:**
  - Caso específico: **subconjuntos.** Quantos subconjuntos podemos formar a partir de um conjunto de **n** elementos? Trata-se de um problema de seleção sem reposição.
  - A seleção ou não de cada um dos n elementos pode ser representada de forma binária (selecionar ou não selecionar): arranjo binário de n posições.
  - Logo, o número de possíveis subconjuntos é $S(n, 2) = 2^n$

| Elementos: | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | ... | $x_n$ |
|---|---|---|---|---|---|---|---|
| Selecionar ou não | 1 | 1 | 0 | 0 | 1 | ... | 0 |

# Análise Combinatória

- **Contagem de associações:**
  - Exemplo 1: Um computador de 32 bits é capaz de endereçar quantos gigabytes de memória?

$$S(32, 2) = 2^{32} = 2^2 \,.\, 2^{30} = 4GB$$

  - Exemplo 2: Quantas senhas diferentes é possível criar utilizando de 8 a 10 letras ou dígitos, considerando letras minúsculas e maiúsculas.

$$S(8, 62) + S(9, 62) + S(10, 62) = 852.836.452.414.603.776$$

# Análise Combinatória

- **Permutação**: é um arranjo de n itens, onde cada item aparece exatamente uma única vez.
- O 1º elemento do arranjo pode assumir qualquer um dos n itens, o 2º pode assumir n-1 itens (qualquer um, exceto o já assumido pelo 1º) e assim por diante.
- Logo, pela regra do produto:

$$P(n) = n(n-1)(n-2)\ldots 1 = n!$$

$$P(n) = n!$$

# Análise Combinatória

- **Permutação**
  - Exemplo: anagramas
  - Quantos anagramas existem da palavra MESA
  - Conjunto de elementos: {M, E, S, A}
  - $P(4) = 4! = 24$ anagramas

# Análise Combinatória

- **Permutação com elementos repetidos**
  - Exemplo: anagrama da palavra CASA
  - A letra “A” aparece duas vezes, se aplicássemos a fórmula da permutação, o anagrama ACSA, por exemplo, seria contado duas vezes, como se cada letra A fosse uma letra diferente. CASA => ACSA, ACSA
  - Considerando um conjunto de elementos, e que o elemento 1 se repete $n_1$ vezes, o 2 se repete $n_2$ vezes, e assim por diante, chegamos em:

$$P^{n_1,\ldots,n_n}(n) = \frac{n!}{n_1!n_2!\ldots n_3!}$$

# Análise Combinatória

- **Arranjo**: quantas possibilidades há de escolher **r** elementos de um conjunto de **n** elementos, em que a ordem de escolha é relevante?
- É uma generalização da permutação. Uma permutação pode ser considerada como um arranjo em que r = n

$$P(n, r) = \frac{n!}{(n-r)!}$$

# Análise Combinatória

- **Combinação**: quantas possibilidades há de escolher **r** elementos de um conjunto de **n** elementos, em que a ordem de escolha **não** é relevante?

$$C(n,r) = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

- Tanto no caso da permutação como da combinação, temos que tomar um pouco de cuidado com nossa implementação (estamos usando muito fatorial), tanto em relação ao tempo quanto ao limite de nossas variáveis.

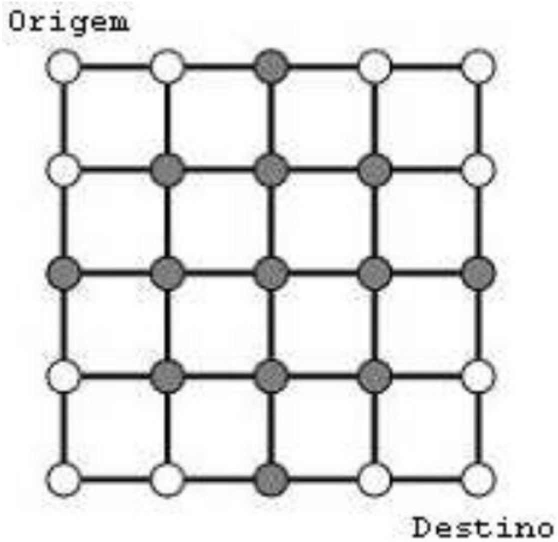
# Análise Combinatória

- Para a combinação, pode-se calcular qualquer coeficiente binomial baseado na seguinte recorrência (que deriva o conhecido Triângulo de Pascal):

$$\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}$$

# Análise Combinatória

- **Exemplo: caminho através de uma grade**
  - Quantas formas temos de caminhar em uma grade n x m a partir do canto superior esquerdo e alcançar o canto inferior direito caminhando apenas para baixo e para a direita?



# Análise Combinatória

- **Exemplo: caminho através de uma grade**
  - 1ª forma de analisar: note que cada caminho é necessariamente constituído de um conjunto de n + m passos, n para baixo e m para a direita.
  - Sendo assim, um caminho nada mais é do que uma permutação de passos para baixo e para a direita. Podemos considerar como se fosse um anagrama (com “letras” repetidas). Ex: BBDDB. Logo, pela fórmula da permutação com elementos repetidos:

$$P^{n,m}(n+m) = \frac{(n+m)!}{n!m!}$$

# Análise Combinatória

- **Exemplo: caminho através de uma grade**
  - 2ª forma de analisar: novamente, considerando que um caminho é constituído de n + m passos, n para baixo e m para a direita.
  - Necessariamente, dois caminhos distintos diferem na ordem de um ou mais dos n passos para baixo, dentro dos n + m passos totais.
  - Exemplo, considerando uma grade 2x2:
    - **baixo** - direita - **baixo** - direita (ordens 1 e 3)
    - **baixo** - **baixo** - direita - direita (ordens 1 e 2)

# Análise Combinatória

- **Exemplo: caminho através de uma grade**
  - Dessa forma, podemos escolher **n** posições dentro das **n + m** possíveis como sendo passos para baixo. Se tratando de um problema de combinação, já que a ordem de escolha das posições não importam, as combinações (1,3) e (3,1) representam o mesmo caminho (pensando no exemplo anterior)

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

# Análise Combinatória

```
int PermutationCoeff(int n, int k)      //O(n)
{
    int Fn = 1, Fk;

    for (int i = 1; i <= n; i++)
    {
        Fn *= i;
        if (i == n - k)
           Fk = Fn;
    }
    int coeff = Fn / Fk;
    return coeff;
}
```

# Análise Combinatória

$$P(n,r) = \frac{n!}{(n-r)!} = \frac{n(n-1)\ldots(n-r-1)\cancel{(n-r)!}}{\cancel{(n-r)!}}$$

$$P(n,r) = n(n-1)\ldots(n-r-1)$$

# Análise Combinatória

```
int PermutationCoeff(int n, int k)
{
    int coeff = 1;

    for (int i = n; i > (n - k); i--)
        coeff *= i;

    return coeff;
}
```

# Análise Combinatória

```
long long bin[MAXN][MAXR];

void calcularCoefBin(int n, int k){ //Pré-calculando em O(n*k)
    int i, j;
    for (i = 0; i <= n; i++){
        for (j = 0; j <= min(i, k); j++){
            if (j == 0 || j == i)
                bin[i][j] = 1;
            else
                bin[i][j] = bin[i - 1][j - 1] + bin[i - 1][j];
        }
    }
}
```

# Referências


Biblioteca de códigos de Thiago Alexandre Domingues de Souza.

Matemática Discreta e Suas Aplicações. Kenneth H. Rosen.

Programming Challenges: The Programming Contest Training Manual. Stevem S. Skiena e Miguel A. Revilla.

https://www.geeksforgeeks.org/permutation-coefficient/

http://wiki.icmc.usp.br/images/a/ac/SCC211Cap6A.pdf

https://mundoeducacao.uol.com.br/matematica/permutacao-envolvendo-elementos-repetidos.htm

https://brasilescola.uol.com.br/matematica/permutacao-com-elementos-repetidos.htm