maratona de programação

UNIFEI

9 – Matemática

Prof. **João Paulo** R. R. Leite
joaopaulo@**unifei**.edu.br
ECOE44 & ECOE45 – Maratona de Programação
sites.google.com/site/unifeimaratona/

Estudaremos três tópicos que são de maior importância em problemas matemáticos:

- Matemática Básica
- Teoria dos Números
- Análise Combinatória

Normalmente, **pelo menos um problema** da competição envolverá algum tipo de solução matemática, onde utilizar um pouco de matemática antes da codificação pode simplificar e reduzir o código.

Encontrando padrões e fórmulas

Alguns problemas possuem soluções que formam algum tipo de padrão. Ao encontrarmos um padrão, solucionamos o problema.

Este tipo de problema pode ser classificado como **matemático *Ad-Hoc*** e requer um pouco de intuição matemática para ser resolvido. Algumas dicas são:

- Resolva algumas instâncias pequenas a mão;
- E veja se as soluções formam algum padrão.
- O padrão envolve algum tipo de sobreposição? Talvez possamos utilizar PD.

Alguns padrões recorrentes...

Com certa frequência, vemos padrões como este:

2, 5, 8, 11, 14, 17, 20...

Que tipo de padrão é este?

Este tipo de padrão é uma Progressão Aritmética, cuja fórmula base é:

$$a_n = a_{n-1} + c$$

Alguns padrões recorrentes...

Quando temos uma P.A., existem alguns tipos clássicos de dados que são pedidos como solução de um problema:

1) É necessário calcular o enésimo elemento:

$$a_n = a_1 + (n-1)c$$

2) É necessário calcular a soma de todos os elementos em uma porção definida da progressão:

$$S_n = \frac{n(a_1 + a_n)}{2}$$

Alguns padrões recorrentes...

Vemos, também, com certa frequência, padrões como:

1, 2, 4, 8, 16, 32, 64, 128...

Que tipo de padrão é este?

Este tipo de padrão é uma Progressão Geométrica, que pode ser, mais genericamente expressa por:

$$a, ar, ar^2, ar^3, ar^4, ar^5, ar^6, \ldots$$

$$a_n = ar^{n-1}$$

Alguns padrões recorrentes...

Quando temos uma P.G., podemos também calcular algumas coisas através de fórmulas:

1) Soma de elementos:

$$\sum_{i=0}^{n} ar^i = \frac{a(1-r^n)}{(1-r)}$$

2) Soma de elementos de m até n.

$$\sum_{i=m}^{n} ar^i = \frac{a(r^m - r^{n+1})}{(1-r)}$$

Um pouco sobre logaritmos e exponenciação

Realizar alguns cálculos utilizando logaritmos pode ser uma alternativa eficientes para alguns problemas. Em C++, temos funções para:

- Logaritmo natural: `double log(double x);`
- Logaritmo na base 10: `double log10(double x);`
- Exponencial: `double exp(double x);`

Todas as funções estão definidas em <**cmath**>

Um pouco sobre logaritmos e exponenciação

Exemplo:

Qual a primeira potência de 17 que contém k dígitos na base b?

Solução simples, ou ingênua (*naive*): Passar pelas potências de 17 e contar o número de dígitos.

Qual o problema? Potências de 17 crescem exponencialmente! $17^{16} > 2^{64}$

E se k == 500? (~1.7×10^{615}), ou ainda maior? Fica impossível de se trabalhar com números de maneira normal. Portanto, porque não utilizar log?

Um pouco sobre logaritmos e exponenciação

Lembre-se de que podemos calcular o comprimento de um número n na base b utilizando $\lfloor log_b(n) + 1 \rfloor$.

Mas como faremos isso tendo apenas ln e log10?

Mudando de base! Veja:

$$log_b(a) = \frac{log_d(a)}{log_d(b)} = \frac{\ln(a)}{\ln(b)}$$

Agora podemos, pelo menos, contar o comprimento sem ter que converter bases.

Um pouco sobre logaritmos e exponenciação

Ainda teremos que percorrer as potências de 17, mas podemos fazê-lo em log:

$$\ln(17^{x-1} \cdot 17) = \ln(17^{x-1}) + \ln(17)$$

Utilizando-se de algumas propriedades:

$$\log_b(xy) = \log_b(x) + \log_b(y)$$

$$\log_b(a^c) = c \cdot \log_b(a)$$

$$\log_b(\frac{x}{y}) = \log_b(x) - \log_b(y)$$

Um pouco sobre logaritmos e exponenciação

Podemos simplificar ainda mais: a solução de nosso problema é, em termos matemáticos, encontrar c para:

$$\log_b(17^x) = k - 1$$

Utilizando as propriedades anteriores, temos que:

$$x = \left\lceil (k-1) \cdot \frac{\ln(10)}{\ln(17)} \right\rceil$$

Que é muito melhor do que verificar um a um.

E por falar em bases...

O que precisamos para realizar conversões de base? Veja um algoritmo simples que transforma números da base 10 para qualquer outra base:

```
vector<int> to_base(int base, int val)
{
    vector<int> res;
    while(val) {
        res.push_back(val % base);
        val /= base;
    }
    return res;
}
```

Este algoritmo coloca em um vetor **res**, o valor convertido, de forma inversa. **Ex:** Para uma conversão do valor 32 para a base 16, ficaríamos com

res[0] = **0**
Res[1] = **2**

Trabalhando com *doubles*

Comparar valores do tipo double pode não ser a melhor das ideias, dado que sua precisão é grande e muitas vezes gostaríamos de comparar apenas até certo grau de precisão apenas (digamos 10^{-9}).

Podemos, portanto, definir que dois números reais são iguais se sua diferença for menor que algum **épsilon** (como um erro aceitável), que nós mesmos definimos. Veja:

```
const double EPS = 1e-9;

if(abs(a-b) < EPS) { // se a e b forem "iguais"
    // ...
}
```

Trabalhando com *doubles*

Podemos fazer o mesmo tipo de solução para:

- Operador menor:

```
if(a < b - EPS) {
    // ...
}
```

- Operador menor ou igual:

```
if(a < b + EPS) {
    //...
}
```

- Etc.

Algumas definições que todos deveríamos saber:

- Número primo: é um inteiro positivo maior que 1 que não possui nenhum divisor positivo diferente de 1 e ele mesmo.
- Máximo Divisor Comum (MDC): O MDC de dois inteiros a e b é o maior número que divide tanto a quanto b.
- Mínimo Múltiplo Comum (MMC): O MMC de dois inteiros a e b é o menor inteiro que tanto a e b dividem.
- Fator primo de um inteiro é um número primo que o divide.
- Fatoração prima: É a decomposição de um inteiro em seus fatores primos. Pelo teorema fundamental da aritmética, cada inteiro > 1 tem uma fatoração prima única.

Algoritmo Euclidiano

O Algoritmo Euclidiano é um algoritmo recursivo que calcula o MDC de dois números. Seu tempo de execução é $O(\log^2 N)$. Veja:

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

Este algoritmo também calcula o MMC, uma vez que é possível provar que:

$$mmc(a,b) = \frac{a.b}{mdc(a,b)}$$

Algoritmo Euclidiano Estendido

Ao reverter os passos do algoritmo euclidiano, temos a identidade de Bézout:

$$mdc(a, b) = ax + bx$$

que simplesmente define que sempre existem x e y tal que a equação acima seja verdadeira.

O algoritmo euclidiano estendido calcula o MDC e os coeficientes x e y. Veja:

```
int egcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    } else {
        int d = egcd(b, a % b, x, y);
        x -= a / b * y;
        swap(x, y);
        return d;
    }
}
```

Entre outras aplicações, este algoritmo é essencial no **algoritmo RSA** de criptografia de dados.

Teste de primalidade:

Um teste de primalidade é um algoritmo para **determinar se um dado número inteiro é primo**. Este tipo de teste é usado em áreas da matemática como a criptografia. Diferentemente da fatoração de inteiros, os testes de primalidade geralmente não fornecem os fatores primos, indicando apenas se o número fornecido é ou não primo.

Existem alguns métodos para se realizar esta verificação:

Teste de primalidade:

1) Método ingênuo: Passar por todos 1 < i < n e checar se n é divisível por i. Roda em O(N).
2) Um pouco melhor: Se n não for primo, ele possui um divisor $\leq \sqrt{n}$. Portanto, podemos iterar apenas até $\sqrt{n}$. Roda em O($\sqrt{n}$).
3) Ainda melhor: Se n não for primo, ele tem um divisor primo $\leq \sqrt{n}$. Podemos iterar pelos números primos até $\sqrt{n}$. Existem aproximadamente N/ln(N) primos menores que N. Portanto, roda em O($\sqrt{n}$/logN)

Geração de números primos

Se quisermos gerar números primos, utilizar o teste de primalidade é muito ineficiente. Ao invés disso, nosso método preferido será o Crivo de Erastóstenes.

Como funciona:

- Para todos os números entre 2 e $\sqrt{n}$:
- Se o número não estiver marcado, itere por cada múltiplo do número até n e os marque.
- Os números não marcados são aqueles que não são múltiplos de nenhum número menor.
- Roda em O($\sqrt{N}\log\sqrt{N}$).

	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

Veja esta **simulação** e, em seguida, o **código**.

```
vector<int> primes;

void erastostenes(int n)
{
    vector<bool> is_prime(n, true);
    for(int i = 2; i < n; ++i) {
        if(is_prime[i]) {
            primes.push_back(i);
            for(int j = 2*i; j < n; j += i)
                is_prime[j] = false;
        }
    }
}
```

Encontra todos os números primos menores que um inteiro n dado.

Fatoração de números inteiros

O teorema fundamental da aritmética define que cada inteiro maior que 1 é uma multiplicação única de números inteiros:

$$n = p_1^{e1} p_2^{e2} p_3^{e3} \ldots p_k^{ek}$$

Para fatorar um inteiro n, temos que:

- Utilizar o Crivo de Erastóstenes para gerar todos os números primos até $\sqrt{n}$.
- Iterar por todos os primos gerados, checando se eles dividem n, e determinando a maior potencia que divide n.

```cpp
map<int, int> factor(int N) {
    vector<int> primes;
    primes = eratosthenes(static_cast<int>(sqrt(N+1)))
    map<int, int> factors;
    for(int i = 0; i < primes.size(); ++i){
        int prime = primes[i], power = 0;
        while(N % prime == 0){
            power++;
            N /= prime;
        }
        if(power > 0){
            factors[prime] = power;
        }
    }
    return factors;
}
```

Revisão de Análise Combinatória

Combinação de n elementos em conjuntos de k. Em combinações, não importa a ordem dos elementos, apenas os integrantes:

$$C_{n,k} = \frac{A_{n,k}}{P_k} = \frac{n!}{k!\,(n-k)!} = \binom{n}{k}$$

Outro conceito importante é o de arranjos, que são conjuntos de tamanho k formados a partir de n elementos. A diferença para a combinação, é que nos arranjos a ordem é importante:

$$A_{n,k} = \frac{n!}{(n-k)!}$$

Revisão de Análise Combinatória

A permutação é um arranjo de n elementos entre si, ou seja:

$$P_n = A_{n,n} = \frac{n!}{(n-n)!} = n!$$

Quando existem elementos repetidos em permutações (como permutações com as letras da palavra MATEMATICA), existe outra fórmula:

$$P^* = \frac{n!}{a!\, b!\, g! \ldots l!}$$

Onde a, b, g... L são as vezes que os elementos se repetem.

Revisão de Análise Combinatória

A permutação circular, ou seja, quando o arranjo não tem começo ou fim (uma mesa redonda, por exemplo) é dada por:

$$P_c(n) = (n-1)!$$

Arranjos com repetição, onde pode haver mais de um dado de cada:

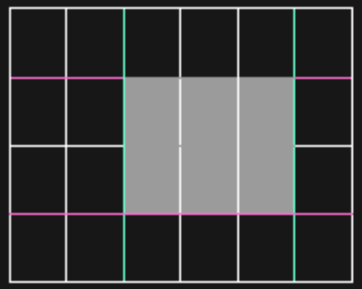
$$A^*_{n,k} = n^k$$

E, finalmente, combinações com repetição:

$$C^*_{n,k} = C_{n+k-1,k} = \binom{n+k-1}{k}$$

Example

How many rectangles can be formed on a $m \times n$ grid?

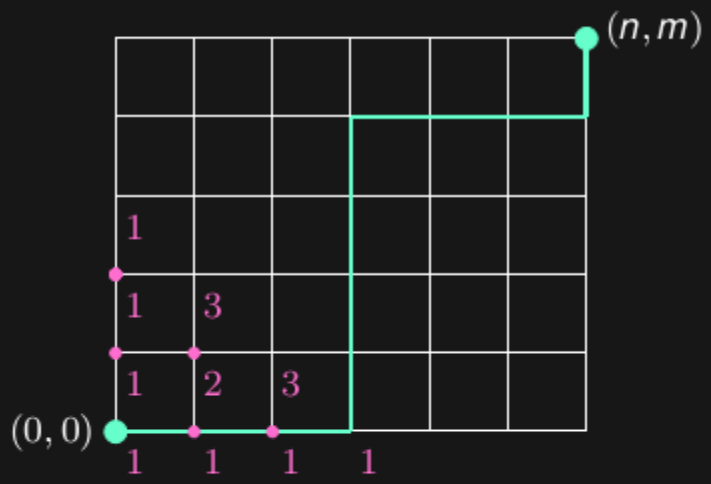


- A rectangle needs 4 edges, 2 vertical and 2 horizontal.
  - 2 vertical
  - 2 horizontal
- Total number of ways we can form a rectangle

$$\binom{n}{2}\binom{m}{2} = \frac{n!m!}{(n-2)!(m-2)!2!2!}$$
$$= \frac{n(n-1)m(m-1)}{4}$$

Binomial coefficients

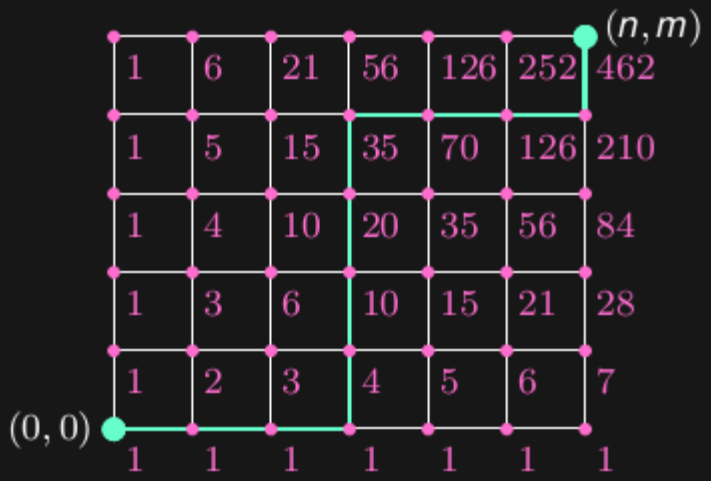
How many different lattice paths are there from $(0,0)$ to (n,m)?



- There is 1 path to $(0,0)$
- There is 1 path to $(1,0)$ and $(0,1)$
- Paths to $(1,1)$ is the sum of number of paths to $(0,1)$ and $(1,0)$.
- Number of paths to (i,j) is the sum of the number of paths to $(i-1,j)$ and $(i,j-1)$.

Binomial coefficients

How many different lattice paths are there from $(0,0)$ to (n,m)?



- There is 1 path to $(0,0)$
- There is 1 path to $(1,0)$ and $(0,1)$
- Paths to $(1,1)$ is the sum of number of paths to $(0,1)$ and $(1,0)$.
- Number of paths to (i,j) is

$$\binom{i+j}{i}$$

Thanks to
Tómas Ken Magnússon
Bjarki Ágúst Guðmundsson
From School of Computer Science
Reykjavík University, Iceland

This material has been based on their original work.