

Programação Competitiva

Vamos começar apresentando alguns dos recursos da linguagem de programação C++ que são úteis na programação competitiva. Todos os comandos serão dados considerando o Linux como Sistema Operacional. Caso esteja usando Windows, use o [CS50 IDE](#) ou [VSCode for CS50](#).

Características da linguagem

```
1 #include <bits/stdc++.h> // ①
2
3 using namespace std; // ②
4
5 int main() {
6     // código
7     return 0;
8 }
```

1. Recurso do compilador `g++` que permite incluir toda a biblioteca padrão. Assim, não é necessário incluir separadamente bibliotecas como `iostream`, `vector`, e `algorithm`. [Clique aqui para saber mais.](#)
2. Declara que as classes e funções da biblioteca padrão podem ser usadas diretamente no código. Sem essa linha teríamos que escrever, por exemplo, `std::cout` ao invés de apenas `cout`.

O código pode ser compilado usando o seguinte comando:

```
g++ main.cpp -o programa
```

O comando produz um arquivo binário, chamado `programa`, a partir do código-fonte `main.cpp`. Leia mais [aqui](#).

Para evitar bugs comuns, **sempre** iremos compilar o código com algumas *flags* de compilação:

```
g++ -O2 -std=c++17 -Wshadow -fsanitize=address,undefined -Wall -Wextra -Wno-sign-compare -Wno-unused-parameter -Wno-unused-variable -Wno-unused-but-set-variable main.cpp -o programa
```

Para facilitar a compilação e execução do código, crie um arquivo chamado `cr` e adicione o seguinte código:

```
g++ -O2 -std=c++17 -Wshadow -fsanitize=address,undefined -Wall -Wextra -Wno-sign-compare -Wno-unused-parameter -Wno-unused-variable -Wno-unused-but-set-variable $1 -o programa && time ./programa < in
```

No terminal, execute o comando `chmod +x cr` para dar direitos de execução ao arquivo `cr`. Para usá-lo basta passar como argumento um arquivo `.cpp`, por exemplo, `./cr main.cpp`. O código será compilado e, em caso de sucesso, será gerado um programa chamado `programa` que será executado considerando o arquivo `in` como dado de entrada. Ao usar o arquivo `in`, não precisamos digitar os dados de entrada.

Entrada e Saída

Na maioria dos contests, é necessário ler da entrada padrão (teclado) e escrever algo. Em C++, é usado o `cin` para leitura e `cout` para saída. Também podem ser usados as funções de C, como `scanf` e `printf`.

A entrada do programa geralmente consiste em números e strings separados por espaços e/ou novas linhas. Eles podem ser lidos a partir do `cin` da seguinte forma:

```
1 int a, b;
2 string c;
3 cin >> a >> b >> c;
```

Considerando que há ao menos um espaço em branco ou uma nova linha entre cada elemento da entrada, esse código sempre funciona.

O `cout` pode ser usado da seguinte forma:

```
1 cout << a << " " << b << " " << c << "\n"; // ①
```

1. Um espaço em branco irá separar cada informação. Ao fim, uma linha em branco (`\n`) será gerada.

Às vezes, a entrada e a saída podem ser um gargalo em um programa. Por isso, é comum ser adicionado as seguintes linhas no início do código:

```
1 ios_base::sync_with_stdio(0); // ①
2 cin.tie(0); // ②
```

1. `std::ios_base::sync_with_stdio`
2. `std::ios::tie`

Atenção

Ao usar o comando `ios_base::sync_with_stdio(0);`, será desativado a sincronização entre as função de C++ e C, por isso **não** use as função de entrada e saída de C (`scanf` e `printf`) junto com esse comando.

Alternativamente ao `\n` podemos usar o comando `endl`. Entretanto, este comando irá liberar o *buffer* de saída e fará com que o código rode mais lento. Por isso, prefira usar o `\n`.

Dica

Use sempre o `\n` ao invés do `endl`. Use uma macro `#define endl '\n'` para não correr o risco de esquecer.

Para se aprofundar mais:

[Input & Output](#)

[Fast Input & Output](#)

Trabalhando com números

- Inteiros: Para evitar *integer overflow*, use sempre `long long` (64bits) ao invés de `int`.
- Reais: Use `double` (64bits) ou `long double` (80bits). Esqueça o `float` 😊. Além disso, nunca compare dois `double` com o operador `==` (é possível que os valores sejam iguais, mas não são devido a erros de precisão). Para verificar se dois `double` use o código a seguir:

```
1 double a, b;
2 ...
3 if (abs(a-b) < 1e-9) {
4     //a e b são iguais
5 }
6 ...
```

Para saber mais: [Data Types](#)

Simplificando o código

[Nomes de tipos](#)

Usando o comando `typedef` é possível dar um nome mais curto a um tipo de dado. Por exemplo:

```

1  typedef long long ll;
2  typedef vector<int> vi;
3  typedef vector<ll> vll;
4  typedef pair<int,int> pii;
5  typedef pair<ll,ll> pll;
6  typedef vector<pii> vpi;
7  typedef vector<pll> vpll;
```

Macros

Uma macro significa que certas palavras no código serão substituídas antes da compilação. Em C++, as macros são definidas usando a palavra-chave `#define`. Veja alguns exemplos:

```

1 #define F first
2 #define S second
3 #define PB push_back
4 #define MP make_pair
5 #define FOR(i,a,b) for(ll i = (a); i < (ll)(b); ++i)
6 #define INF 0x3f3f3f3f
7 #define INFL 0x3f3f3f3f3f3f3f3f
8 #define all(x) x.begin(),x.end()
9 #define sz(x) (ll)x.size()
10 #define MOD 1000000007ll
11 #define endl '\n'
```

Assim, por exemplo, o código `for(long long i = 0; i < n; ++i)` pode ser simplificado por `FOR(i, 0, n)`.

Dicas e truques de C++

A seguir, são listados alguns links com dicas e truques de C++ úteis para programação competitiva. Leia **todos** com atenção:

- [C++ tips and tricks](#)
- [Top 20 C++ Tricks for Competitive Programming](#)
- [Truques de programação competitiva para programadores de C++](#)

Manipulação de Bits

Todos os dados em um programa de computador são internamente armazenados como números binários, ou seja, uma sequência de 0's ou 1's. Em C++ um número do tipo `int` é uma variável de 32-bits, ou seja, todo número `int` consiste de uma sequência de 32 0's ou 1's. Por exemplo, a representação do número `int` 43 é:

```
0000000000000000000000000000101011
```

Normalmente, é usada a representação de bits com sinal de um número, o que significa que números negativos e positivos podem ser representados. Por exemplo, o `int` de C++ é um tipo com sinal, logo uma variável desse tipo pode armazenar valores inteiros entre -2^{31} e $2^{31} - 1$. O primeiro bit de uma representação com sinal indica o sinal do número (0 para números não-negativos e 1 para números negativos). O **complemento de dois** é usado, o que significa que o oposto de um número é calculado primeiramente invertendo todos os bits do número e depois aumentando o número em um. Por exemplo, a representação do número `int` -43 é:

```
1111111111111111111111111010101
```

⚡ Atenção

Se um número for maior que o limite superior da representação de bits, ocorrerá um *overflow*. Considerando uma variável do tipo `int`, o próximo número depois de $2^{31} - 1$ é -2^{31} .

```
int v = 2147483647; // ①  
v++; // ②  
cout << v << "\n"; // -2147483648 ③
```

1. $011111111111111111111111111111_2$.
2. Deveria ser 2147483648_{10} , mas esse valor não pode ser representado em bits usando uma variável de 32-bits.
3. $10000000000000000000000000000000_2$ (lembre-se que é utilizado complemento de dois).

Operadores sobre Bits

Operador AND (`&` e `&=`)

Os bits são definidos como 1 no resultado, se os bits correspondentes em ambos os operadores forem 1. Exemplos:

```
a      = 5 // 00000101
b      = 9 // 00001001
a & b -> 1 // 00000001

c      = 10 // 00001010
d      = 12 // 00001100
c & d -> 8 // 00001000
```

Dica

Com o operador `&`, podemos verificar se um número `x` é par usando `x & 1`. De forma geral, `x` é divisível por 2^k se $x \& (2^k - 1) = 0$.

Operador OR inclusivo (`|` e `|=`)

Os bits são definidos como 1 no resultado, se pelo menos um dos bits correspondentes em ambos os operandos for 1. Exemplos:

```
a      = 5 // 00000101
b      = 9 // 00001001
a | b -> 13 // 00001101

c      = 10 // 00001010
d      = 12 // 00001100
c | d -> 14 // 00001110
```

Operador OR exclusivo (`^` e `^=`)

Os bits são definidos como 1 no resultado, se exatamente um dos bits correspondentes em ambos os operandos for 1. Exemplos:

```
a      = 5 // 00000101
b      = 9 // 00001001
a ^ b -> 12 // 00001100

c      = 10 // 00001010
d      = 12 // 00001100
c ^ d -> 6 // 00000110
```

Operador NOT (`~` e `~=`)

Produz um número onde todos os bits são invertidos, ou seja, todos os bits 0 são definidos como 1 e vice-versa. O resultado do operador NOT depende do tamanho da representação do

bit, pois a operação inverte todos os bits. Por exemplo, considerando um número do tipo `int` (32-bits), o resultado será:

```
a = 5 // 00000000 00000000 00000000 00000101
~a = -6 // 11111111 11111111 11111111 11111010
```

Dica

A fórmula `~x = -x - 1` é válida. Por exemplo, `~5 = 6`.

Operador de deslocamento à esquerda (`<<` e `<=`)

Desloca os bits do primeiro operando à esquerda pelo número de bits especificado pelo segundo operando (deve ser um valor positivo): preenche a partir da direita com zero (0).

Exemplos:

```
a = 1      // 00000001 -> 1
a = a << 1 // 00000010 -> 2
a = a << 1 // 00000100 -> 4
a = a << 1 // 00001000 -> 8
a = a << 1 // 00010000 -> 16

b = 7      // 00000111
b = b << 1 // 00001110 -> 14

c = 7      // 00000111
c <= 3     // 00111000 -> 56
```

Operador de deslocamento à direita (`>>` e `>=`)

Desloca os bits do primeiro operando à direita pelo número de bits especificado pelo segundo operando (deve ser um valor positivo): preenche a partir da esquerda com zero (0). Exemplos:

```
a = 16      // 00010000 -> 16
a = a >> 1 // 00001000 -> 8
a = a >> 1 // 00000100 -> 4
a = a >> 1 // 00000010 -> 2
a = a >> 1 // 00000001 -> 1
a = a >> 1 // 00000000 -> 0

b = 56      // 00111000
b = b >> 3 // 00000111 -> 7

c = 56      // 00111000
c >= 3     // 00000111 -> 7
```

Funções adicionais

O compilador `g++` fornece as seguintes funções para contar bits:

- `__builtin_clz(x)`: o número de zeros no início do número `int`;
- `__builtin_ctz(x)`: o número de zeros no final do número `int`;
- `__builtin_popcount(x)`: o número de uns no número `int`;
- `__builtin_parity(x)`: a paridade (par ou ímpar) de uns de um número `int`;

Veja um exemplo de utilização dessas funções:

```

1 int x = 5328; // 000000000000000000001010011010000
2 cout << __builtin_clz(x) << "\n"; // 19
3 cout << __builtin_ctz(x) << "\n"; // 4
4 cout << __builtin_popcount(x) << "\n"; // 5
5 cout << __builtin_parity(x) << "\n"; // 1

```

Embora as funções acima sejam apenas para números `int`, também existem versões das funções que suportam `long` e `long long` bastando adicionar o sufixo `l` ou `ll` no nome das funções.

- `__builtin_clzl(x)` ou `__builtin_clzll(x)`
- `__builtin_ctzl(x)` ou `__builtin_ctzll(x)`
- `__builtin_popcountl(x)` ou `__builtin_popcountll(x)`
- `__builtin_parityl(x)` ou `__builtin_parityll(x)`

Alguns truques e aplicações

Um número da forma `1 << k` tem um bit na posição `k` e todos os outros bits são zero, então esse número pode ser usado para acessar bits únicos de números. Em particular, o k -ésimo bit de um número é 1 exatamente quando `x & (1 << k)` não é zero. Por exemplo, para imprimir a representação de bits de um número `int` pode ser usado o seguinte código:

```

1 for (int i = 31; i >= 0; i--) {
2     if (x&(1<<i)) cout << "1";
3     else cout << "0";
4 }

```

Outras aplicações:

- `x = x | (1 << k)`: define o k -ésimo bit de `x` para 1;
- `x = x & ~(1 << k)`: define o k -ésimo bit de `x` para 0;

- `x = x ^ (1 << k)`: inverte o k -ésimo bit de `x`;
- `x = x & (x - 1)`: define o último bit 1 de `x` como zero;
- `x = x & -x`: define todos os bits 1 como 0, exceto o último bit 1;
- `x = x | (x - 1)`: inverte todos os bits após o último bit 1;
- `x & (x - 1)`: se `x` for um número positivo, verifica se `x` é uma potência de dois.

std::bitset

[Bitset](#) é um contêiner da Standard Template Library do C++ que representa uma sequência de tamanho fixo de N bits. Bitsets podem ser manipulados por operadores lógicos padrão e convertidos para inteiros ou strings. Exemplos:

```

1  bitset<8> a(42);           // 00101010
2  bitset<32> b(42);         // 0000000000000000000000000000101010
3  bitset<32> c("110010");   // 0000000000000000000000000000110010
4  int d = c.to_ulong();     // 50
5
6  bitset<4> b1("0110");
7  bitset<4> b2("0011");
8  cout << "b1 & b2: " << (b1 & b2) << '\n'; // b1 & b2: 0010
9  cout << "b1 | b2: " << (b1 | b2) << '\n'; // b1 | b2: 0111
10 cout << "b1 ^ b2: " << (b1 ^ b2) << '\n'; // b1 ^ b2: 0101
11
12 bitset<8> e; //00000000
13 e.set();      //11111111
14 e.reset();    //00000000
15 e.set(3);    //00001000
16 e.set(3, 0); //00000000
17 e.flip(0);   //00000001
18 e.flip(1);   //00000011
19 e.flip(7);   //10000011
20 e.flip();    //01111100
21
22 bitset<8> f{0b01110010};
23 cout << f << "\n";        //01110010
24 cout << (f>>=1) << "\n"; //00111001
25 cout << (f>>=1) << "\n"; //00011100
26 cout << (f>>=2) << "\n"; //00000111
27 cout << (f<<=3) << "\n"; //00111000
28 cout << f.count() << "\n"; // 3

```

Explore mais o contêiner através da [documentação](#).

Material complementar

- [Bitwise Operations tutorial #1 | XOR, Shift, Subsets](#)
- [C++ Bitsets in Competitive Programming](#)

- C Bitwise Operators: AND, OR, XOR, Complement and Shift Operations
- Bitwise Operators in C/C++ - GeeksforGeeks
- Bits manipulation (Important tactics) - GeeksforGeeks
- Bitwise Hacks for Competitive Programming - GeeksforGeeks
- Bit Tricks for Competitive Programming - GeeksforGeeks
- Bitwise Algorithms - GeeksforGeeks
- Builtin functions of GCC compiler - GeeksforGeeks
- Bit Manipulation | HackerEarth
- Manipulação de Bits | Neps Academy
- C++ bitset and its application

Teoria dos Números¹

Teoria dos Números é um ramo da matemática que estuda os números inteiros. Dominar o maior número possível de tópicos da teoria dos números é importante, pois alguns problemas matemáticos se tornam fáceis (ou mais fáceis) se você conhecer a teoria por trás do problema.

Divisibilidade

Um inteiro n é divisível por um inteiro d (denotado por $d|n$) se houver outro inteiro q tal que $n = d \times q$. Também é dito que d é um divisor de n . Dividindo os dois lados da igualdade $n = dq$ por d tem-se uma definição quase equivalente, ou seja, que $\frac{n}{d}$ é um inteiro.

Exemplo

O número 12 possui 6 divisores:

1 ($1 \times 12 = 12$), 2 ($2 \times 6 = 12$), 3 ($3 \times 4 = 12$), 4 ($4 \times 3 = 12$), 6 ($6 \times 2 = 12$) e 12 ($12 \times 1 = 12$).

O conceito de divisibilidade traz muitas questões. A primeira é como verificar se um número é divisível por outro. Para números pequenos, que podem ser armazenados em variáveis, por exemplo, do tipo `long long`, pode-se usar o operador módulo ou resto da divisão (%): n é divisível por d se e somente se $n \% d == 0$. Entretanto, para números inteiros grandes a solução não é tão simples. Na Seção [Aritmética Modular](#) será discutido como implementar o operador módulo para números inteiros grandes.

Outra questão é como calcular os divisores de um número. Todo número n tem pelo menos dois divisores (1 e n). Para encontrar os outros divisores, pode-se usar o fato que qualquer divisor d de n deve satisfazer $|d| \leq |n|$. Assim, pode-se testar se os inteiros entre 1 e n são divisores de n , ou seja, um algoritmo $O(n)$. Entretanto, sempre que tem-se um divisor d , tem-se outro divisor q (veja o exemplo anterior). Por exemplo, ao afirmar que 3 é um divisor de 12, pois $3 \times 4 = 12$, tem-se outro divisor, 4. Ou seja, os divisores vêm em pares. Veja outros exemplos:

Exemplo

O número 16 possui 5 divisores: 1 ($1 \times 16 = 16$), 2 ($2 \times 8 = 16$), 4 ($4 \times 4 = 16$), 8 ($8 \times 2 = 16$) e 16 ($16 \times 1 = 16$).

Dessa forma, pode-se limitar a encontrar cada elemento desses pares. Além disso, um dos valores de cada par deve ser limitado por \sqrt{n} . Por quê? Esse limite ajuda a reduzir a complexidade do algoritmo que encontra todos os divisores de um número em $O(\sqrt{n})$. A função abaixo retorna um `vector` com todos os divisores de n .

```

1  vector<long long> divisores(long long n) {
2      vector<long long> ans;
3      for(long long a = 1; a*a <= n; a++) { // ①
4          if(n % a == 0) {
5              long long b = n / a;
6              ans.push_back(a);
7              if(a != b) ans.push_back(b);
8          }
9      }

```

```

10     return ans; // ②
11 }
```

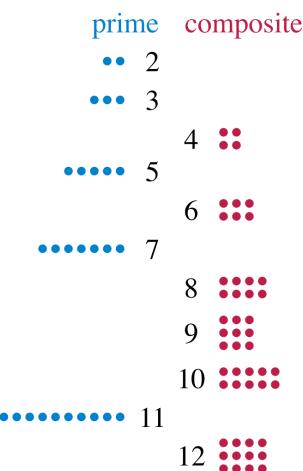
1. `x <= sqrt(n)` é o mesmo que `x*x <= n`.
2. Em alguns casos, é interessante ou necessário retornar os divisores ordenados

Complemente sua leitura e seu conhecimento:

- Number of divisors / sum of divisors
- Divisibility
- Counting Divisors of a Number in $O(n^{\frac{1}{3}})$
- How many divisors does a number have?

Números Primos

Um número inteiro $n > 1$ é chamado de **número primo** se e somente se possui dois divisores: 1 e n . Um número que não é primo é chamado de **número composto** (veja a figura abaixo). O primeiro e único número primo par é 2. Os próximos números primos são: 3, 5, 7, 11, 13, Como você deve imaginar, existe um número infinito de primos (Veja a prova [aqui](#)).



Números primos são os números naturais maiores que um que não são produtos de dois números naturais menores. (Fonte: [Wikipédia](#))

Número primo é um tópico importante da teoria dos números e a fonte de muitos problemas em programações competitivas. Por isso é de extrema importância conhecer e dominar alguns algoritmos que envolvam números primos.

Testes de Primalidade

Se um número n não é primo, então ele pode ser representado pelo produto de dois inteiros $a \times b$, onde $a \leq \sqrt{n}$ ou $b \leq \sqrt{n}$. Com isso, pode-se testar se um número é primo ou não e encontrar uma decomposição (fatoração) em fatores primos em $O(\sqrt{n})$. A função abaixo verifica se um dado número n é primo ou não.

```

1  bool ehPrimo(long long n) {
2      if (n < 2)
3          return false;
4      for (long long x = 2; x*x <= n; x++) { // ①
```

```

5         if (n % x == 0)
6             return false;
7     }
8     return true;
9 }
```

1. $x \leq \sqrt{n}$ é o mesmo que $x*x \leq n$.

Ou, alternativamente:

```

1 bool isPrimeFast(long long n) { // ①
2     if (n < 5 || n % 2 == 0 || n % 3 == 0)
3         return (n == 2 || n == 3);
4     long long maxP = sqrt(n) + 2;
5     for (long long p = 5; p < maxP; p += 6) {
6         if (p < n && n % p == 0) return false;
7         if (p+2 < n && n % (p+2) == 0) return false;
8     }
9     return true;
10 }
```

1. Fonte: [primes.cpp](#)

Complemente sua leitura e seu conhecimento:

- Primality tests

Decomposição em fatores primos

Todo número positivo n possui uma decomposição (fatoração) em fatores primos única: uma forma de decompor n em um produto de números primos, ou seja:

$$n = p_1^{a_1} \times p_2^{a_2} \times \cdots \times p_k^{a_k},$$

onde p_i são números primos distintos e a_i inteiros positivos.

A função abaixo retorna um `vector` com a decomposição em fatores primos de n .

```

1 vector<long long> factor(long long n) {
2     vector<long long> ans;
3     for (long long i = 2; i * i <= n; i++) {
4         while (n % i == 0) {
5             ans.push_back(i);
6             n /= i;
7         }
8     }
9     if (n > 1) ans.push_back(n);
10    return ans;
11 }
```

Note que cada fator primo aparece no vetor o número de vezes que ele divide n . Por exemplo, $24 = 2^3 \times 3$, então o resultado da função é $[2, 2, 2, 3]$.

Complemente sua leitura e seu conhecimento:

- Integer factorization
- Primalidade e fatoração

Crivo de Eratóstenes

O **Crivo de Eratóstenes** é um algoritmo para encontrar todos os números primos até um certo limite usando $O(n \log \log n)$ operações.

A ideia do algoritmo é a seguinte: inicialmente, escreve-se todos os números entre 2 e n . Então, marca-se todos os múltiplos de 2 (já que 2 é o menor número primo). Em seguida, pega-se o próximo valor que não foi marcado como composto, neste caso, é o 3. Isso significa que 3 é primo. Então, marca-se todos os múltiplos de 3 como compostos. O próximo número não marcado é o 5 (próximo número primo). Marca-se todos os múltiplos de 5. Este processo é repetido até n . A animação abaixo exemplifica a execução do algoritmo para $n = 120$.

	2	3	4	5	6	7	8	9	10	Prime numbers
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	

Crivo de Eratóstenes: passos do algoritmo para primos abaixo de 121. (Fonte: [Wikipédia](#))

O código abaixo exemplifica uma possível implementação do algoritmo. Esse algoritmo possui complexidade $O(n \log \log n)$ (veja a prova [aqui](#)).

```

1 vector<bool> crivo(long long n) {
2     vector<bool> primo(n+1, true); // (1)
3     primo[0] = primo[1] = false;
4     for (long long i = 2; i <= n; i++) {
5         if (primo[i] && i * i <= n) {
6             for (long long j = i * i; j <= n; j += i) // (2)
7                 primo[j] = false;
8         }
9     }
10    return primo;
11 }
```

1. Cria um array (`vector`) booleano de tamanho $n + 1$, onde todas as posições são inicializadas com `true`.

2. Iteramos por todos os números divisíveis pelo primo `i`

Complemente sua leitura e seu conhecimento:

- [Sieve of Eratosthenes](#)
- [Linear Sieve](#)
- [Math note - linear sieve](#)

Primo de Mersenne

Número de Mersenne é todo número natural da forma $M_n = 2^n - 1$, onde n é um número natural. Os primeiros números Mersenne são: 0, 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, ... Um subconjunto particularmente interessante é o constituído pelos números de Mersenne que são também primos: os **primos de Mersenne**. Note que nem todo número de Mersenne é primo, assim como nem todo número primo é de Mersenne.

Os primeiros primos de Mersenne são:

$M_2 = 3, M_3 = 7, M_5 = 31, M_7 = 127, M_{13} = 8191, M_{17} = 131071, M_{19} = 524287, \dots$

Um resultado elementar sobre os números de Mersenne afirma que se $2^n - 1$ é um número primo, então n também é um número primo.

Algoritmo de Euclides (MDC/MMC)

O **máximo divisor comum** (GCD, do inglês *greatest common divisor*) dos números a e b , $\gcd(a, b)$, é o maior número que divide a e b , e o **mínimo múltiplo comum** (LCM, do inglês *least common multiple*) de a e b , $\text{lcm}(a, b)$, é o menor número que é divisível por a e b .

O algoritmo de Euclides fornece uma maneira eficiente de encontrar o máximo divisor comum de dois números. O algoritmo é baseado na seguinte definição:

$$\gcd(a, b) = \begin{cases} a, & \text{se } b = 0 \\ \gcd(b, a \bmod b), & \text{caso contrário.} \end{cases}$$

Usando essa definição, o algoritmo é facilmente implementado usando recursão:

```

1 long long gcd(long long a, long long b) {
2     if (b == 0)
3         return a;
4     else
5         return gcd(b, a % b);
6 }
```

Dica

Você pode usar a função integrada `_gcd(a, b)` do C++.

Pode-se mostrar que o algoritmo de Euclides possui complexidade $O(\log n)$, onde $n = \min(a, b)$.

O **mínimo múltiplo comum** (LCM) pode ser calculado da seguinte forma:

$$\text{lcm}(a, b) = \frac{a \times b}{\gcd(a, b)}$$

Para calcular o GCD ou LCM de mais de dois valores, pode-se calcular o valor de dois em dois (em qualquer ordem). Por exemplo:

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

Função totiente de Euler

A função totiente de Euler, também conhecida como função $\phi(n)$, conta o número de inteiros entre 1 e n , no qual são **coprimos** de n . Dois números são coprimos se o **máximo divisor comum** entre eles for 1 (1 é considerado ser coprimo para qualquer número). Por exemplo, $\phi(12) = 4$, pois 1, 5, 7 e 11 são coprimos de 12.

Abaixo estão valores de $\phi(n)$ para os primeiros números inteiros positivos:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$\phi(n)$	1	1	2	2	4	2	6	4	6	4	10	4	12	6	8	8	16	6	18	8	12

A função abaixo calcula o valor de $\phi(n)$ em $O(\sqrt{n})$.

```

1 long long phi(long long n) {
2     long long result = n;
3     for (long long i = 2; i * i <= n; i++) {
4         if (n % i == 0) {
5             while (n % i == 0)
6                 n /= i;
7             result -= result / i;
8         }
9     }
10    if (n > 1)
11        result -= result / n;
12    return result;
13 }
```

Complemente sua leitura e seu conhecimento:

- [Euclidean algorithm for computing the greatest common divisor](#)
- [Euler's totient function](#)

Aritmética Modular

Na **aritmética modular**, o conjunto de números é limitado de forma que apenas os números $0, 1, 2, \dots, m - 1$ são usados, onde m é uma constante. Cada número x é representado pelo número $x \bmod m$: o resto da divisão de x por m . Por exemplo, se $m = 23$, em vez $x = 247$, considera-se $x \bmod 23 = 17$. Normalmente, m será um primo grande, dado no problema, normalmente, $10^9 + 7$. A aritmética modular é usada para evitar [integer overflow](#).

As seguintes propriedades valem no cálculo do módulo:

$$(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m$$

$$(a - b) \bmod m = (a \bmod m - b \bmod m) \bmod m$$

$$(a \cdot b) \pmod{m} = ((a \bmod m) \cdot (b \bmod m)) \bmod m$$

$$a^b \bmod m = (a \bmod m)^b \bmod m$$

O que significa que se a resposta está sendo computada por meio de adições, subtrações e multiplicações, e no final você precisa tirar o módulo dela, você pode tirar o módulo em todas as operações intermediárias que isso

não afetará a resposta.

Exponenciação Binária

A **exponenciação binária** é um truque que permite calcular x^n usando apenas multiplicações $O(\log n)$ (em vez das multiplicações $O(n)$ exigidas pela abordagem ingênua).

Sabe-se que $x^a \cdot x^b = x^{a+b}$. Em particular, $x^{2b} = x^b \cdot x^b$. Logo, se o expoente n de x^n for par, pode-se dizer que $x^n = x^{\frac{n}{2}} \cdot x^{\frac{n}{2}}$. No entanto, se n for ímpar, tem-se algo similar: $x^n = x^{\frac{n-1}{2}} \cdot x^{\frac{n-1}{2}} \cdot x$. Dessa forma, pode-se montar a seguinte recorrência:

$$x^n = \begin{cases} 1 & \text{se } n = 0 \\ \left(x^{\frac{n}{2}}\right)^2 & \text{se } n \text{ par} \\ \left(x^{\frac{n-1}{2}}\right)^2 \cdot x & \text{se } n \text{ ímpar} \end{cases}$$

Note que se n for par, o valor $x^{n/2}$ deve ser calculado apenas uma vez. Isso garante que a complexidade do algoritmo seja $O(\log n)$. A função abaixo calcula o valor de x^n :

```

1 long long binpow(long long x, long long n) {
2     if (n == 0)
3         return 1;
4     long long res = binpow(x, n / 2);
5     if (n % 2)
6         return res * res * x;
7     else
8         return res * res;
9 }
```

Alternativamente, sem usar recursão e usando [manipulação de bits](#):

```

1 long long binpow(long long x, long long n) {
2     long long res = 1;
3     while (n > 0) {
4         if (n & 1)
5             res = res * x;
6         x = x * x;
7         n >>= 1;
8     }
9     return res;
10 }
```

Em alguns casos, é necessário calcular o valor de $x^n \bmod m$. Sabendo que $(a \cdot b) \pmod m = ((a \pmod m) \cdot (b \pmod m)) \pmod m$, pode-se usar diretamente código anterior e apenas substituir cada multiplicação por uma multiplicação modular:

```

1 long long binpow(long long x, long long n, long long m = 1) { // (1)
2     x %= m;
3     long long res = 1;
4     while (n > 0) {
5         if (n & 1)
6             res = res * x % m;
7         x = x * x % m;
8         n >>= 1;
9     }
10    return res;
11 }
```

1. Deixando o valor padrão de m como 1, pode-se usar essa função sem passar o valor de m como parâmetro.

 Dica

Se m é um número primo, pode-se acelerar um pouco este algoritmo calculando $x^{n \bmod (m-1)}$ em vez de x^n .

Complemente sua leitura e seu conhecimento:

- [Binary Exponentiation](#)

1. O texto dessa página são traduções e adaptações encontrados aqui: [1](#), [2](#), [3](#), [4](#) e [5](#) ←

Ordenação¹

A maioria das linguagens de programação modernas implementa uma função de ordenação eficiente. Em C++, tem-se a função `sort` da biblioteca `<algorithm>`. Veja alguns exemplos de uso desta função:

```

1 int n = 7;
2 int a[] = {4, 2, 5, 3, 5, 8, 3};
3 sort(a, a+n); // ①
4
5 vector<int> v = {4, 2, 5, 3, 5, 8, 3};
6 sort(v.begin(), v.end()); // ②
7
8 vector<pair<int, string>> alunos = {{789, "Paulo"}, {456, "Ana"}, {123,
9 "Paulo"}};
10 sort(alunos.begin(), alunos.end()); // ③
11 for (auto a: alunos)
    cout << a.first << " " << a.second << endl; // ④

```

1. [2, 3, 3, 4, 5, 5, 8]
2. [2, 3, 3, 4, 5, 5, 8]
3. Por padrão, um `pair` sempre é ordenado pelo campo `first`.
4. {123, Paulo} {456, Ana} {789, Paulo}

A ordenação padrão é a não-decrescente, mas pode-se obter a ordem inversa da seguinte forma:

```

1 vector<pair<int, string>> alunos = {{789, "Paulo"}, {456, "Ana"}, {123,
2 "Paulo"}};
3 sort(alunos.rbegin(), alunos.rend());
4 for (auto a: alunos) // ①
    cout << a.first << " " << a.second << endl;

```

1. {789, Paulo} {456, Ana} {123, Paulo}

Para ordenarmos um `struct` ou `class` ou outra coleção de objetos (por exemplo, um `pair` pelo campo `second`), tem-se duas alternativas: (i) definir uma função de comparação ou (ii) fazer a sobrecarga do operador `<`.

Agora suponha que seja necessário ordenar objetos do tipo `Pessoa`:

```

1 class Pessoa {
2 public:
3     string nome, sobrenome;

```

```

4     int idade;
5     Pessoa() { // ①
6         this->nome = "";
7         this->sobrenome = "";
8         this->idade = 0;
9     }
10    Pessoa(string _nome, string _sobrenome, int _idade) {
11        this->nome = _nome;
12        this->sobrenome = _sobrenome;
13        this->idade = _idade;
14    }
15    void imprime() {
16        cout << "(" << nome << ", " << sobrenome << ", " << idade <<
17        "\n";
18    }
19

```

1. Construtor padrão usado quando não passamos nenhum argumento para a classe.

Sempre faça o construtor padrão. Caso contrário, não será possível fazer, por exemplo:

```
Pessoa p; //Chama o construtor padrão.
```

Para ordenarmos um `struct` ou `class` ou outra coleção de objetos (por exemplo, um `pair` pelo campo `second`), tem-se duas alternativas: (i) definir uma função de comparação ou (ii) fazer a sobrecarga do operador `<`.

Usando uma função de comparação

A primeira alternativa é definir uma função que compara dois objetos e retorna `true` caso o primeiro seja considerado menor que o segundo e `false`, caso contrário. Por exemplo,

```

1  bool compara(const Pessoa &p1, const Pessoa &p2) { // (1)
2      if (p1.nome < p2.nome)
3          return true;
4      else if (p1.nome == p2.nome) {
5          if (p1.sobrenome < p2.sobrenome)
6              return true;
7          else if (p1.sobrenome == p2.sobrenome)
8              return p1.idade > p2.idade;
9          else
10             return false;
11     }
12     else
13         return false;
14 }

```

1. O `const` diz ao compilador que os objetos passados para a função não serão alterados internamente. Já o `&` representa **passagem por referência**, uma alternativa a **passagem por ponteiro**. Dessa forma, não passamos uma cópia dos objetos e sim referências (endereços) dos mesmos, o que torna o código mais eficiente. Por quê?

Com a função de comparação definida, basta passá-la como argumento na função `sort`:

```

1 int main() {
2     Pessoa p1("Paulo", "Roberto", 35);
3     Pessoa p2("Paulo", "Alberto", 30);
4     Pessoa p3("Paulo", "Roberto", 40);
5
6     vector<Pessoa> lista;
7     lista.push_back(p1); // (1)
8     lista.push_back(p2);
9     lista.push_back(p3);
10    for (auto p: lista)
11        p.imprime();
12    cout << "\n";
13    sort(lista.begin(), lista.end(), compara); // (2)
14    for (auto p : lista)
15        p.imprime();
16
17    return 0;
18 }
```

1. Inclui o elemento no fim do `vector`. Leia mais sobre `vector` [aqui](#).

2. Usa a função `compara` para fazer a ordenação.

Também é possível usar uma [função lambda](#) como função de comparação. Por exemplo, para ordenar um `pair` pelo campo `second` (ou usando-o como critério de desempate), pode-se fazer:

```

1 bool compara(const pair<int, int> &p1, const pair<int, int> &p2) {
2     if (p1.first < p2.first)
3         return true;
4     else if (p1.first == p2.first)
5         return p1.second < p2.second;
6     else
7         return false;
8 }
9
10 int main() {
11     vector<pair<int, int>> pontos = {{0, 2}, {0, 1}, {0, 0}, {0, -5}, {1,
12 2}, {-1, 10}};
13
14     sort(pontos.begin(), pontos.end(), compara); // (1)
15     for (auto a: pontos) // (2)
16         cout << "(" << a.first << ", " << a.second << ") ";
17
18     sort(pontos.begin(), pontos.end(), [] (const pair<int, int> &p1, const
19 pair<int, int> &p2) { // (3)
20         if (p1.first < p2.first)
21             return true;
22         else if (p1.first == p2.first)
23             return p1.second < p2.second;
24         else
25             return false;
26     });
27     for (auto a: pontos)
```

```

28         cout << "(" << a.first << ", " << a.second << ") ";
29
30     return 0;
31 }
```

1. Usando a função `compara`

2. (-1, 10) (0, -5) (0, 0) (0, 1) (0, 2) (1, 2)

3. Usando função lambda

Fazendo a sobrecarga do operador <

Ao invés de ser definidas funções de comparações, pode-se fazer a sobrecarga do operador `<` (`operator<`). Isso é comum ao se usar `class` ou `struct`. Veja como ficaria ao ser considerado a classe `Pessoa`:

```

1 class Pessoa {
2 public:
3     string nome, sobrenome;
4     int idade;
5     Pessoa(){
6         this->nome = "";
7         this->sobrenome = "";
8         this->idade = 0;
9     }
10    Pessoa(string _nome, string _sobrenome, int _idade){
11        this->nome = _nome;
12        this->sobrenome = _sobrenome;
13        this->idade = _idade;
14    }
15    bool operator<(const Pessoa &p){ // (1)
16        if (nome < p.nome)
17            return true;
18        else if (nome == p.nome) {
19            if (sobrenome < p.sobrenome)
20                return true;
21            else if (sobrenome == p.sobrenome)
22                return idade > p.idade;
23            else
24                return false;
25        }
26        else
27            return false;
28    }
29    void imprime() {
30        cout << "(" << nome << ", " << sobrenome << ", " << idade <<
31        ")" \n";
32    }
33 }
34 };
35
36 int main() {
37     Pessoa p1("Paulo", "Roberto", 35);
```

```
38     Pessoa p2("Paulo", "Alberto", 30);
39     Pessoa p3("Paulo", "Roberto", 40);
40
41     vector<Pessoa> lista;
42     lista.push_back(p1);
43     lista.push_back(p2);
44     lista.push_back(p3);
45     for (auto p: lista)
46         p.imprime();
47     cout << "\n";
48     sort(lista.begin(), lista.end()); // (2)
49     for (auto p : lista)
50         p.imprime();
51
52     return 0;
}
```

1. Sobrecarga/definição do operador `<` para a classe `Pessoa`. Dessa forma, é possível fazer a comparação `p1 < p2`, considerando que `p1` e `p2` são objetos do tipo `Pessoa`.
2. Usa o operador `<` para fazer a ordenação.

Material complementar

- [Competitive Programmer's Handbook](#)
- [Sorting \(CS Academy\)](#)
- [std::sort](#)
- [std::partial_sort](#)
- [std::stable_sort](#)
- [std::nth_element](#)

1. O texto dessa página são traduções e adaptações encontrados aqui: [1](#), [2](#) e [3](#) ↪

Busca¹

Busca sequencial

A forma mais intuitiva para procurar um elemento em um *array* é usar um *loop* que percorre todos os elementos do array e parando assim que o elemento buscado é encontrado. No pior caso, é necessário percorrer todos os elementos, logo a complexidade será $O(n)$. O código abaixo verifica se `x` está no *array*:

```

1  for (int i = 0; i < n; i++) {
2      if (array[i] == x) {
3          // x encontrado na posição i
4      }
5  }
```

Em C++, pode-se usar a função `search`.

Busca binária

Se os elementos do *array* estiverem ordenados, pode-se usar uma estratégia diferente e mais eficiente para realizar a busca: verifique se o elemento do meio do *array* é o elemento buscado, se for, a busca termina. Caso não seja, verique se o elemento do meio é menor que elemento buscado, se for, repita o processo considerando apenas a segunda metade do *array*. Senão, considere a primeira metade do *array*. Assim, a cada passo da busca, o tamanho do *array* é reduzido a metade, logo, a complexidade do algoritmo é $O(\log n)$.

```

1  int buscaBinaria(vector<int> v, int x) {
2      int ini = 0, fim = v.size() - 1;
3      while (ini <= fim) {
4          int meio = ini + (fim - ini) / 2; // ①
5          if (v[meio] == x) return meio;
6          else if (v[meio] < x) ini = meio + 1;
7          else fim = meio - 1;
8      }
9      return -1; // ②
10 }
```

1. Evite usar `meio = (ini + fim) / 2;`, já que `ini + fim` pode gerar *integer overflow*.
2. `x` não está no vetor `v`.

Em C++, pode-se usar a função `std::binary_search`. As funções abaixo também são úteis e baseadas na busca binária:

- `std::lower_bound`: retorna um ponteiro para o primeiro elemento do *array* cujo valor é pelo menos `x`;
- `std::upper_bound`: retorna um ponteiro para o primeiro elemento do *array* cujo valor é maior que `x`;

As funções assumem que o *array* está ordenado. Se o valor procurado não for encontrado, é retornado um ponteiro para o elemento após o último elemento do *array*. Por exemplo, o código a seguir verifica se o *array* contém um elemento com valor `x`:

```

1 vector<int> v = {1, 2, 3, 5, 8, 10, 20};
2 int x = 10;
3 auto k = lower_bound(v.begin(), v.end(), x);
4 if (k != v.end() && *k == x) {
5     // x encontrado no índice k
6     cout << "Valor encontrado na posição: " << k - v.begin() << endl; //
7 }
```

1. Pode-se usar a função `std::distance` ao invés de `k - v.begin()`, ou seja,
`distance(v.begin(), k)`.

Complemente sua leitura e seu conhecimento:

- [Binary Search tutorial \(C++ and Python\)](#)
- [Busca Binária](#)
- [Binary Search](#)
- [Binary Search \(ITMO Academy\)](#) 🎯
- [Binary Search \(CS Academy\)](#)

Busca binária em funções monotônicas²

Considere uma função booleana $f(x)$ e se deseja encontrar o valor máximo (ou mínimo) de x tal que $f(x)$ seja `true`. Da mesma forma que a busca binária só funciona se o *array* estiver ordenado, só é possível aplicar a busca binária em uma função **monótona**, ou seja, é sempre não-decrescente ou sempre não-crescente.

Seja `check(x)` uma função que verifica uma propriedade de `x`. Se para todo `x`, `check(x) = true` implica `check(x+1) = true`, ou para todo `x`, `check(x) = false` implica `check(x+1) = false`, então a função `check` é monótona.

Suponha a função `check` abaixo que verifica se um elemento é maior ou igual a `x`. Se `x = 11` e o vetor `v = [1, 2, 3, 5, 8, 11, 12, 14, 16]`, então teremos o seguinte vetor de saída ao aplicarmos `check` em `v`: `[0, 0, 0, 0, 0, 1, 1, 1, 1]`.

```

1  bool check(int val) {
2      return val >= x;
3 }
```

Dessa forma, a função `check` para essa situação é monótona e isso é relevante porque se um valor do vetor satisfizer a condição, todos os valores a direita também vão satisfazê-la, e de forma análoga, todos os valores a esquerda de um índice que não satisfaz a condição, também não vão satisfazer, e é isso que nos permite aplicar busca binária. Além disso, a função `check` só se torna monótona nesse exemplo quando o vetor está ordenado, por isso a busca binária só é feita em vetores ordenados.

Como encontrar o menor valor que torna `check` verdadeiro? R. inicia-se o processo "chutando" um intervalo onde a resposta com certeza estará. Para cada intervalo, checa-se o meio e, dependendo da resposta, descarta-se os elementos a direita ou a esquerda, mas sempre divide-se o tamanho do intervalo por 2, até que o intervalo tenha tamanho 1. Veja uma solução:

```

1  int l = a;// sei que a resposta não é menos que a
2  int r = b;// sei que a resposta não é mais que b
3
4  while(r > l+1){// repita enquanto o intervalo tiver tamanho > 2
5      int mid = l + (r - l)/2;
6      if(check(mid)){ // mid é válido
7          r = mid; // como queremos minimizar a resposta, e mid é uma
8          resposta válida
9              //descartamos tudo a direita de mid (mas não mid)
10     }
11     else{
12         l = mid + 1; // Se mid não é válido, descartamos ele e tudo
13         abaixo.
14     }
15 }
16 // Ao final desse laço, a resposta pode estar em l ou r.
17 // Queremos minimizar a resposta, então se l for válido,
18 // ficaremos com l, e caso contrário, com r
int ans = r;
if(check(l)) ans = l;
```

Exemplo: Encontrar o maior valor de $x \in [0, 10]$ tal que $x^2 \leq 30$.

```

1  bool check(int val) {
2      return val*val <= 30;
3 }
4  int lastTrue(int ini, int fim) {
5      ini--; // Se nenhum valor no intervalo for true, retorna ini - 1
6      while (ini < fim) {
7          int m = ini + (fim - ini) / 2;
8          if (check(m)) ini = m; // (1)
9          else fim = m - 1; // (2)
10     }
```

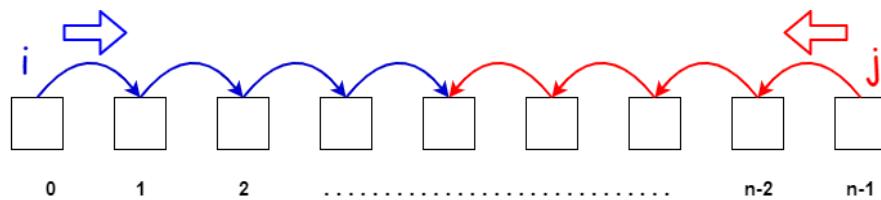
```

11     return ini;
12 }
```

1. Se `check(m)` é `true`, então todos os números menores que `m` também serão `true`.
2. Se `check(m)` é `false`, então todos os números maiores que `m` também serão `false`.

Two-Pointers

Na técnica chamada **Two-Pointers** dois "apontadores" caminham pelo vetor. Normalmente, esses apontadores são "colocados" nas extremidades opostas do vetor e caminham um em direção ao outro, como mostra a figura abaixo.



Fonte: [AfterAcademy](#)

Você consegue pensar em como usar esse técnica para resolver o problema de inverter os elementos de um vetor sem usar um vetor auxiliar? A ideia é simples: coloque cada apontador (digamos `i` e `j`) em uma extremidade do vetor, ou seja, `i = 0` e `j = n - 1`, e, a cada iteração, troque os elementos que estão nas posições `i` e `j` (ou seja, `v[i] <-> v[j]`). Após a troca, incremente o apontador `i` e decremente o apontador `j`. Repita esse processo enquanto `i < j`. A figura abaixo ilustra parte desse processo.



Fonte: [AfterAcademy](#)

O código abaixo ilustra essa estratégia (note a simplicidade):

```
1 void inverte(vector<int> &v) { // (1)
2     int i = 0;
3     int j = v.size() - 1;
4     while ( i < j ) {
5         swap(v[i], v[j]); // (2)
6         i++;
7         j--;
8     }
9 }
```

1. A `std::reverse` também pode ser usada com o mesmo objetivo. O intuito é mostrar com a técnica Two-Pointers funciona.
2. `std::swap`

Complemente sua leitura e seu conhecimento:

- [Two Pointers Method \(ITMO Academy\)](#) 🎨
- [Two Pointers Technique](#)
- [What is the Two pointer technique?](#)

-
1. O texto dessa página são traduções e adaptações encontrados aqui: [1](#), [2](#) e [3](#) ←
 2. Conteúdo extraído de [Busca Binária](#) ←

Estrutura de Dados e STL¹

Estrutura de dados (ED) é a forma como os dados são armazenados na memória do computador com o objetivo de tornar o processamento mais fácil e eficiente. Cada uma possui suas próprias vantagens e desvantagens, por isso, é crucial conhecer diferentes estruturas de dados (básicas e avançadas) para conseguir definir qual as mais apropriada para um determinado problema. Em programação competitiva, com o objetivo de economizar uma grande quantidade de tempo ao implementar um algoritmo, também é muito importante saber quais EDs estão disponíveis na biblioteca padrão, como usá-las e qual a complexidade de cada operação da ED.

A seguir são apresentados as estruturas de dados presentes na biblioteca padrão do C++ comumente usadas em competições.

Leitura recomendada:

- [C++ Standard Library Containers](#)
- [More Operations on Sorted Sets](#)
- [C++ Sets with Custom Comparators](#)
- [C++ Containers library](#)

vector

Um `vector` é um array dinâmico que permite adicionar e remover elementos de forma eficiente no final da estrutura. Por exemplo, o código a seguir cria um vetor vazio e adiciona três elementos a ele:

```

1 vector<int> v;
2 v.push_back(3); // [3]
3 v.push_back(2); // [3,2]
4 v.push_back(5); // [3,2,5]
```

Observe que os elementos são inseridos no fim. Então, os elementos podem ser acessados como em um array comum:

```

1 cout << v[0] << "\n"; // 3
2 cout << v[1] << "\n"; // 2
3 cout << v[2] << "\n"; // 5
```

Outra maneira de criar um vetor é fornecer uma lista de seus elementos:

```
1 vector<int> v = {2, 4, 2, 5, 1};
```

Também pode-se fornecer o número de elementos e seus valores iniciais:

```
1 vector<int> a(8); // tamanho 8, valor inicial: 0
2 vector<int> b(8, 2); // tamanho 8, valor inicial: 2
```

A função `size()` retorna o número de elementos no `vector`. Por exemplo:

```
1 for (int i = 0; i < v.size(); i++) {
2     cout << v[i] << "\n";
3 }
```

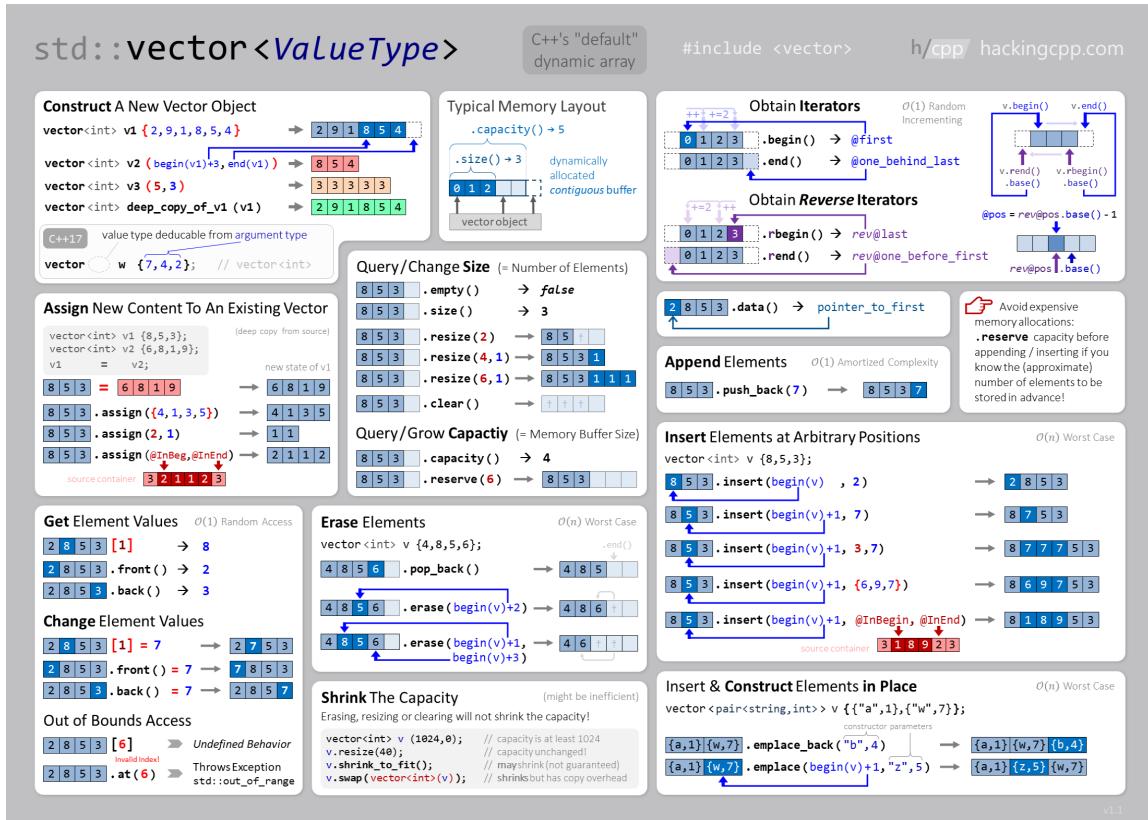
Uma alternativa mais simples é a seguinte:

```
1 for (auto x : v) {
2     cout << x << "\n";
3 }
```

A função `back` retorna o último elemento de um vetor e a função `pop_back` remove o último elemento:

```
1 vector<int> v = {2, 4, 2, 5, 1};
2 cout << v.back() << "\n"; // 1
3 v.pop_back();
4 cout << v.back() << "\n"; // 5
```

A figura abaixo mostra mais operações e uso do `vector`.

Fonte: [Hacking C++](#)

Complemente sua leitura e seu conhecimento:

- `std::vector`
- `std::vector`

deque

Um `deque` é um array dinâmico que pode ser manipulado eficientemente em ambas as extremidades da estrutura. Como um `vector`, um `deque` fornece as funções `push_back` e `pop_back`, mas também fornece as funções `push_front` e `pop_front` que não estão disponíveis em um `vector`. veja uma exemplo:

As principais funções da `deque` são:

- `push_front(x)` : adiciona o elemento `x` no início da estrutura;
- `push_back(x)` : adiciona o elemento `x` no fim da estrutura;
- `pop_front()` : remove o primeiro elemento da estrutura;
- `pop_back()` : remove o último elemento da estrutura;
- `front()` : retorna o primeiro elemento da estrutura;

- `back()` : retorna o último elemento da estrutura;
- `size()` : retorna o número de elementos da estrutura.

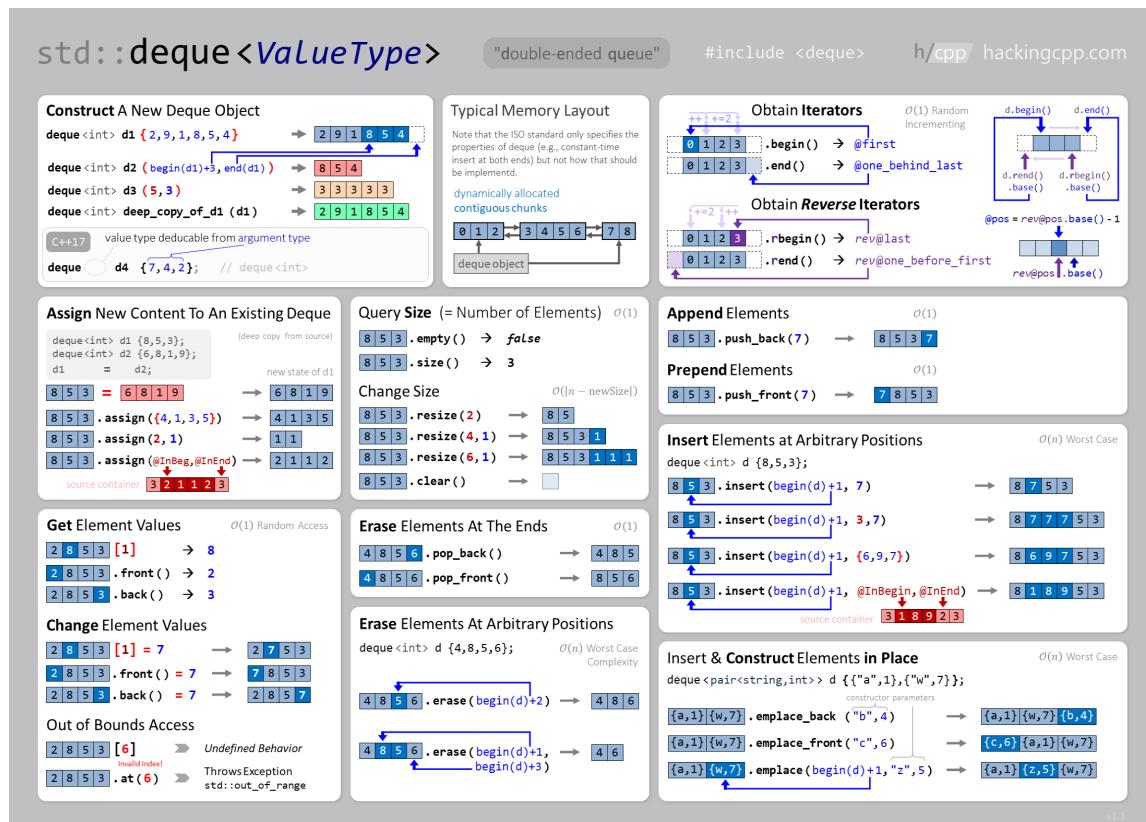
O código a seguir ilustra a utilização da estrutura:

```

1  deque<int> d;
2  d.push_back(5); // [5]
3  d.push_back(2); // [5,2]
4  d.push_front(3); // [3,5,2]
5  d.pop_back(); // [3,5]
6  d.pop_front(); // [5]

```

As operações de um `deque` funcionam em tempo médio $O(1)$. A figura abaixo mostra mais operações e uso do `deque`.



Fonte: [Hacking C++](#)

Complemente sua leitura e seu conhecimento:

- `std::deque`
- `std::deque`

queue

A estrutura da `queue` (fila) corresponde a uma fila simples da vida real e segue a regra *First In First Out* (FIFO). Suas principais operações são: inserir um elemento no fim da fila, acessar e remover o primeiro elemento da fila. Essas operações possuem complexidade em $O(1)$.

As principais funções da `queue` são:

- `push(x)` : adiciona o elemento `x` no fim da fila;
- `pop()` : remove o primeiro elemento da fila;
- `front()` : retorna o primeiro elemento da fila;
- `size()` : retorna o número de elementos da fila.

O código a seguir ilustra a utilização da estrutura:

```

1 queue<int> q;
2 q.push(2);           // [2]
3 q.push(5);           // [2, 5]
4 cout << q.size() << "\n"; // 2
5 cout << q.front() << "\n"; // 2
6 q.pop();             // [5]
7 cout << q.back() << "\n"; // 5

```

Complemente sua leitura e seu conhecimento:

- [std::queue](#)
- [Queue Data Structure \(GeeksforGeeks\)](#)

stack

Uma `stack` (pilha) é uma estrutura muito semelhante a uma fila, mas que segue a regra *Last In First Out* (LIFO). Ou seja, ao inserir um elemento na pilha, ele é adicionado no topo e esse é o elemento que se tem acesso. Suas principais operações são: inserir um elemento no topo da pilha, acessar e remover o elemento do topo da pilha. Essas operações possuem complexidade em $O(1)$.

As principais funções da `stack` são:

- `push(x)` : adiciona o elemento `x` no topo da pilha;
- `pop()` : remove o elemento do topo da pilha;
- `top()` : retorna o elemento do topo da pilha;
- `size()` : retorna o número de elementos da pilha.

Veja um exemplo de utilização da estrutura:

```

1 stack<int> s;
2 s.push(2);           // [2]

```

```

3 s.push(5);           // [2, 5]
4 cout << s.size() << "\n"; // 2
5 cout << s.top() << "\n"; // 5
6 s.pop();             // [2]
7 cout << s.top() << "\n"; // 2

```

Complemente sua leitura e seu conhecimento:

- [std::stack](#)
- [Stack Data Structure \(GeeksforGeeks\)](#)
- [Stacks](#)

priority_queue

Uma fila de prioridade (`priority_queue`) é uma estrutura semelhante a uma fila ou pilha, mas ao invés de inserções e remoções acontecerem em uma das extremidades da estrutura, o **maior** (ou **menor**) elemento é sempre retornado durante o acesso/remoção. Inserções e remoções possuem complexidade $O(\log n)$ e o acesso ao elemento de maior prioridade é $O(1)$. Uma fila de prioridade geralmente é implementada usando uma estrutura chamada [heap](#) que é muito mais simples do que uma árvore binária balanceada usado em um [set](#).

As principais funções da `priority_queue` são:

- `push(x)` : adiciona o elemento `x` na fila de prioridade;
- `pop()` : remove o elemento de maior prioridade;
- `top()` : retorna o elemento de maior prioridade;
- `size()` : retorna o número de elementos da fila de prioridade.

O código a seguir ilustra a utilização da estrutura:

```

1 priority_queue<int> q;
2 q.push(3);
3 q.push(5);
4 q.push(7);
5 q.push(2);
6 cout << q.top() << "\n"; // 7
7 q.pop();
8 cout << q.top() << "\n"; // 5
9 q.pop();
10 q.push(6);
11 cout << q.top() << "\n"; // 6
12 q.pop();

```

Se for necessário criar uma fila de prioridade que suporte encontrar e remover o menor elemento, pode-se fazer da seguinte forma:

```

1  bool cmp(const int& a, const int& b) {
2      return a > b;
3  }
4  priority_queue<int, vector<int>, cmp> pq; // ①

```

1. Note o uso da função de comparação `cmp`. Essa função deve receber dois argumentos do tipo armazenado na fila de prioridade e retorna `true` se o primeiro for considerado menor que o segundo.

Para o exemplo anterior, outra alternativa é usar a função `greater`:

```
1 priority_queue<int, vector<int>, greater<int>> pq;
```

Complemente sua leitura e seu conhecimento:

- [Priority Queue in C++ Standard Template Library \(STL\)](#)
- [std::priority_queue](#)

set e multiset

Semelhante a um conjunto matemático, um `set` é uma coleção de **elementos únicos**, ou seja, todos os seus elementos são distintos. A estrutura é baseada em uma árvore binária balanceada (*red-black tree*) e acessar seus elementos é $O(\log n)$. Consequentemente, não é possível acessar os elementos do `set` usando o operador `[]`, como acontece em um `vector`. Além disso, os elementos são mantidos **ordenados**.

As principais funções do `set` (e `multiset`) são:

- `insert(x)`: adiciona o elemento `x` no conjunto. Se ele já estiver no conjunto, nada é feito;
- `erase(x)`: remove o elemento `x`;
- `count(x)`: retorna o número de elemento cuja chave seja `x`;
- `find(x)`: retorna um iterador para o elemento com chave `x`;
- `size()`: retorna o número de elementos do conjunto.

O código a seguir ilustra a utilização da estrutura `set`:

```

1 set<int> s;
2 s.insert(3);
3 s.insert(2);
4 s.insert(5);
5 cout << s.count(3) << "\n"; // 1
6 cout << s.count(4) << "\n"; // 0
7 s.erase(3);
8 s.insert(4);
9 cout << s.count(3) << "\n"; // 0

```

```

10 cout << s.count(4) << "\n"; // 1
11 if (s.find(7) == s.end()){
12     cout << "7 não está no set" << "\n";
13 }
14 auto menor = s.begin();
15 auto maior = s.end(); maior--; // ①
16 cout << "O menor elemento do conjunto é: " << *menor << "\n";
17 cout << "O maior elemento do conjunto é: " << *maior << "\n";
18
19 set<int> c1 = {2, 5, 6, 8};
20 cout << c1.size() << "\n"; // 4
21 for (auto x: c1)
22     cout << x << "\n";
23
24 set<int> c2;
25 c2.insert(5);
26 c2.insert(5);
27 c2.insert(5);
28 cout << c2.count(5) << "\n"; // 1 ②

```

1. Como `end()` aponta para um elemento após o último elemento, deve-se diminuir o iterador em uma unidade.
2. Lembre-se que em um `set` todos seus elementos são distintos. Assim, a função `count` sempre retornar ou 0 (elemento não está no `set`) ou 1 (o elemento está no `set`).

A estrutura `set` também fornece as funções `lower_bound(x)` e `upper_bound(x)` que retornam um iterador para o menor elemento em um `set` cujo valor é pelo menos ou maior que `x`, respectivamente. Em ambas as funções, se o elemento solicitado não existir, o valor de retorno é `end()`.

Um `multiset` é um conjunto que pode conter várias cópias do mesmo elemento. Por exemplo, o código abaixo adiciona três cópias do valor 5 ao `multiset`:

```

1 multiset<int> s;
2 s.insert(5);
3 s.insert(5);
4 s.insert(5);
5 cout << s.count(5) << "\n"; // 3

```

A função `erase` remove todas as cópias de um valor do `multiset`:

```

1 s.erase(5);
2 cout << s.count(5) << "\n"; // 0

```

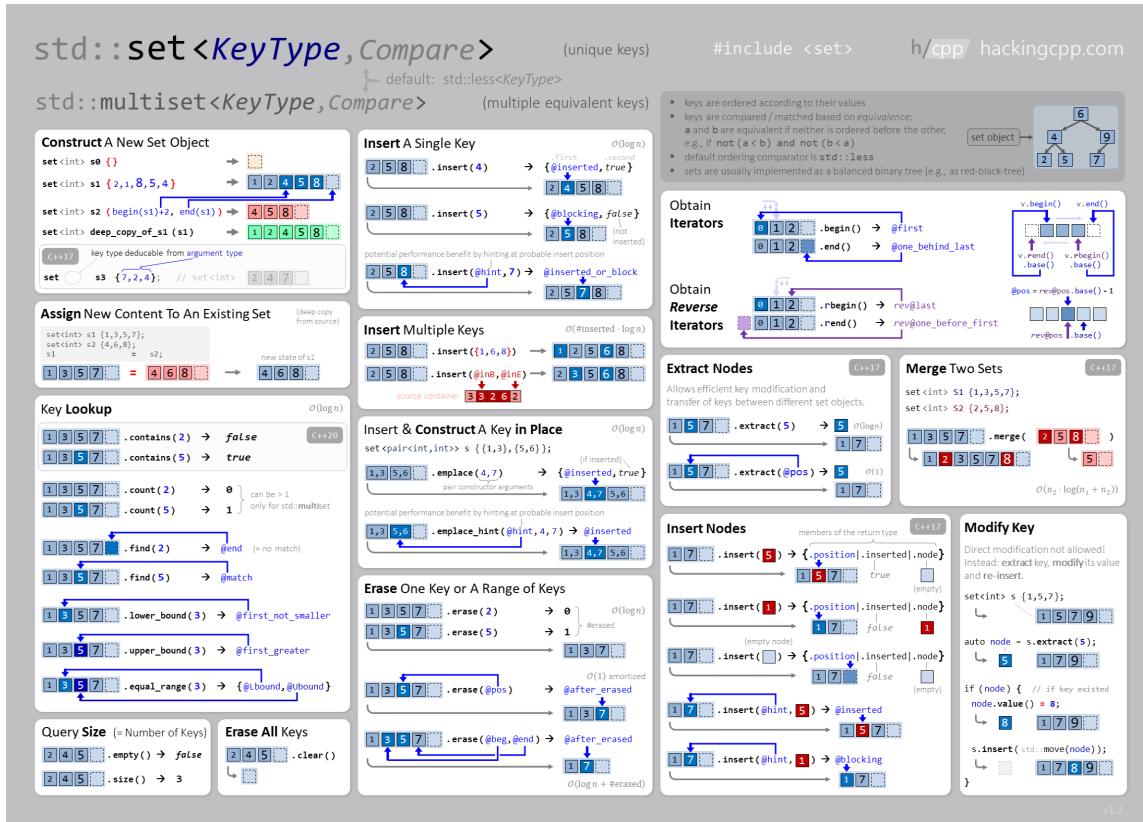
Caso seja necessário remover apenas uma cópia, pode-se fazer da seguinte forma:

```

1 s.erase(s.find(5));
2 cout << s.count(5) << "\n"; // 2

```

A figura abaixo mostra mais operações e uso do `set` e `multiset`.

Fonte: [Hacking C++](#)

Complemente sua leitura e seu conhecimento:

- Standard Associative Containers
- Set in C++ Standard Template Library (STL)
- Multiset in C++ Standard Template Library (STL)
- `std::set`
- `std::multiset`

map e multimap

Um `map` é um conjunto que consiste em pares de valores-chave. Um `map` também pode ser visto como um `array` comum. Enquanto as chaves em um `array` são sempre inteiros consecutivos $0, 1, \dots, n - 1$, onde n é o tamanho do `array`, as chaves em um `map` podem ser de qualquer tipo de dados e não precisam ser valores consecutivos. Como no `set`, o `map` é baseado em uma árvore binária balanceada (*red-black tree*) e acessar seus elementos é $O(\log n)$.

O código a seguir cria um `map` cujas chaves são strings e os valores são `int`:

```

1 map<string,int> m;
2 m["Paulo"] = 22;
3 m["Daniel"] = 38;
4 m["Bia"] = 19;
5 cout << m["Daniel"] << "\n"; // 38

```

Se o valor de uma chave for solicitado, mas que não existe no `map`, a chave será adicionada automaticamente ao `map` com um valor padrão do tipo (se for uma classe o construtor padrão é chamado). Por exemplo, no código a seguir, a chave `"Ana"` com valor 0 é adicionada ao `map`.

```

1 map<string,int> m;
2 cout << m["Ana"] << "\n";

```

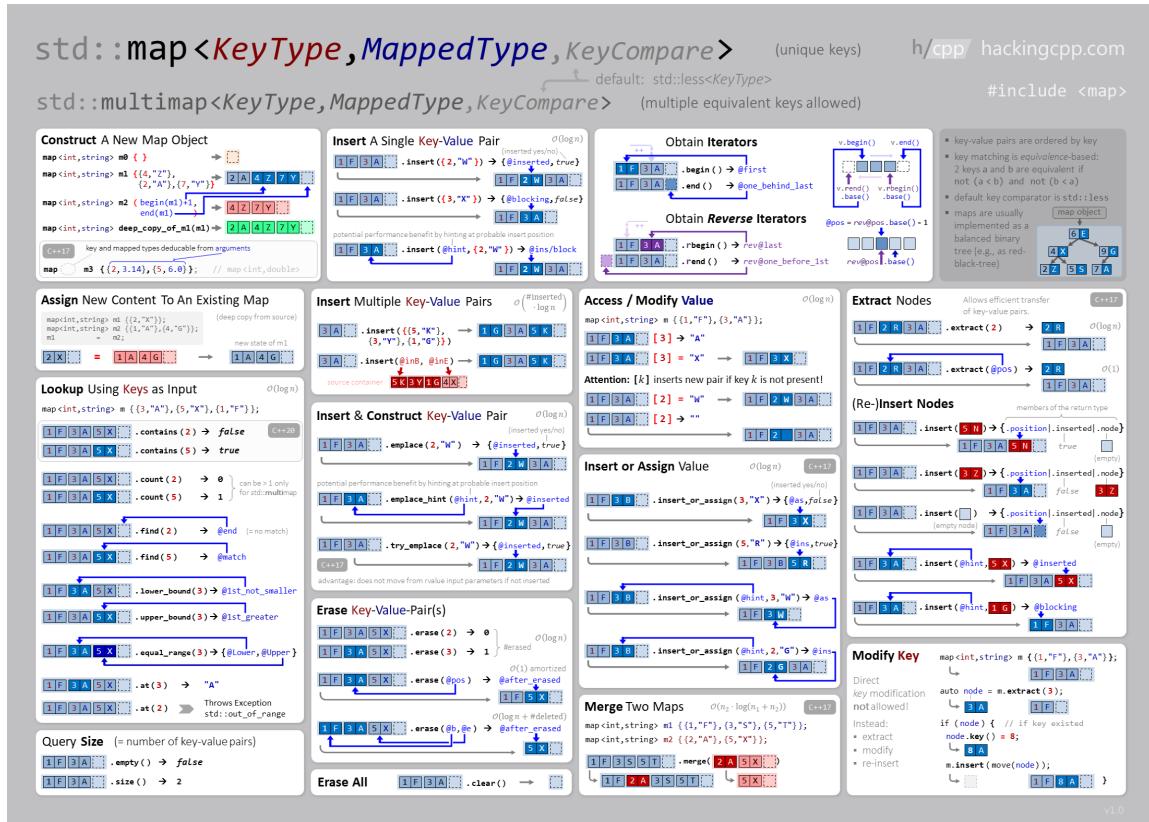
Veja mais alguns exemplos de utilização da estrutura:

```

1 map<string,int> m;
2 m["Paulo"] = 22;
3 m["Daniel"] = 38;
4 m["Bia"] = 19;
5
6 if (m.find("Ana") == m.end())
7     cout << "Não temos a idade da Ana registrada" << "\n";
8
9 cout << "Paulo tem " << m["Paulo"] << " anos\n";
10 cout << "Bia tem " << m["Bia"] << " anos\n";
11
12 for (auto x : m)
13     cout << x.first << " " << x.second << "\n";
14
15 m.erase("Paulo");
16 cout << "Paulo tem " << m["Paulo"] << " anos\n";
17
18 auto ultimo = --m.end();
19 cout << ultimo->first << " tem " << ultimo->second << " anos\n";

```

A figura abaixo mostra mais operações e uso do `map` e `multimap`.

Fonte: [Hacking C++](#)

Complemente sua leitura e seu conhecimento:

- Standard Associative Containers
- Map in C++ Standard Template Library (STL)
- Multimap in C++ Standard Template Library (STL)
- `std::map`
- `std::multimap`

unordered_set e unordered_multiset

Um `unordered_set` é um contêiner associativo que contém um conjunto de objetos exclusivos. Já um `unordered_multiset` permite cópia dos elementos. Ambas as estruturas são baseadas em tabela hash e suas operações possuem complexidade, em média, em tempo $O(1)$. Internamente, os elementos não seguem nenhuma ordem. Veja um exemplo de uso das estruturas:

```

1  unordered_set<int> s = {10, 5, -1, 20, 15, 5, 19};
2
3  cout << s.count(10) << "\n";
4  cout << s.count(4) << "\n";

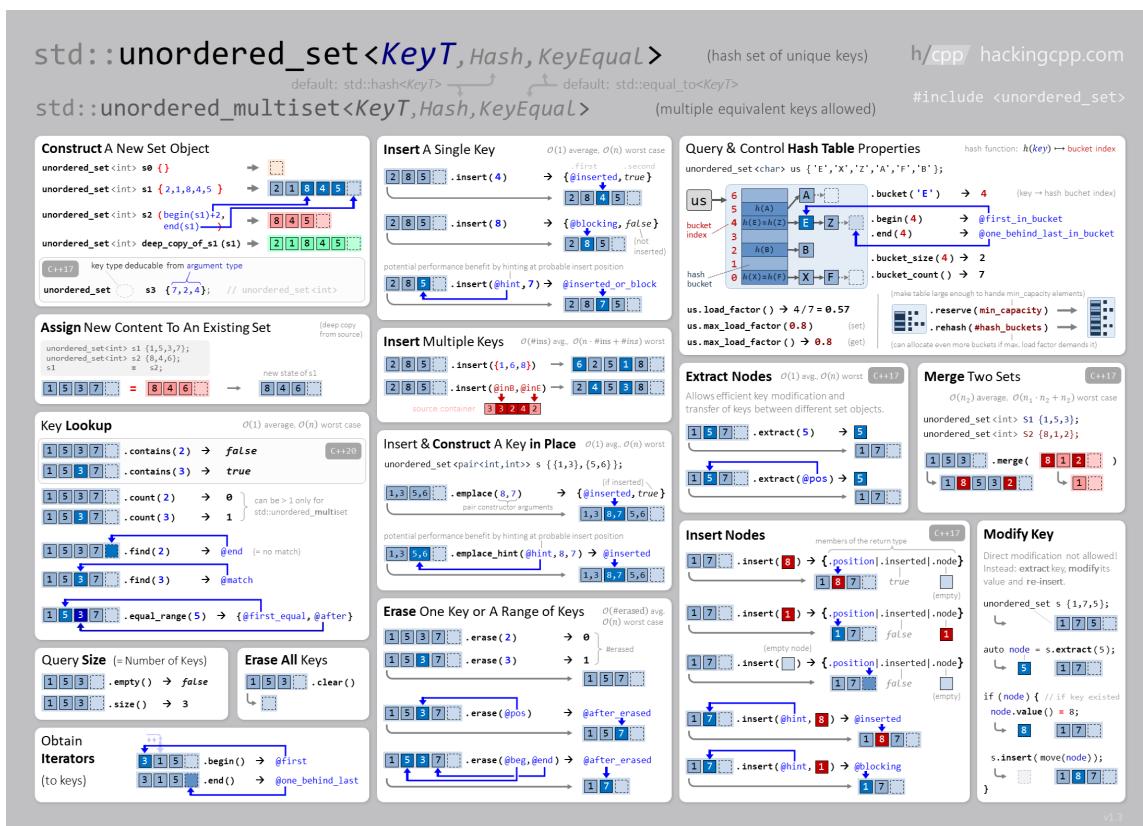
```

```

5   s.erase(10);
6   s.insert(4);
7   cout << s.count(10) << "\n";
8   cout << s.count(4) << "\n";
9   if (s.find(7) == s.end())
10      cout << "7 não está no set" << "\n";
11
12
13  for (auto x: s)
14      cout << x << " ";
15  cout << endl;
16
17  unordered_multiset<int> ms = {10, 5, -1, 20, 15, 5, 19};
18  for (auto x: ms)
19      cout << x << " ";
20  cout << endl;

```

A figura abaixo mostra mais operações e uso do `unordered_set` e `unordered_multiset`.



Fonte: [Hacking C++](https://www.hackingcpp.com/)

Complemente sua leitura e seu conhecimento:

- Standard Associative Containers
- `std::unordered_set`
- `std::unordered_multiset`

unordered_map e unordered_multimap

é um conjunto que consiste em pares de valores-chave

Um `unordered_map` é um contêiner associativo que contém pares de valores-chave com chaves exclusivas. Já um `unordered_multimap` permite múltiplas cópias das chaves. Ambas as estruturas são baseadas em tabela hash e suas operações possuem complexidade, em média, em tempo $O(1)$. Internamente, os elementos não seguem nenhuma ordem. Veja um exemplo de uso das estruturas:

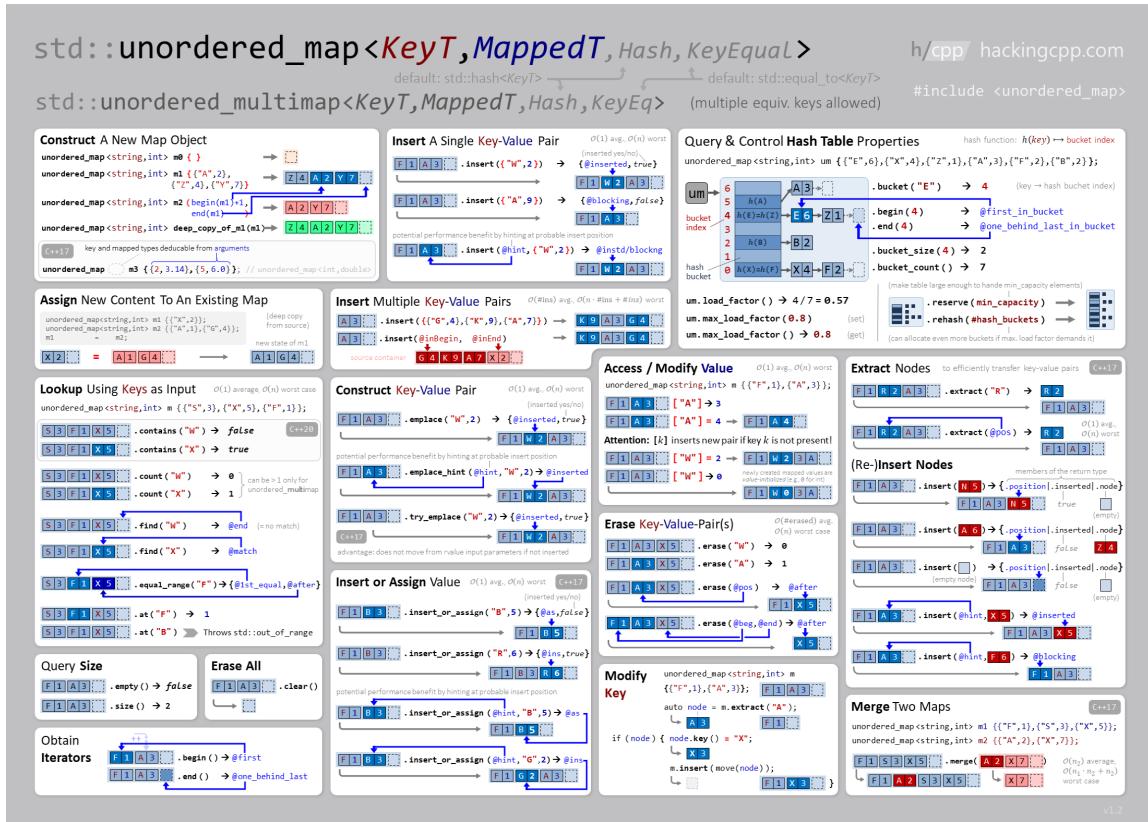
```

1  unordered_map<string, int> m;
2  m[ "Paulo" ] = 22;
3  m[ "Daniel" ] = 38;
4  m[ "Bia" ] = 19;
5  m[ "Bia" ] = 28; // Atualiza o valor da chave "Bia"
6
7  for (auto [chave, valor] : m)
8      cout << chave << " -> " << valor << "\n";
9
10 cout << endl;
11 unordered_multimap<string, int> mm;
12 mm.insert(make_pair("Paulo", 22)); // ①
13 mm.insert(make_pair("Paulo", 43));
14 mm.insert(make_pair("Bia", 19));
15 mm.insert(make_pair("Bia", 22));
16
17
18 for (auto [chave, valor] : mm)
19     cout << chave << " -> " << valor << "\n";

```

1. Não é possível usar o `operator[]` como no `map` e `unordered_map`.

A figura abaixo mostra mais operações e uso do `unordered_map` e `unordered_multimap`.

Fonte: [Hacking C++](#)

Complemente sua leitura e seu conhecimento:

- Standard Associative Containers
- std::unordered_map
- std::unordered_multimap

1. O texto dessa página são traduções e adaptações encontrados aqui: [1](#), [2](#), [3](#), [4](#) ←

Estrutura de Dados

Nesta parte, serão apresentados duas estruturas baseadas em árvores e uma implementação de `set` do GNU C++.

GNU C++ Policy-Based Sets

O compilador GNU g++ também suporta algumas estruturas de dados que não fazem parte da biblioteca padrão do C++. Essas estruturas são chamadas de *estruturas de dados baseadas em políticas* ([Policy-Based Data Structures - PBDS](#)). Para usar essas estruturas é necessário incluir o seguinte código:

```
1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
```

Alternativamente, pode-se simplificar o código anterior da seguinte forma:

```
1 #include <bits/extc++.h>
2 using namespace __gnu_pbds;
```

Após isso, é possível definir um **conjunto ordenado indexado** (entre as disponíveis, essa é a principal estrutura usada em competições de programação). Esta estrutura é como o `set`, mas com a possibilidade de acessar os elementos de acordo com a sua posição (índice), como em um `array`. O código abaixo, define um conjunto ordenado indexado usando *template*.

```
1 template <class T>
2 using indexed_set = tree<T, null_type, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;
```

Veja alguns exemplos de uso da definição anterior:

```
1 indexed_set<int> setInt;           // Conjunto ordenado indexado de
2 inteiros
3 indexed_set<double> setDouble;      // Conjunto ordenado indexado de
double
4 indexed_set<pair<int, int>> setPairInt; // Conjunto ordenado indexado de
pares de inteiros
5 indexed_set<Pessoa> setPessoa;       // Conjunto ordenado indexado de
Pessoas (precisa do operator<)
```

Além das operações comuns do `set` (`insert`, `erase`, `size`, `clear`, `find`, `begin`, `end`), ainda existem duas funções muito úteis e que justificam o uso do PBDS:

- `find_by_order(k)` : retorna um iterador para o k -ésimo menor elemento (contando a partir de zero) do conjunto em tempo $O(\log n)$;
- `order_of_key(x)` : retorna ao número de itens **estritamente menores** que o `x` em tempo $O(\log n)$.

Veja um exemplo completo de utilização do PBDS (Fonte: [pbds.cpp](#)):

```

1 #include <bits/stdc++.h>
2 #include <bits/extc++.h> // pbds
3 using namespace std;
4 using namespace __gnu_pbds;
5
6 template <class T>
7 using indexed_set = tree<T, null_type, less<T>, rb_tree_tag,
8 tree_order_statistics_node_update>;
9
10 int main() {
11     int n = 9;
12     vector<int> A = {71, 10, 2, 4, 23, 7, 10, 15, 23, 50, 65, 71};
13     indexed_set<int> iset;
14     for (int i = 0; i < A.size(); ++i)                      // O(n log n)
15         iset.insert(A[i]);
16     cout << "Set size: " << iset.size() << "\n";    // 9
17
18     for (auto x: iset)
19         cout << x << " ";
20     cout << "\n";
21
22     // O(log n) select
23     cout << *iset.find_by_order(0) << "\n";           // 1-smallest = 2
24     cout << *iset.find_by_order(n-1) << "\n";          // 9-smallest/largest = 71
25     cout << *iset.find_by_order(4) << "\n";            // 5-smallest = 15
26     // O(log n) rank
27     cout << iset.order_of_key(2) << "\n";              // index 0 (rank 1)
28     cout << iset.order_of_key(71) << "\n";             // index 8 (rank 9)
29     cout << iset.order_of_key(15) << "\n";            // index 4 (rank 5)
30
31     iset.erase(2);
32     cout << "Set size: " << iset.size() << "\n";    // 8
33
34     return 0;
}

```

Complemente sua leitura e seu conhecimento:

- [Ordered Set and GNU C++ PBDS \(GeeksforGeeks\)](#)
- [C++ STL: Policy based data structures](#)
- [PBDS Examples](#)

Fenwick (Binary Indexed) Tree

Considere o seguinte problema:

Você possui n caixas com bolinhas de gude e deseja:

1. Adicionar bolinhas de gude a caixa i ;
2. Saber a quantidade de bolinhas de gude que existem da caixa a até a caixa b , ou seja, a soma das bolinhas nas caixas $[a, a + 1, a + 2, \dots, b]$.

O objetivo é implementar essas duas operações de forma eficiente. Esse problema é conhecido como **Range Sum Queries (RSQ)**.

Uma abordagem ingênua, mostrada abaixo, possui $O(1)$ para a operação 1 e $O(n)$ para a 2. Se forem feitas m operações 2, no pior caso, a complexidade de tempo será $O(n * m)$.

```

1  vector<ll> caixas(n, 0);
2
3  void add(vector<ll>& caixas, ll i, ll n) { // O(1)
4      caixas[i] += n;
5  }
6
7  void sum(vector<ll>& caixas, ll a, ll b){ // O(n)
8      ll s = 0;
9      for(ll i = a; i <= b; i++)
10         s += caixas[i];
11     return s;
12 }
```

Para responder *range sum queries* em sequências dinâmicas com melhor complexidade é preciso usar estruturas mais sofisticadas, que permitam a atualização das somas pré-computadas de forma eficiente. Uma destas estruturas é a **Binary Indexed Tree (BIT ou Fenwick Tree)**, proposta por [Peter M. Fenwick em 1994](#). Usando BIT, ambas as operações são feitas em $O(\log n)$.

A ideia da BIT é que, assim como um número pode ser representado como uma soma de algumas potências de dois, uma soma cumulativa pode ser representada como uma soma de algumas somas cumulativas parciais. Para isso, cada índice i no array de soma cumulativa é responsável pela soma cumulativa do índice i até $(i - (1 << r) + 1)$, onde r representa a posição do último bit 1 da representação binária de i ($p(i) = (1 << r)$ também pode ser visto como a maior potência de 2 que divide i). Por exemplo, o índice $15_{10} = 1111_2$ ficará responsável pela soma do intervalo $15 - (1 << 0) + 1 = 15 - 1 + 1$ a 15. Já o índice $12_{10} = 1100_2$ ficará responsável pela soma do intervalo $12 - (1 << 3) + 1 = 12 - 8 + 1 = 9$ a 12, e assim por diante. A tabela abaixo mostra a responsabilidade dos índices 1 a 16.

Índice	Responsável pela soma parcial do intervalo
16	[01, 16]

Índice	Responsável pela soma parcial do intervalo
15	[15, 15]
14	[13, 14]
13	[13, 13]
12	[09, 12]
11	[11, 11]
10	[09, 10]
09	[09, 09]
08	[01, 08]
07	[07, 07]
06	[05, 06]
05	[05, 05]
04	[01, 04]
03	[03, 03]
02	[01, 02]
01	[01, 01]

Perceba que todo índice ímpar fica responsável por um intervalo de tamanho 1 (porquê?). Já os índices i que são potência de 2, ficam responsáveis pelo intervalo $[1, i]$.

Tipicamente, a Fenwick Tree é implementada usando um array (`vector`), digamos `ft`, de modo que:

```
1 ft[i] = sum(i - p(i) + 1, i);
```

Assim, cada posição `i` contém a soma dos valores de um intervalo do array original cujo comprimento é `p(i)` e que termina na posição `i`. Por exemplo, como `p(6) = 2`, `ft[6]` contém

o valor de `sum(5, 6)`.

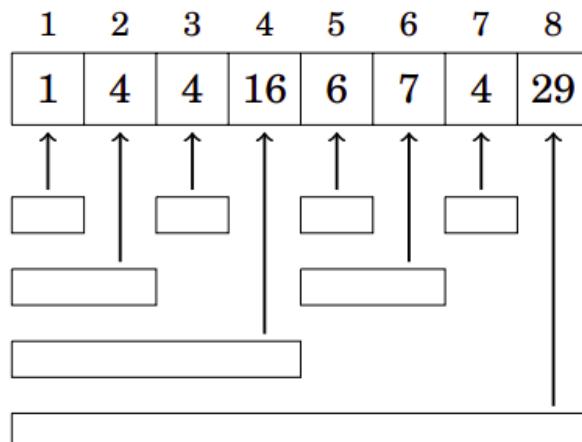
Considere o seguinte array original:

1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

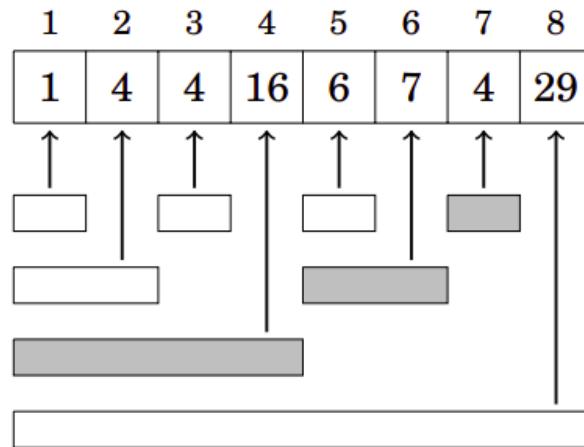
A Fenwick Tree correspondente é a seguinte:

1	2	3	4	5	6	7	8
1	4	4	16	6	7	4	29

A figura abaixo mostra mais claramente como cada valor na BIT corresponde a um intervalo no array original:



Usando uma BIT, qualquer valor de `sum(1, i)` pode ser calculado em tempo $O(\log n)$, pois um intervalo $[1, i]$ sempre pode ser dividido em intervalos $O(\log n)$ cujas somas são armazenadas na árvore. Por exemplo, o intervalo $[1, 7]$ consiste nos seguintes intervalos:



Assim, pode-se calcular a soma correspondente da seguinte forma:

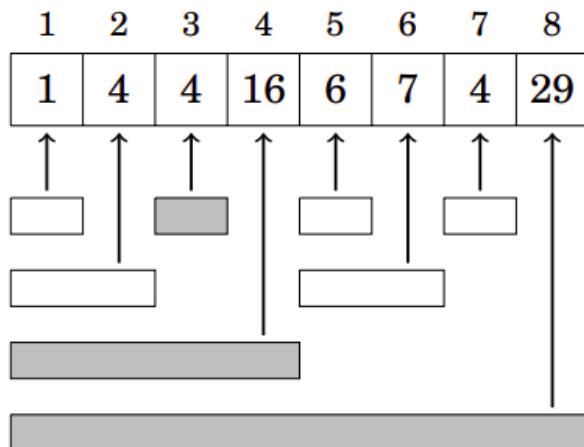
$$\text{sum}(1, 7) = \text{sum}(1, 4) + \text{sum}(5, 6) + \text{sum}(7, 7) = 16 + 7 + 4 = 27$$

Para calcular o valor de $\text{sum}(a, b)$ onde $a > 1$, pode-se usar o mesmo truque que usado na soma de prefixo:

$$\text{sum}(a, b) = \text{sum}(1, b) - \text{sum}(1, a - 1).$$

Como pode-se calcular tanto $\text{sum}(1, b)$ quanto $\text{sum}(1, a - 1)$ em tempo $O(\log n)$, a complexidade de tempo total é $O(\log n)$.

Além de calcular a soma parcial, também é possível/necessário atualizar o valor de uma determinada posição no array original. Após a atualização, alguns valores na BIT devem ser atualizados. Por exemplo, se o valor na posição 3 mudar, as somas dos seguintes intervalos mudam:



Como cada elemento do array pertence a $O(\log n)$ intervalos na BIT, basta atualizar os $O(\log n)$ valores na árvore.

Implementação

O primeiro passo é definir como calcular o valor de $p(i)$, ou seja, a maior potência de 2 que divide i . Este valor é conhecido como o **bit menos significativo**, abreviado *lsb*, da representação binária de i . O função abaixo calcula esse valor:

```

1 int p(int i) {
2     int bit = 0;
3     while ((i >> bit) & 1 == 0) {
4         ++bit;
5     }
6     return (1 << bit);
7 }
```

Essa função possui complexidade $O(\log n)$. Entretanto, usando [manipulação de bits](#), pode-se encontrar esse valor de forma mais eficiente. Lembre-se que um número negativo é representado usando complemento de 2, ou seja, pega-se a representação binária de x , inverte-se os bits e adiciona 1 (ignorando o resto final). Por exemplo, considere $x = 108$ e que os valores são representados em 8 bits:

```

1 x = 01101100
2 ~x = 10010011
3 -x = 10010100
```

Observe que os bits mais significativos que o *lsb* são diferentes para x e $-x$, enquanto o *lsb* e os Os seguintes são os mesmos. Portanto, $x \& -x$ dá a resposta desejada, ou seja: $x \& -x = 00000100$. Com essa estratégia, pode-se calcular o valor de *lsb*, consequentemente, o valor de $p(i)$ em $O(1)$:

```

1 long long lsb(long long i) { // ①
2     return i & -i;
3 }
```

1. Normalmente, essa função é feita como uma macro: `#define lsb(i) ((i) & -(i))`

A funções a seguir calculam o valor de $\text{sum}(1, i)$ e $\text{sum}(a, b)$, respectivamente:

```

1 long long sum(long long i) {
2     long long s = 0;
3     while (i >= 1) {
4         s += ft[i];
5         i -= lsb(i);
6     }
7     return s;
8 }
9 long long sum(long long a, long long b) {
```

```

10     return sum(b) - sum(a-1);
11 }
```

A seguinte função aumenta o valor do array na posição i em x (x pode ser positivo ou negativo):

```

1 void add(int i, int x) { // Normalmente, essa função também é chamada de
2     update
3     while (i < ft.size()) {
4         ft[i] += x;
5         i += lsb(i);
6     }
}
```

Para construir a BIT a partir de um array original, uma primeira estratégia é iniciar o array `ft` com valor 0 em todas as posições. Em seguida, para cada posição i , chama-se a função `add` passando o valor do array original como valor a ser incrementado, o que daria resultaria em $O(n \log n)$ operações. Veja a função abaixo:

```

1 void build(const vector<ll>& a) {
2     ft.assign(a.size() + 1, 0);
3     for (ll i = 1; i < a.size(); i++)
4         add(i, a[i]);
5 }
```

Entretanto, note que após atualizar o valor de `ft[i]`, pode-se verificar se o "pai" de i é menor que N (tamanho do array). Se for, atualiza-se o "pai" também. Dessa forma, temos uma função com complexidade $O(n)$.

```

1 void build(const vector<ll>& a) {
2     ll m = (ll)a.size() - 1;
3     ft.assign(m + 1, 0);
4     for (ll i = 1; i <= m; i++) {
5         ft[i] += a[i];
6         if (i + lsb(i) <= m)
7             ft[i + lsb(i)] += a[i];
8     }
9 }
```

Também é possível encontrar o menor índice i tal que soma acumulativa do intervalo $[1..i] \geq k$. Como as somas acumulativas dos valores estão ordenadas, pode-se usar a busca binária para encontrar tal índice. Basicamente, testa-se o índice do meio $m = n/2$ do intervalo inicial $[1..n]$ e verifica se `sum(1, i)` é menor que k ou não. Para cada valor de m , tem-se $O(\log n)$ operações, logo, essa função terá complexidade $O(\log n \times \log n) = O(\log^2 n)$. O código abaixo ilustra tal função:

```

1 ll select(ll k) {
2     ll ini = 1, fim = ft.size() - 1;
3     for (int i = 0; i < 30; ++i) { // ①
4         ll m = (ini + fim) / 2;
5         if (sum(1, m) < k) ini = m;
6         else fim = m;
```

```

7     }
8     return fim;
9 }
```

$1 \cdot 2^{30} > 10^9$, normalmente é suficiente.

O código abaixo mostra uma implementação completa de uma BIT. Note que foi usado uma versão baseada em orientação à objetos, mas isso não é necessário.

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 typedef long long ll;
5 typedef vector<ll> vll;
6
7 #define lsb(i) ((i) & -(i))
8
9 class FenwickTree {
10 private:
11     vll ft;
12 public:
13     FenwickTree(ll m) { ft.resize(m + 1, 0); }
14     FenwickTree(const vll &a) { build(a); }
15
16     ll sum(ll i) {
17         ll s = 0;
18         while (i >= 1) {
19             s += ft[i];
20             i -= lsb(i);
21         }
22         return s;
23     }
24     ll sum(ll a, ll b) {
25         return sum(b) - sum(a-1);
26     }
27
28     void add(ll i, ll x) {
29         while (i < (ll)ft.size()) {
30             ft[i] += x;
31             i += lsb(i);
32         }
33     }
34
35     void build(const vll& a) {
36         ll m = (ll)a.size() - 1;
37         ft.resize(m + 1, 0);
38         for (ll i = 1; i <= m; i++) {
39             ft[i] += a[i];
40             if (i + lsb(i) <= m)
41                 ft[i + lsb(i)] += ft[i];
42         }
43     }
44
45     ll select(ll k) {
46         ll ini = 1, fim = (ll)ft.size() - 1;
47         for (int i = 0; i < 30; ++i) {
48             ll m = ini + (fim - ini) / 2;
```

```

49         if (sum(1, m) < k) ini = m;
50     }
51 }
52 return fim;
53 }
54 ;
55
56
57 int main() {
58     //      1  2  3  4  5  6  7  8
59     vll f = {0, 1, 3, 4, 8, 6, 1, 4, 2}; // Índice 0 não é utilizado
60     FenwickTree ft(f);
61     cout << ft.sum(1, 8) << "\n";    // ft[8] = 29
62     cout << ft.sum(1, 3) << "\n";    // ft[3] + ft[2] = 4 + 4 = 8
63     cout << ft.sum(1, 7) << "\n";    // ft[7] + ft[6] + ft[4] = 16 + 7 + 4 =
64     27
65     cout << ft.sum(2, 4) << "\n";    // sum(1, 4) - sum(1, 1) = ft[4] - ft[1]
66     = 16 - 1 = 15
67     cout << ft.sum(6, 8) << "\n";    // sum(1, 8) - sum(1, 5) = ft[8] -
68     (ft[5] + ft[4]) = 29 - (6 + 16) = 7
69     ft.add(8, 2);                    // Aumenta em duas unidades o valor da
70     posição 8
71     cout << ft.sum(1, 8) << "\n";    // ft[8] = 31
72     ft.add(8, -2);                  // Reduz em duas unidades o valor da
73     posição 8
74     cout << ft.sum(1, 8) << "\n";    // ft[8] = 29
75     ft.add(8, -f[8]);                // Zera o valor da posição 8
76     cout << ft.sum(1, 8) << "\n";    // ft[8] = 27
77     cout << ft.select(4) << "\n";    // Índice 2, sum(1, 2) == 4, que é >= 4
    cout << ft.select(5) << "\n";    // Índice 3, sum(1, 3) == 8, que é >= 8
    cout << ft.select(25) << "\n";   // Índice 7, sum(1, 7) == 27, que é >=
25
    return 0;
}

```

Quando usar?

Para utilizar uma árvore de Fenwick para realizar a operação \odot em um intervalo de índices $[i, j]$ da sequência a_k , é necessário que esta operação tenha duas propriedades:

1. Associatividade: para quaisquer $x, y, z \in a_k$, deve valer que: $(x \odot y) \odot z = x \odot (y \odot z)$;
2. Invertibilidade: para qualquer $x \in a_k$, deve existir um valor y tal que $x \odot y = I$, onde I é o elemento neutro/identidade da operação \odot .

Como exemplos de operações que têm ambas propriedades tem-se a adição e a multiplicação de racionais, a adição de matrizes e o ou exclusivo (xor).

Material complementar

- [Binary Indexed Tree or Fenwick Tree \(GeeksforGeeks\)](#)

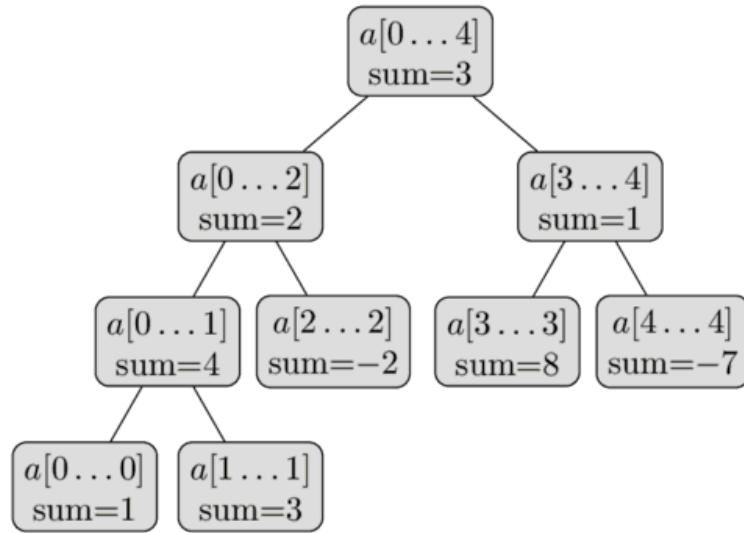
- [Fenwick Tree](#)
- [Binary Indexed Trees \(Topcoder\)](#)
- [Fenwick Tree \(Binary Index Tree\) - Quick Tutorial and Source Code Explanation](#)
- [Tutorial: Binary Indexed Tree \(Fenwick Tree\)](#)
- [Binary Indexed \(Fenwick\) Tree - VisuAlgo](#)

Segment Tree

Segment tree (Árvore de Segmento ou *SegTree*) é outra estrutura de dados para lidar com problemas de consulta em intervalos. O que torna as SegTrees poderosas é sua capacidade de fazer atualização e consulta em intervalos com complexidade $O(\log n)$, além do tipo da consulta ser bem abrangente. Entretanto, comparada a BIT, uma SegTree requer mais memória e é um pouco mais difícil de implementar.

A ideia da SegTree é a seguinte: cria-se uma árvore de forma que cada nó representa a informação que deseja-se saber a respeito de um segmento do vetor (por exemplo, a soma). Cada nó (exceto as folhas) possui dois filhos, um filho representa a metade esquerda do intervalo e o outro, a metade direita. Esse processo repete-se (recursivamente) até que os intervalos atinjam tamanho 1. Assim, considerando o *Range Sum Queries* (RSQ), o nó raíz da árvore armazena a soma de todo o array, ou seja, a soma do segmento $a[0 \dots n - 1]$. Em seguida, essa segmento é dividido em dois: o filho da esquerda fica responsável pelo calcular/armazenar a soma da metade da esquerda do segmento (ou seja, $a[0 \dots n/2]$) e o filho da direita responsável pela metade da direita do segmento (ou seja, $a[n/2 + 1 \dots n - 1]$). Cada um desses nós é novamente dividido e calcula-se a soma de cada novo intervalo. Repete-se o processo até todos os segmentos atingirem tamanho 1. Nesse [link](#) você encontra uma demonstração visual de como a SegTree funciona.

Veja uma representação visual de uma SegTree para o array `a = [1, 3, -2, 8, -7]`:



Fonte: [Algorithms for Competitive Programming](#)

Note que o primeiro nível da árvore contém um único nó (a raiz), o segundo nível contém 2 nós, no terceiro contém 4, e assim por diante, até que o número de nós alcance n . Portanto, o número de nós, no pior caso, pode ser estimado pela soma

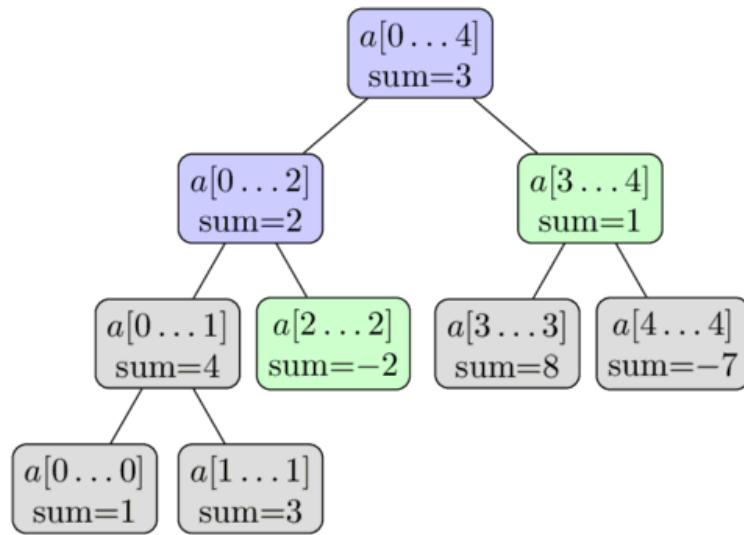
$$1 + 2 + 4 + \dots + 2^{\lceil \log_2 n \rceil} = 2^{\lceil \log_2 n \rceil + 1} < 4n.$$

Consultas

Suponha que deseja-se fazer uma consulta (por exemplo, calcular a soma, valor mínimo/máximo, etc.) no intervalo l e r , ou seja, deseja-se fazer um consulta no segmento $a[l \dots r]$. Para isso, deve-se percorrer a árvore de segmentos e usar os valores pré-computados dos segmentos. Considere que a consulta esteja no vértice que cobre o segmento $a[tl \dots tr]$. Existem três possibilidades:

1. O nó está fora do intervalo de interesse, ou seja, `tr < l || r < tl`. Retorne um valor neutro que não afete a consulta (por exemplo, se a operação for a soma, retorne 0);
2. O nó está completamente incluído no intervalos de interesse, ou seja, `tl >= l && tr <= r`. Retorne a informação armazenada no nó;
3. O nó está parcialmente contido no intervalo de interesse, ou seja, `(l <= tr && tr <= r) || (l <= tl && tl <= r)`. Então, a consulta continua nos nós filhos.

A figura abaixo ilustra esse processo para encontrar a soma do segmento $a[2 \dots 4]$ para o array `a = [1, 3, -2, 8, -7]`. Os nós coloridos serão visitados e usa-se os valores pré-calculados dos nós verdes. Assim, o resultado será $-2 + 1 = -1$. O número de nós visitados é proporcional à altura da árvore, ou seja, $O(\log n)$.



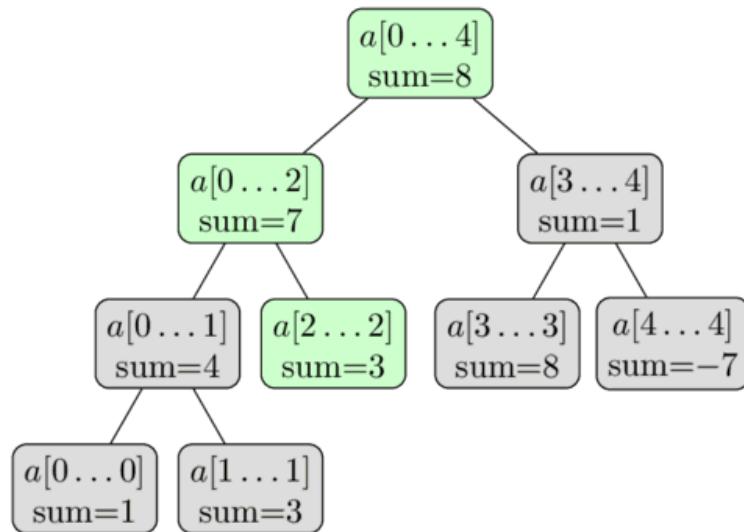
Fonte: [Algorithms for Competitive Programming](#)

Atualizações

Fazer uma atualização no array original (por exemplo, fazer `a[i] = x`) implica que a SegTree também deve ter alguns nós atualizados, de modo que ela corresponda ao novo array modificado. Perceba que cada nível de uma SegTree forma uma partição do array. Portanto, um elemento `a[i]` apenas contribui para **um** segmento de cada nível. Assim, apenas $O(\log n)$ nós necessitam ser atualizadas.

A atualização dos elementos pode ser facilmente implementada usando uma função recursiva. A função passa pelo nó atual da árvore e recursivamente chama a si mesma com um dos dois nós filhos (aquele que contém `a[i]` em seu segmento) e, em seguida, recalcula seu valor.

Considerando a SegTree do array `a = [1, 3, -2, 8, -7]`, ao fazer `a[2] = 3`, os nós verdes são os nós visitados e atualizados.



Fonte: [Algorithms for Competitive Programming](#)

Implementação

A primeira função a ser analisada/implementada é a `build`, responsável por criar a SegTree. Assim como na BIT, aqui também pode-se usar um array para armazenar os nós da SegTree (considere que o array se chama `st`). O índice 0 é a raiz e os filhos da esquerda e direita do índice i são os índices $2 \times i + 1$ e $2 \times i + 2$, respectivamente.

A raiz da SegTree representa o segmento completo `[0, n-1]` do array `A`. Para cada segmento `[L, R]` armazenado no índice i , onde $L \neq R$, o segmento é dividido no sub-segmento `[L, (L+R)/2]` (armazenado no índice $2 \times i + 1$) e sub-segmento `[(L+R)/2 + 1, R]` (armazenado no índice $2 \times i + 2$). Essa divisão continua até que o segmento tenha tamanho 1 (isto é, $L == R$), o que indica que chegou-se em um nó folha e, pela definição de SegTree, esse nó deve receber o valor correspondente de `A`, ou seja, `st[i] = A[L]` ou `st[i] = A[R]`.

A função `build` abaixo ilustra o processo de construção de uma SegTree a partir do array `A`. Perceba que foi usada a função `funcao` para fazer a atualização dos nós da árvore. Nesse exemplo, a SegTree será responsável por realizar a soma dos elementos (RSQ).

```

1 #define filhoEsq(i) (2*(i) + 1)
2 #define filhoDir(i) (2*(i) + 2)
3
4 long long funcao(long long a, long long b) { // Função usada na SegTree:
5     soma
6         return a + b;
7 }
8
9 void build(long long i, long long L, long long R) {
10     if (L > R)
11         return;
  
```

```

12     //Nó (folha) que deve conter o valor de A[L] ou A[R]
13     if (L == R)
14         st[i] = A[L];
15     else {
16         //Recurativamente constroi-se a SegTree
17         long long m = L + (R - L) / 2;
18         build(filhoEsq(i), L, m);
19         build(filhoDir(i), m + 1, R);
20         //Após atualizar o valor dos nós filhos de 'i', atualiza-se o valor
21     do nó 'i'
22         st[i] = funcao(st[filhoEsq(i)], st[filhoDir(i)]);
    }
}

```

A próxima função a ser analisada/implementada é a `query`, responsável por fazer as consultas. Ao analisar o nó i , responsável pelo intervalo $[tl, tr]$, como explicado na seção [Atualizações](#), existem três possibilidades. O código abaixo mostra uma implementação da função `query`, ela recebe como parâmetros informações sobre o nó/segmento atual (ou seja, o índice i e os limites tl e tr) e também os limites da consulta, l e r . Para fazer uma consulta no intervalo $[l, r]$, a função deve ser chamada da seguinte forma: `query(0, 0, n - 1, l, r)`, onde n representa o número de elementos de `A`.

```

1 long long query(long long i, long long tl, long long tr, long long l, long
2 long r){
3     //O nó i está fora do intervalo de interesse.
4     //Retorne o elemento neutro que não afeta a consulta.
5     if(tr < l || r < tl)
6         return 0; // Elemento neutro da RSQ
7
8     // O nó i está completamente incluído no intervalo de interesse.
9     // Retorne a informação contida no nó.
10    if(tl >= l and tr <= r)
11        return st[i];
12
13    // Se a execução chegou nesse ponto, significa que o nó i está
14    // parcialmente
15    // contido no intervalo de interesse. Então, continua-se a procura nos
16    // nós filhos.
17    long long m = tl + (tr - tl) / 2;
18    return funcao(query(filhoEsq(i), tl, m, l, r), query(filhoDir(i), m + 1,
19    tr, l, r));
}

```

A função `update` é responsável por fazer atualizações no array `A` e, consequentemente, na SegTree. O código abaixo mostra uma implementação da função `update`, ela recebe como parâmetros informações sobre o nó/segmento atual (ou seja, o índice no e os limites tl e tr), o índice i que se deseja atualizar e o novo valor a ser armazenado na posição i . Para fazer `A[i] = x`, a função deve ser chamada da seguinte forma: `update(0, 0, n - 1, i, x);`, onde n representa o número de elementos de `A`.

```

1 void update(long long no, long long tl, long long tr, long long i, long long
2 novoValor){
3     // Índice cujo valor deve ser atualizado
}

```

```

4     if(tl == i and tr == i){
5         st[no] = novoValor;
6         A[i] = novoValor;
7         return;
8     }
9
10    // O intervalo não contém o índice a ser atualizado
11    if(tl > i or tr < i)
12        return;
13
14    // O intervalo contém o índice, mas deve-se chegar no nó específico.
15    // O processo é repetido recursivamente nos filhos.
16    long long m = tl + (tr - tl) / 2;
17    update(filhoEsq(no), tl, m, i, novoValor);
18    update(filhoDir(no), m + 1, tr, i, novoValor);
19
20    // Após atualizar os filhos (esquerdo ou direito), deve-se atualizar o
21    valor do nó.
22    st[no] = funcao(st[filhoEsq(no)], st[filhoDir(no)]);
23 }
```

O código abaixo mostra uma implementação completa de uma SegTree. Note que foi usado uma versão baseada em orientação à objetos, mas isso não é necessário.

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 typedef long long ll;
5 typedef vector<ll> vll;
6
7 #define filhoEsq(i) (2*(i) + 1)
8 #define filhoDir(i) (2*(i) + 2)
9
10 class SegTree {
11 private:
12     vll st, A;
13     ll size;
14
15     ll el_neutro = 0;
16
17     ll funcao(ll a, ll b) { // Função usada na SegTree: soma
18         return a + b;
19     }
20
21     ll query(ll no, ll tl, ll tr, ll l, ll r){
22         // O nó está fora do intervalo de interesse.
23         // Retorne o elemento neutro que não afeta a consulta.
24         if(tr < l || r < tl)
25             return el_neutro;
26
27         // O nó está completamente incluído no intervalo de interesse.
28         // Retorne a informação contida no nó.
29         if(tl >= l and tr <= r)
30             return st[no];
31
32         // Se chegarmos aqui, é porque esse nó está parcialmente contido no
33         // intervalo de interesse.
34 }
```

```

34         // Então, continuamos procurando nos filhos.
35         ll m = tl + (tr - tl) / 2;
36         return funcao(query(filhoEsq(no), tl, m, l, r), query(filhoDir(no),
37 m + 1, tr, l, r));
38     }
39
40     void update(ll no, ll tl, ll tr, ll i, ll novoValor){
41         // Chegamos no índice que queremos atualizar o valor
42         if(tl == i and tr == i){
43             st[no] = novoValor;
44             A[i] = novoValor;
45             return;
46         }
47
48         // O intervalo que estamos não contém o índice que queremos
49         atualizar, retorne
50         if(tl > i or tr < i)
51             return;
52
53         // O intervalo contém o índice, mas temos que chegar no nó
54         específico.
55         // Repetimos o processo recursivamente nos filhos.
56         ll m = tl + (tr - tl) / 2;
57         update(filhoEsq(no), tl, m, i, novoValor);
58         update(filhoDir(no), m + 1, tr, i, novoValor);
59
60         // Após atualizar o filho (esquerdo ou direito), precisamos
61         atualizar o valor do nó.
62         st[no] = funcao(st[filhoEsq(no)], st[filhoDir(no)]);
63     }
64
65     void build(ll no, ll L, ll R) {
66         //Chegamos no nó que deve conter o valor de A[L] ou A[R]
67         if (L == R)
68             st[no] = A[L];
69         else {
70             //Recursivamente construimos a SegTree
71             ll m = L + (R - L) / 2;
72             build(filhoEsq(no), L, m);
73             build(filhoDir(no), m + 1, R);
74             st[no] = funcao(st[filhoEsq(no)], st[filhoDir(no)]);
75         }
76     }
77
78     public:
79     SegTree(ll n): st(4*n, 0), A(n, 0), size(n){}
80     SegTree(const vll &a) {
81         A = a;
82         st.resize(4*A.size(), 0);
83         size = A.size();
84         build(0, 0, size - 1);
85     }
86     ll query(ll l, ll r) {
87         return query(0, 0, size - 1, l, r);
88     }
89     void update(ll i, ll x){
90         update(0, 0, size - 1, i, x);
91     }
92 };

```

```

93
94 int main() {
95     //      0  1  2  3  4  5  6  7  8
96     vll a = {0, 1, 3, 4, 8, 6, 1, 4, 2};
97     SegTree st(a);
98     cout << st.query(0, 8) << "\n";    // 29
99     cout << st.query(0, 0) << "\n";    // 0
100    cout << st.query(0, 3) << "\n";    // 8
101    cout << st.query(0, 7) << "\n";    // 27
102    cout << st.query(2, 4) << "\n";    // 15
103    cout << st.query(6, 8) << "\n";    // 7
104
105    st.update(0, 10);                  // A[0] = 10
106
107    cout << st.query(0, 8) << "\n";    // 39
108    cout << st.query(0, 0) << "\n";    // 10
109    cout << st.query(0, 3) << "\n";    // 18
110
111    st.update(8, 7);                  // A[8] = 7;
112
113    cout << st.query(8, 8) << "\n";    // 7
114    cout << st.query(0, 8) << "\n";    // 44
115
116    return 0;
117 }
```

Atualizações em Intervalos (Lazy Propagation)

Como visto, a função `update` atualiza apenas uma posição do array, mas em alguns casos é necessário alterar o valor de um intervalo do array, por exemplo, alterar os elementos da posição l a r para x . Uma estratégia ingênua é chamar a função `update` para cada posição do array (complexidade $O(n)$). Note que se for necessário alterar todos os elementos do array, uma alternativa é alterar apenas a raiz e postergar a atualização dos outros nós. Dessa forma, os nós filhos são atualizados somente quando for realmente necessário. Esse processo é conhecido como **Lazy Propagation**. É importante destacar que, para implementar *lazy propagation*, cada tipo de SegTree vai requerer uma implementação um pouco diferente, por isso, será importante entender como essa estratégia funciona.

Na SegTree com *Lazy Propagation*, cada segmento (nó) terá também um valor **lazy** associado. Uma atualização dividirá o segmento da mesma forma feita anteriormente e definirá o valor `lazy` em cada segmento alcançado. Assim, sempre que um nó for analisado, seja por meio de uma atualização ou uma consulta, uma função, `propagate`, será chamada. Essa função fica responsável por atualizar a informação no segmento atual e passar o valor `lazy` para seus filhos.

Considere o problema de saber a soma dos elementos de um intervalo e que uma atualização de intervalo significa alterar o valor de todos os elementos do intervalo para um determinado valor. Para isso é necessário, além do vetor que armazena a SegTree, outros dois vetores: `lazy` e `marked`. O vetor `lazy` fica responsável por armazenar o valor do segmento que ainda não foi

propagado para os nós filhos. Já o vetor `marked` (vetor booleano) indica se há uma atualização para ser feita no nó.

A função de propagação (`propagate`) é a função que atualiza o valor de um nó e posterga a atualização para os filhos. Essa função deve ser chamada toda vez que um nó for analisado (seja por uma atualização ou uma consulta). O código abaixo ilustra essa função.

```

1 void propagate(ll p, ll L, ll R) { // (1)
2     // Verifica se o nó precisa ser atualizado
3     if (marked[p]) {
4         // O valor do nó será: número de elementos que esse intervalo
5         // representa vezes o novo valor de cada elemento do intervalo
6         st[p] = (R - L + 1) * lazy[p];
7         // Se o nó não for uma folha, propague a atualização para os filhos.
8         if (L != R) {
9             lazy[filhoDir(p)] = lazy[filhoEsq(p)] = lazy[p];
10            marked[filhoDir(p)] = marked[filhoEsq(p)] = true;
11        }
12        // Não é mais necessário atualizar esse nó
13        marked[p] = false;
14    }
}

```

1. Parâmetros:

- `p` : nó atual
- `L` : limite inferior do intervalo que `p` representa (inclusivo)
- `R` : limite superior do intervalo que `p` representa (inclusivo)

A função abaixo altera o valor dos elementos do intervalo `[l, r]` para `novoValor`.

```

1 void update(ll no, ll tl, ll tr, ll l, ll r, ll novoValor) { // (1)
2     //Precisamos propagar a possível atualização do nó
3     propagate(no, tl, tr);
4
5     // Chegamos no índice que queremos atualizar o valor
6     if (tl >= l and tr <= r) {
7         lazy[no] = novoValor;
8         marked[no] = true;
9         propagate(no, tl, tr);
10        return;
11    }
12
13    // O intervalo que estamos não contém o índice que queremos atualizar,
14    retorno
15    if (tl > r or tr < l)
16        return;
17
18    // O intervalo contém o índice, mas temos que chegar no nó específico.
19    // Repetimos o processo recursivamente nos filhos.
20    ll m = tl + (tr - tl) / 2;
21    update(filhoEsq(no), tl, m, l, r, novoValor);
22    update(filhoDir(no), m + 1, tr, l, r, novoValor);
23

```

```

24     // Após atualizar o filho (esquerdo ou direito), precisamos atualizar o
25     valor do nó.
    st[no] = funcao(st[filhoEsq(no)], st[filhoDir(no)]);
}

```

1. Parâmetros:

- `no` : nó atual
- `tl` : limite inferior do intervalo que `no` representa (inclusivo)
- `tr` : limite superior do intervalo que `no` representa (inclusivo)
- `l` : limite inferior do intervalo que se deseja atualizar no vetor
- `r` : limite superior do intervalo que se deseja atualizar no vetor
- `novoValor` : novo valor dos elementos no intervalo

Versão final da classe SegTree com alguns exemplos de utilização:

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 typedef long long ll;
5 typedef vector<ll> vll;
6
7 #define filhoEsq(i) (2*(i) + 1)
8 #define filhoDir(i) (2*(i) + 2)
9
10 class SegTree {
11 private:
12     vll st, lazy;
13     vector<bool> marked;
14     ll size;
15
16     ll funcao(ll a, ll b) { // Função usada na SegTree: soma
17         return a + b;
18     }
19
20     void propagate(ll p, ll L, ll R) {
21         if (marked[p]) {
22             // O valor do nó será: número de elementos que esse intervalo
23             // representa vezes o novo valor de cada elemento do intervalo
24             st[p] = lazy[p] * (R - L + 1);
25             if (L != R) {
26                 lazy[filhoDir(p)] = lazy[filhoEsq(p)] = lazy[p];
27                 marked[filhoDir(p)] = marked[filhoEsq(p)] = true;
28             }
29             marked[p] = false;
30         }
31     }
32
33     ll query(ll no, ll tl, ll tr, ll l, ll r){
34         //Precisamos propagar a possível atualização do nó
35         propagate(no, tl, tr);
36
37         //O nó está fora do intervalo de interesse. Retorne o elemento

```

```

38     neutro que não afeta a consulta.
39     if(tr < l || r < tl)
40         return 0;
41
42         // O nó está completamente incluído no intervalo de interesse.
43         Retorne a informação contida no nó.
44         if(tl >= l and tr <= r)
45             return st[no];
46
47         // Se chegarmos aqui, é porque esse nó está parcialmente contido no
48         intervalo de interesse. Então, continuamos procurando nos filhos.
49         ll m = tl + (tr - tl) / 2;
50         return funcao(query(filhoEsq(no), tl, m, l, r), query(filhoDir(no),
51 m + 1, tr, l, r));
52     }
53
54     void update(ll no, ll tl, ll tr, ll l, ll r, ll novoValor){
55         //Precisamos propagar a possível atualização do nó
56         propagate(no, tl, tr);
57
58         // Chegamos no índice que queremos atualizar o valor
59         if (tl >= l and tr <= r) {
60             lazy[no] = novoValor;
61             marked[no] = true;
62             propagate(no, tl, tr);
63             return;
64         }
65
66         // O intervalo que estamos não contém o índice que queremos
67         atualizar, retorno
68         if (tl > r or tr < l)
69             return;
70
71         // O intervalo contém o índice, mas temos que chegar no nó
72         específico. Repetimos o processo recursivamente nos filhos.
73         ll m = tl + (tr - tl) / 2;
74         update(filhoEsq(no), tl, m, l, r, novoValor);
75         update(filhoDir(no), m + 1, tr, l, r, novoValor);
76
77         // Após atualizar o filho (esquerdo ou direito), precisamos
78         atualizar o valor do nó.
79         st[no] = funcao(st[filhoEsq(no)], st[filhoDir(no)]);
80     }
81
82     void build(ll no, ll L, ll R, const vll &A) {
83         //Chegamos no nó que deve conter o valor de A[L] ou A[R]
84         if (L == R)
85             st[no] = A[L];
86         else {
87             //Recursivamente construimos a SegTree
88             ll m = L + (R - L) / 2;
89             build(filhoEsq(no), L, m, A);
90             build(filhoDir(no), m + 1, R, A);
91             st[no] = funcao(st[filhoEsq(no)], st[filhoDir(no)]);
92         }
93     }
94
95     public:
96         SegTree(ll n) : st(4 * n, 0), lazy(4 * n, 0), marked(4 * n, false),

```

```

97     size(n) {}
98     SegTree(const vll &a) {
99         size = a.size();
100        st.resize(4 * size, 0);
101        lazy.resize(4 * size, 0);
102        marked.resize(4 * size, false);
103        build(0, 0, size - 1, a);
104    }
105    ll query(ll l, ll r) {
106        return query(0, 0, size - 1, l, r);
107    }
108    void update(ll i, ll x){
109        update(0, 0, size - 1, i, i, x);
110    }
111    void update(ll l, ll r, ll x){
112        update(0, 0, size - 1, l, r, x);
113    }
114};
115
116 int main() {
117     SegTree st(10);
118     st.update(0, 4, 1);
119     cout << st.query(0, 4) << "\n"; // 5
120     cout << st.query(5, 9) << "\n"; // 0
121     cout << st.query(0, 9) << "\n"; // 5
122     cout << st.query(0, 0) << "\n"; // 1
123     cout << st.query(5, 5) << "\n"; // 0
124
125     st.update(5, 9, 2);
126     cout << st.query(0, 4) << "\n"; // 5
127     cout << st.query(5, 9) << "\n"; // 10
128     cout << st.query(0, 9) << "\n"; // 15
129     cout << st.query(0, 0) << "\n"; // 1
130     cout << st.query(5, 5) << "\n"; // 2
131
132     st.update(0, 9, 1);
133     cout << st.query(0, 4) << "\n"; // 5
134     cout << st.query(5, 9) << "\n"; // 5
135     cout << st.query(0, 9) << "\n"; // 10
136     cout << st.query(0, 0) << "\n"; // 1
137     cout << st.query(5, 5) << "\n"; // 1
138
139     st.update(5, 20);
140     cout << st.query(0, 4) << "\n"; // 5
141     cout << st.query(5, 9) << "\n"; // 24
142     cout << st.query(0, 9) << "\n"; // 29
143     cout << st.query(0, 0) << "\n"; // 1
144     cout << st.query(5, 5) << "\n"; // 20
145
146     return 0;
147 }
```

Material complementar

- ★ Segment Tree, part 1 (ITMO Academy) 🐻
- ★ Segment Tree, part 2 (ITMO Academy) 🐻

- [⭐ Segment Tree](#)
- [⭐ Segment Tree \(Árvore de Segmentos\)](#)
- [⭐ Segment Tree com Lazy Propagation](#)
- [Segment Tree \(CS Academy\)](#)
- [Segment Trees](#)
- [Segment Tree - VisuAlgo](#)
- [Segment Tree Data Structure - Min Max Queries](#)

Disjoint Set Union (DSU)

Para entender DSU, considere o seguinte problema (extraído desse [link](#)): suponha que exista um grupo de N amigos em um jogo. Inicialmente, cada um deles joga contra todos os outros.

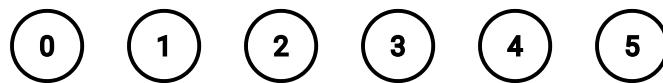
Conforme o jogo avança, alianças são formadas entre eles. A relação de aliança é transitiva, o que significa que se A e B são aliados e B e C são aliados, então A e C também são aliados. Você sabe quando as alianças são formadas. Em certos momentos você precisa saber se dois amigos em particular estão no mesmo time ou não.

Uma estratégia natural seria construir um grafo e, a cada atualização, incluir uma nova aresta ligando dois vértices do grafo. Para cada consulta pode-se realizar um percurso no grafo (usando [Busca em Largura](#) ou [Busca em Profundidade](#)) começando no nó A e verificando se o nó B é visitado. Adicionar uma aresta é muito rápido, mas as consultas são lentas. Uma estratégia mais eficiente para resolver esse problema é usar **Disjoint Set Union**.

Um **conjunto disjunto**, também chamado **union-find**, é uma estrutura de dados que opera com um conjunto particionado em vários subconjuntos disjuntos. As duas principais operações dessa estrutura são:

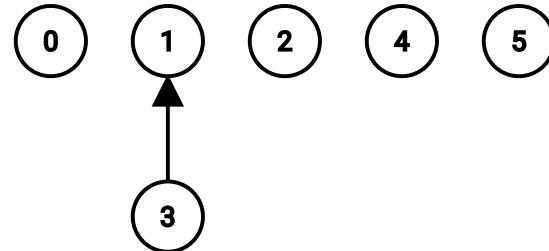
- **find** : Dado um elemento particular do conjunto, a função identifica o subconjunto do elemento. Para isso, a função retorna o representante do conjunto.
- **join** : Une dois subconjuntos em um único subconjunto.

Um conjunto disjunto funciona representando cada componente conectado como uma árvore, onde a raiz de cada árvore é o representante do componente. O pai de cada nó é outro nó no mesmo componente. Considere o exemplo dos amigos jogando para ver como a estrutura funciona. Suponha que existam 6 amigos jogando. Inicialmente, não há alianças, então cada nó é a raiz de uma árvore (os amigos são numerados de 0 a 5):



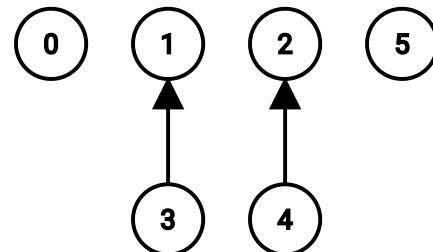
Fonte: [Disjoint-set Data Structures](#)

Em seguida, suponha que 1 e 3 se tornem aliados. Considere que o nó 1 seja escolhido como representante:



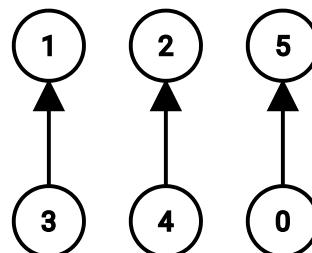
Fonte: Disjoint-set Data Structures

Então 2 e 4 se tornam aliados:



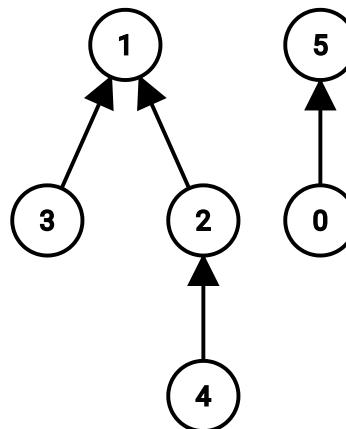
Fonte: Disjoint-set Data Structures

Os amigos 0 e 5 são os próximos. Considere que 5 seja escolhido como representante:



Fonte: Disjoint-set Data Structures

Finalmente, uma aliança é formada entre 4 e 3. Nesse caso, é preciso pegar a raiz de uma árvore e anexá-la como filho da raiz da outra árvore:



Fonte: *Disjoint-set Data Structures*

Implementação

A estrutura DSU pode ser implementada usando arrays. Na implementação a seguir, o array `pai` contém para cada elemento o próximo elemento do encadeamento ou o próprio elemento se ele for um representante. Inicialmente, cada elemento pertence a um conjunto distinto:

```

1 vll pai;
2
3 void init(ll N) {
4     pai.resize(N + 1);
5     iota(pai.begin(), pai.end(), 0); // (1)
6 }
```

1. std:iota

A função `find` retorna o representante do elemento `x`. O representante pode ser encontrado seguindo o encadeamento que começa em `x`. O código abaixo ilustra essa operação, note que a complexidade da função é $O(n)$:

```

1 ll find(ll x) {
2     if(x == pai[x])
3         return x;
4     else
5         return find(pai[x]);
6 }
```

```

1 void join(ll a, ll b) {
2     pai[find(a)] = find(b);
3 }
```

Otimizações do Union-Find

A primeira otimização do algoritmo, conhecida como *Path Compression*, está na busca do representante de um elemento, ou seja, na função `find`. Observe que se a função `find` for eficiente, a função `join` também se torna eficiente. Note que o que gasta tempo na função são as chamadas recursivas da função que precisam passar por todos os ancestrais de um determinado elemento. Porém, pode-se usar um princípio da Programação Dinâmica: evitar o recálculo! Uma vez que calculado o representante de um elemento `x` (ou seja, `find(x)`), pode-se salvá-lo diretamente como seu pai (`pai[x]=find(x);`). Assim, nas próximas vezes que for calculado o valor de `find(x)`, a função retornará seu representante rapidamente, pois ele já estará salvo em `pai[x]`, o que evita a necessidade de percorrer todos os ancestrais que estavam entre `x` e o representante. Para fazer essa otimização basta que, na hora de o valor de `find(x)` for retornado, o mesmo seja salvo em `pai[x]`. Segue a implementação da função `find` otimizada:

```

1  ll find(ll x) {
2      if(x == pai[x])
3          return x;
4      else
5          return pai[x] = find(pai[x]);
6  }
```

Outra otimização, conhecida como *Union by size*, altera a forma como é feita a união dos conjuntos. Na primeira implementação da função `join`, o segundo conjunto sempre é anexado ao primeiro, o que pode levar a árvores degeneradas. Nessa otimização, o menor conjunto (menor número de elementos) é unido ao maior conjunto. Segue a implementação da função `join` otimizada:

```

1  vll pai;
2  vll tamanho;
3
4  void init(ll N) {
5      pai.resize(N + 1);
6      tamanho.resize(N + 1, 1);
7      iota(pai.begin(), pai.end(), 0);
8  }
9
10 void join(ll a, ll b) {
11     a = find(a);
12     b = find(b);
13     if (a != b) {
14         if (tamanho[a] < tamanho[b])
15             swap(a, b);
16         pai[b] = a;
17         tamanho[a] += tamanho[b];
18     }
19 }
```

Material complementar

- ★ Disjoint Sets Union 🎊

- [!\[\]\(f9c2f0279f1ed67440afe88c128db2d8_img.jpg\) Disjoint Set Union](#)
- [!\[\]\(53488aa40711d079aa3cc3de2f45edd9_img.jpg\) Union-Find \(Neps Academy\)](#)
- [Disjoint-set Data Structures \(CSAcademy\)](#)
- [Disjoint Set Data Structures \(GeeksforGeeks\)](#)

Busca Exaustiva (Recursão + Backtracking)¹

Uma **busca exaustiva** ou **busca completa** tem por objetivo gerar todas as soluções possíveis para um determinado problema usando **força bruta** e então selecionar a melhor solução ou contar o número de soluções, dependendo do problema.

A busca exaustiva é uma boa técnica se houver tempo suficiente para gerar todas as soluções, pois geralmente é fácil de implementar e sempre fornece a resposta correta. Entretanto, se o número de soluções for muito grande ou a geração das soluções for demorada, outras técnicas, como **algoritmos gulosos** ou **programação dinâmica**, podem ser necessárias.

Gerando subconjuntos

Nesse problema, deseja-se gerar todos os subconjuntos de n elementos ou de um conjunto específico. Por exemplo, os subconjuntos de $\{0, 1, 2\}$ são: $\emptyset, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}$ e $\{0, 1, 2\}$. Existem dois métodos comuns para gerar subconjuntos: realizar uma busca recursiva ou explorar a representação de bits de inteiros.

Método 1

Uma maneira elegante de gerar todos os subconjuntos de um conjunto é usar recursão. A função abaixo gera os subconjuntos do conjunto $\{0, 1, \dots, n - 1\}$. A função utiliza um vetor booleano para definir se o elemento i pertence ou não ao subconjunto.

```

1  typedef long long ll;
2  vector<bool> subconj;
3
4  void imprime(ll n) {
5      cout << "{ ";
6      for(ll i = 0; i < n; i++)
7          if(subconj[i]) cout << i << " ";
8      cout << "}\n";
9  }
10
11 void geraSubconjuntos(ll n, ll i = 0) {
12     if (i == n) {
13         imprime(n);
14         return;
15     }
16
17     subconj[i] = false; // Elemento i não está no subconjunto
18     geraSubconjuntos(n, i + 1);
19     subconj[i] = true; // Elemento i está no subconjunto
20     geraSubconjuntos(n, i + 1);
21 }
```

```

22
23 int main() {
24     ll n;
25     cin >> n;
26     subconj.resize(n + 1, false);
27     geraSubconjuntos(n);
28     return 0;
29 }
```

Método 2

Outra maneira de gerar subconjuntos é baseada na representação de bits de inteiros. Cada subconjunto de um conjunto de n elementos pode ser representado como uma sequência de n bits, que corresponde a um inteiro entre $0 \dots 2^n - 1$. Os bits 1 da sequência de bits indicam quais elementos estão incluídos no subconjunto. A convenção usual é que o último bit corresponde ao elemento 0, o penúltimo bit corresponde ao elemento 1 e assim por diante. Por exemplo, a representação de bit de 25 é 11001, que corresponde ao subconjunto $\{0, 3, 4\}$

O código abaixo gera os subconjuntos de um conjunto de n elementos baseado nesse método.

```

1 void imprime(vector<ll> subconj) {
2     cout << "{ ";
3     for(auto i: subconj)
4         cout << i << " ";
5     cout << "}\n";
6 }
7
8 void geraSubconjuntos(ll n) {
9     for (ll b = 0; b < (1<<n); b++) {
10         vector<ll> subconj;
11         for (ll i = 0; i < n; i++) {
12             if (b&(1<<i)) subconj.push_back(i);
13         }
14         imprime(subconj);
15     }
16 }
17
18 int main() {
19     ll n;
20     cin >> n;
21     geraSubconjuntos(n);
22     return 0;
23 }
```

Exemplos

- Gerar todos os subconjuntos de um conjunto de valores lidos

```

1  typedef long long ll;
2
3  vector<bool> subconj;
4  vector<ll> valores;
5
6  void imprime(ll n) {
7      cout << "{ ";
8      for(ll i = 0; i < n; i++)
9          if(subconj[i]) cout << valores[i] << " ";
10     cout << "}\n";
11 }
12
13 void geraSubconjuntos(ll n, ll i = 0) {
14     if (i == n) {
15         imprime(n);
16         return;
17     }
18
19     subconj[i] = false; // Elemento i não está no subconjunto
20     geraSubconjuntos(n, i + 1);
21     subconj[i] = true; // Elemento i está no subconjunto
22     geraSubconjuntos(n, i + 1);
23 }
24
25 int main() {
26     ll n;
27     cin >> n;
28     subconj.resize(n + 1, false);
29     valores.resize(n);
30     for(ll i = 0; i < n; i++)
31         cin >> valores[i];
32     geraSubconjuntos(n);
33     return 0;
34 }
```

- Gerar todos os subconjuntos de um conjunto de valores lidos cuja soma seja menor ou igual a um limite

```

1  typedef long long    ll;
2
3  vector<bool> subconj;
4  vector<ll> valores;
5  ll limite;
6
7  void imprime(ll n) {
8      cout << "{ ";
9      for(ll i = 0; i < n; i++)
10         if(subconj[i]) cout << valores[i] << " ";
11     cout << "}\n";
12 }
13
14 ll soma(ll n) {
15     ll s = 0;
16     for (ll i = 0; i < n; i++)
```

```

17     if (subconj[i])
18         s += valores[i];
19     return s;
20 }
21
22 void geraSubconjuntos(ll n, ll i = 0) {
23     if (i == n) {
24         if (soma(n) <= limite)
25             imprime(n);
26         return;
27     }
28
29     subconj[i] = true; // Elemento i está no subconjunto
30     geraSubconjuntos(n, i + 1);
31     subconj[i] = false; // Elemento i não está no subconjunto
32     geraSubconjuntos(n, i + 1);
33 }
34
35 int main() {
36     ll n;
37     cin >> n;
38     cin >> limite;
39     subconj.resize(n + 1, false);
40     valores.resize(n);
41     for(ll i = 0; i < n; i++)
42         cin >> valores[i];
43
44     geraSubconjuntos(n);
45
46     return 0;
47 }
```

- Mesmo que o anterior, mas "corta" o backtracking assim que chega em um subconjunto cuja soma seja maior que o limite. Assim, não espera terminar de gerar o subconjunto para testar a soma, por isso é bem mais rápido.

```

1  typedef long long ll;
2
3  vector<bool> subconj;
4  vector<ll> valores;
5  ll limite;
6
7  void imprime(ll n) {
8      cout << "{ ";
9      for(ll i = 0; i < n; i++)
10         if(subconj[i]) cout << valores[i] << " ";
11     cout << "}\n";
12 }
13
14 ll soma(ll n) {
15     ll s = 0;
16     for (ll i = 0; i < n; i++)
17         if (subconj[i])
18             s += valores[i];
19     return s;
```

```

20 }
21
22 void geraSubconjuntos(ll n, ll i = 0) {
23     if (soma(n) > limite)
24         return;
25
26     if (i == n) {
27         imprime(n);
28         return;
29     }
30
31     subconj[i] = true; // Elemento i está no subconjunto
32     geraSubconjuntos(n, i + 1);
33     subconj[i] = false; // Elemento i não está no subconjunto
34     geraSubconjuntos(n, i + 1);
35 }
36
37 int main() {
38
39     ll n;
40     cin >> n;
41     cin >> limite;
42     subconj.resize(n + 1, false);
43     valores.resize(n);
44     for(ll i = 0; i < n; i++)
45         cin >> valores[i];
46
47     geraSubconjuntos(n);
48
49     return 0;
50 }
```

- Mesmo que o anterior, mas não recalcula a soma dos elementos (chamada da função `soma` que é $O(n)$).

```

1 typedef long long ll;
2
3 vector<bool> subconj;
4 vector<ll> valores;
5 ll limite;
6
7 void imprime(ll n) {
8     cout << "{ ";
9     for(ll i = 0; i < n; i++)
10        if(subconj[i]) cout << valores[i] << " ";
11     cout << "}\n";
12 }
13
14 void geraSubconjuntos(ll n, ll i = 0, ll soma = 0 ) {
15     if (soma > limite)
16         return;
17
18     if (i == n) {
19         imprime(n);
20         return;
21     }
```

```

22     subconj[i] = true; // Elemento i está no subconjunto
23     geraSubconjuntos(n, i + 1, soma + valores[i]);
24     subconj[i] = false; // Elemento i não está no subconjunto
25     geraSubconjuntos(n, i + 1, soma);
26   }
27 }
28
29 int main() {
30
31   ll n;
32   cin >> n;
33   cin >> limite;
34   subconj.resize(n + 1, false);
35   valores.resize(n);
36   for(ll i = 0; i < n; i++)
37     cin >> valores[i];
38
39   geraSubconjuntos(n);
40
41   return 0;
42 }
```

Gerando permutações

Considere agora o problema de gerar todas as permutações de um conjunto de n elementos. Por exemplo, as permutações de $\{0, 1, 2\}$ são $(0, 1, 2)$, $(0, 2, 1)$, $(1, 0, 2)$, $(1, 2, 0)$, $(2, 0, 1)$ e $(2, 1, 0)$. Novamente, existem duas abordagens: usar a recursão ou percorrer as permutações iterativamente.

Método 1

Assim como no problema de gerar os subconjuntos, as permutações também podem ser geradas usando recursão.

```

1  typedef long long ll;
2
3  vector<ll> permutacao;
4  vector<bool> pertence;
5
6  void imprime(ll n) {
7    cout << "(";
8    for(ll i = 0; i < n; i++)
9      cout << permutacao[i] << " ";
10   cout << ")"\n";
11 }
12
13 void geraPermutacao(ll n) {
14   if (permutacao.size() == n) {
15     imprime(n);
16     return;
17   }
18 }
```

```

19     for (ll i = 0; i < n; i++) {
20         if(pertence[i])
21             continue;
22         pertence[i] = true;      // Inclui o elemento i na permutação
23         permutacao.push_back(i);
24         geraPermutacao(n);    // Gera o restante da permutação
25
26         pertence[i] = false;    // Remove o elemento i da permutação
27         permutacao.pop_back();
28     }
29 }
30
31 int main() {
32
33     ll n;
34     cin >> n;
35     pertence.resize(n, false);
36     geraPermutacao(n);
37
38     return 0;
39 }
```

Método 2

Outra alternativa é usar a função da biblioteca padrão do C++ `next_permutation`, que a cada chamada constrói a próxima permutação em ordem crescente.

```

1  typedef long long ll;
2
3  vector<ll> permutacao;
4
5  void imprime() {
6      cout << "(";
7      for(auto x: permutacao)
8          cout << x << " ";
9      cout << ")"\n";
10 }
11
12 int main() {
13
14     ll n;
15     cin >> n;
16     permutacao.resize(n, false);
17     iota(permutacao.begin(), permutacao.end(), 0); // ①
18     do {
19         //Processa a permutação
20         imprime();
21     } while (next_permutation(permutacao.begin(),permutacao.end()));
22
23     return 0;
24 }
```

1. `std:iota`

Exemplos

- Gera todas as permutações de a partir de valores lidos

```

1  typedef long long ll;
2
3  vector<ll> permutacao, valores;
4  vector<bool> pertence;
5
6  void imprime(ll n) {
7      cout << "(";
8      for(ll i = 0; i < n; i++)
9          cout << valores[permutacao[i]] << " ";
10     cout << ")"\n";
11 }
12
13 void geraPermutacao(ll n) {
14     if (permutacao.size() == n) {
15         imprime(n);
16         return;
17     }
18
19     for (ll i = 0; i < n; i++) {
20         if(pertence[i])
21             continue;
22         pertence[i] = true; // Inclui o elemento i na permutação
23         permutacao.push_back(i);
24         geraPermutacao(n); // Gera o restante da permutação
25
26         pertence[i] = false; // Remove o elemento i da permutação
27         permutacao.pop_back();
28     }
29 }
30
31 int main() {
32
33     ll n;
34     cin >> n;
35     pertence.resize(n, false);
36     valores.resize(n);
37     for (ll i = 0; i < n; i++)
38         cin >> valores[i];
39     geraPermutacao(n);
40
41     return 0;
42 }
```

- Gera todas as permutações de $0, 1, 2, 3, \dots, n - 1$ em que 2 e 3 não aparecem seguidos.

```

1  typedef long long ll;
2
3  vector<ll> permutacao;
4  vector<bool> pertence;
5
```

```

6 void imprime(ll n) {
7     cout << "(" ;
8     for(ll i = 0; i < n; i++)
9         cout << permutacao[i] << " ";
10    cout << ")"\n";
11 }
12
13 bool seguidos() {
14     for (ll i = 0; i < permutacao.size() - 1; i++)
15         if (permutacao[i] == 2 && permutacao[i + 1] == 3)
16             return true;
17     return false;
18 }
19
20 void geraPermutacao(ll n) {
21     if (permutacao.size() == n) {
22         if (!seguidos())
23             imprime(n);
24         return;
25     }
26
27     for (ll i = 0; i < n; i++) {
28         if(pertence[i])
29             continue;
30         pertence[i] = true;      // Inclui o elemento i na permutação
31         permutacao.push_back(i);
32         geraPermutacao(n);      // Gera o restante da permutação
33
34         pertence[i] = false;     // Remove o elemento i da permutação
35         permutacao.pop_back();
36     }
37 }
38
39 int main() {
40
41     ll n;
42     cin >> n;
43     pertence.resize(n, false);
44
45     geraPermutacao(n);
46
47     return 0;
48 }
```

- Mesmo que o anterior, mas corta o backtracking tão logo 2 e 3 apareçam seguidos

```

1 typedef long long ll;
2
3 vector<ll> permutacao;
4 vector<bool> pertence;
5
6 void imprime(ll n) {
7     cout << "(" ;
8     for(ll i = 0; i < n; i++)
9         cout << permutacao[i] << " ";
10    cout << ")"\n";
```

```

11 }
12
13 bool seguidos() {
14     for (ll i = 0; i < permutacao.size() - 1; i++)
15         if (permutacao[i] == 2 && permutacao[i + 1] == 3)
16             return true;
17     return false;
18 }
19
20 void geraPermutacao(ll n) {
21     if (seguidos())
22         return;
23
24     if (permutacao.size() == n) {
25         imprime(n);
26         return;
27     }
28
29     for (ll i = 0; i < n; i++) {
30         if(pertence[i])
31             continue;
32         pertence[i] = true;      // Inclui o elemento i na permutação
33         permutacao.push_back(i);
34         geraPermutacao(n);    // Gera o restante da permutação
35
36         pertence[i] = false;    // Remove o elemento i da permutação
37         permutacao.pop_back();
38     }
39 }
40
41 int main() {
42
43     ll n;
44     cin >> n;
45     pertence.resize(n, false);
46
47     geraPermutacao(n);
48
49     return 0;
50 }
```

Backtracking

Backtracking (também chamado de **tentativa e erro** ou **força bruta**) é uma técnica que usa recursividade para gerar/enumerar/percorrer sistematicamente todas as alternativas no espaço de soluções. O método constrói soluções candidatas incrementalmente e abandona (*backtrack*) uma solução parcial quando identifica que tal solução parcial não levará a uma solução do problema. Como dito, é um método de tentativa e erro: tenta um caminho, se não der certo, volta e tenta outro. Assim, a ideia por trás de um algoritmo backtracking é que ele busca uma solução para um problema entre **todas** as opções disponíveis.

Existem três tipos de problemas que são comumente resolvidos com Backtracking:

1. Problema de Decisão – Neste, busca-se uma solução viável.
2. Problema de Otimização – Neste, deseja-se a melhor solução.
3. Problema de Enumeração – Neste, busca-se encontrar todas as soluções viáveis.

Algoritmo Geral

O código abaixo ilustra a ideia geral de um algoritmo backtracking. Note que essa ideia foi usada nos algoritmos para gerar [subconjuntos e permutações](#)

```

1 // int a[] - solução parcial, de a[0] até a[k-1]
2 // int k - posição para inserir o próximo passo
3 // int n - tamanho máximo da solução
4 void backtrack(int a[], int k, int n) {
5     // É importante cortar a busca tão logo se descubra que não levará a
6     // uma solução
7     if ( /* não há como continuar */ )      // Corte
8         return;
9
10    if ( /* terminou */ ) {                  // se chegou numa possível
11        solução
12        /* processa solução */
13        return;
14    }
15    for ( /* toda alternativa */ )          // para toda alternativa
16        if ( /* alternativa viável */ ) { // se pode incluí-la
17            a[k] = /* alternativa */;      // inclui na solução parcial
18            backtrack(a, k+1, n);        // gera o restante
19        }
20    }
}

```

Exemplos

- Procura a saída de um labirinto, sendo `.` espaço livre e `#` as paredes

```

1 /*
2 Exemplo de entrada:
3 7 8
4 #####
5 #..#...
6 ##...#..
7 #.#.##.#
8 #....#.#
9 ##.###.##
10 #####
11 5 3
12
13 Saída:
14
15 Saída encontrada:
16 #####
17 #..#ooo#
18 ##.oo#os

```

```

19  #.#o##.#  

20  #.oo.#.#  

21  ##.#.##  

22  #####  

23  */  

24  

25  #include <bits/stdc++.h>  

26  

27  using namespace std;  

28  

29  typedef long long ll;  

30  

31  ll nl, nc, li, ci;  

32  vector<vector<char>> mapa;  

33  bool achouSaida = false;  

34  

35  void imprime() {  

36      for (ll l = 0; l < nl; l++, cout << "\n")  

37          for (ll c = 0; c < nc; c++)  

38              cout << mapa[l][c];  

39  }  

40  

41  void caminha(ll l, ll c) {  

42      // Se achou solução, não precisa continuar  

43      if (achouSaida)  

44          return;  

45      // Verifica se 'l' e 'c' são valores válidos  

46      if (l < 0 || c < 0 || l >= nl || c >= nc)  

47          return;  

48      // Verifica se a posição não é parede  

49      if (mapa[l][c] != '.')  

50          return;  

51      // Verifica se chegou em uma das extremidades do mapa.  

52      // Se chegou, achou a saída.  

53      if (l == 0 || c == 0 || l == nl - 1 || c == nc - 1) {  

54          mapa[l][c] = 's';  

55          achouSaida = true;  

56          return;  

57      }  

58      // Se não retornou ainda, é porque o processo deve continuar  

59      mapa[l][c] = 'o'; // Marca o mapa, indicando que foi passado nessa  

60      posição  

61      caminha(l, c - 1); // Tenta caminhar para trás  

62      caminha(l + 1, c); // Tenta caminhar para baixo  

63      caminha(l, c + 1); // Tenta caminhar para frente  

64      caminha(l - 1, c); // Tenta caminhar para cima  

65  

66      // Se não achou saída, desmarca no mapa  

67      if (!achouSaida)  

68          mapa[l][c] = '.';  

69  }  

70  

71  int main() {  

72  

73      cin >> nl >> nc;  

74      mapa.resize(nl, vector<char>(nc));  

75      for (ll l = 0; l < nl; l++)

```

```

76         for(ll c = 0; c < nc; c++)
77             cin >> mapa[1][c];
78         cin >> li >> ci;
79         caminha(li - 1, ci - 1);
80
81     if (achouSaida) {
82         cout << "Saída encontrada:" << endl;
83         imprime();
84     }
85     else
86         cout << "Solução não encontrada" << endl;
87
88     return 0;
}

```

- Conta o número de regiões. Semelhante ao do labirinto, mas caminha nos `#` marcando toda uma região cercada por "água", simbolizada por `.`

```

1  /*
2  Exemplo de entrada:
3  7 8
4  ..#...#.
5  ..#####.
6  #...#...
7  ##.....
8  ###...#.
9  .....###
10 ..#...#.
11
12
13 Saída:
14
15 Número de regiões: 4
16 ..1...1.
17 ..11111.
18 2...1...
19 22.....
20 222...3.
21 .....333
22 ..4...3.
23 */
24 #include <bits/stdc++.h>
25
26 using namespace std;
27
28 typedef long long ll;
29
30 ll nl, nc;
31 vector<vector<string>> mapa;
32
33 void imprime() {
34     for (ll l = 0; l < nl; l++, cout << "\n")
35         for (ll c = 0; c < nc; c++)
36             cout << mapa[l][c];
37 }
38

```

```

39 void caminha(ll l, ll c, ll nr) {
40     // Verifica se 'l' e 'c' são valores válidos
41     if (l < 0 || c < 0 || l >= nl || c >= nc)
42         return;
43     // Se a posição não for uma região, retorne
44     if (mapa[l][c] != "#")
45         return;
46
47     // Se não retornou ainda, é porque o processo deve continuar
48     mapa[l][c] = to_string(nr); // Marca no mapa o número da região
49     caminha(l, c - 1, nr); // Tenta caminhar para trás
50     caminha(l + 1, c, nr); // Tenta caminhar para baixo
51     caminha(l, c + 1, nr); // Tenta caminhar para frente
52     caminha(l - 1, c, nr); // Tenta caminhar para cima
53
54 }
55
56 int main() {
57
58     cin >> nl >> nc;
59     mapa.resize(nl, vector<string>(nc));
60     char v;
61     for (ll l = 0; l < nl; l++) {
62         for (ll c = 0; c < nc; c++) {
63             cin >> v; // ①
64             mapa[l][c] = v;
65         }
66
67         ll numRegioes = 0;
68         // Procura pelas regiões ainda não marcadas
69         for (ll l = 0; l < nl; l++) {
70             for (ll c = 0; c < nc; c++) {
71                 if (mapa[l][c] == "#") { // Encontrou uma região
72                     numRegioes++;
73                     caminha(l, c, numRegioes); // Marca a região
74                 }
75             }
76
77             cout << "Número de regiões: " << numRegioes << endl;
78             imprime();
79
80         return 0;
81     }

```

1. Se for feito `cin >> mapa[l][c];`, a linha toda será atribuída a posição `mapa[l][c]`.

Material complementar

- [Backtracking Algorithms \(GeeksforGeeks\)](#)

1. Os textos dessa página são traduções e adaptações encontrados nesse link: [1 ↪](#)

C++ Template

```
#include <bits/stdc++.h>
#include <bits/extc++.h>
using namespace __gnu_pbds;
using namespace std;

typedef long long ll;
typedef unsigned long long llu;
typedef pair<int,int> pii;
typedef pair<ll,ll> pll;
typedef vector<int> vi;
typedef vector<ll> vll;
typedef vector<pii> vpi;
typedef vector<pll> vpll;
#define F first
#define S second
#define PB push_back
#define MP make_pair
#define FOR(i,a,b) for(ll i = (a); i < (ll)(b); ++i)
#define INF 0x3f3f3f3f3f3f3f3f
#define INFLL 0x3f3f3f3f3f3f3f3f
#define all(x) x.begin(),x.end()
#define sz(x) (ll)x.size()
#define MOD 1000000007ll
#define endl '\n'
#define mdc(a, b) (__gcd((a), (b)))
#define mmc(a, b) (((a)/__gcd(a, b)) * b)
#define W(x) cerr << "\033[31m" << #x << " = " << x << "\033[0m" << endl;
#define FASTIO ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);

int main() {
    FASTIO;

    return 0;
}
```