#include <bits/stdc++.h>

[1 Cálculo numérico 1](#_Toc461197885)

[1.1 Zero de funções 1](#_Toc461197886)

[1.2 Integração 1](#_Toc461197887)

[1.3 Derivação 1](#_Toc461197888)

[2 Funções de recorrência 2](#_Toc461197889)

[2.1 Números eulerianos 2](#_Toc461197890)

[2.2 Triângulo de Bell 2](#_Toc461197891)

[2.3 Triângulo de Catalão 2](#_Toc461197892)

[3 Algoritmos 4](#_Toc461197893)

[3.1 Algoritmo de Euclides Estendido 4](#_Toc461197894)

[3.2 Máximo divisor comum 4](#_Toc461197895)

[3.3 Algoritmo de Pollard Rho 4](#_Toc461197896)

[3.4 Mínimo múltiplo comum 5](#_Toc461197897)

[3.5 Algoritmo Húngaro 5](#_Toc461197898)

[3.6 Fatoração em números primos 7](#_Toc461197899)

[3.7 Matching máximo em grafo bipartido 8](#_Toc461197900)

[3.8 ModPow 9](#_Toc461197901)

[3.9 Máximo e mínimo de funções 9](#_Toc461197902)

[3.10 Mergesort 10](#_Toc461197903)

[3.11 Suffix Tree 11](#_Toc461197904)

[4 Bibliotecas e estruturas 13](#_Toc461197905)

[4.1 Geometria computacional reduzido 13](#_Toc461197906)

[4.2 Geometria computacional completo 15](#_Toc461197907)

[4.3 Grafos 18](#_Toc461197908)

[4.4 Segtree com lazy propagation 22](#_Toc461197909)

[4.5 Matrizes 24](#_Toc461197910)

[4.6 Union-Find com ranking 25](#_Toc461197911)

[5 Bibliotecas STD 26](#_Toc461197912)

[5.1 Vector 26](#_Toc461197913)

[5.2 Map 26](#_Toc461197914)

[5.3 Queue / Priority\_queue 26](#_Toc461197915)

# Cálculo numérico

## Zero de funções

## Integração

## Derivação

# Funções de recorrência

## Números eulerianos

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 |  |  |  |  |  |  |
| 1 | 1 |  |  |  |  |  |
| 1 | 4 | 1 |  |  |  |  |
| 1 | 11 | 11 | 1 |  |  |  |
| 1 | 26 | 66 | 26 | 1 |  |  |
| 1 | 57 | 302 | 302 | 57 | 1 |  |
| 1 | 120 | 1191 | 2416 | 1191 | 120 | 1 |

Função de recorrência:

## Triângulo de Bell

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 |  |  |  |  |  |  |
| 1 | 2 |  |  |  |  |  |
| 2 | 3 | 5 |  |  |  |  |
| 5 | 7 | 10 | 15 |  |  |  |
| 15 | 20 | 27 | 37 | 52 |  |  |
| 52 | 67 | 87 | 113 | 151 | 203 |  |
| 203 | 255 | 322 | 409 | 523 | 674 | 877 |

Modo de construção:

## Triângulo de Catalão

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 |  |  |  |  |  |  |
| 1 | 1 |  |  |  |  |  |
| 1 | 2 | 2 |  |  |  |  |
| 1 | 3 | 5 | 5 |  |  |  |
| 1 | 4 | 9 | 14 | 14 |  |  |
| 1 | 5 | 14 | 28 | 42 | 42 |  |
| 1 | 6 | 20 | 48 | 90 | 132 | 132 |

Modo de construção:

# Algoritmos

## Algoritmo de Euclides Estendido

|  |  |
| --- | --- |
| Objetivo: | Dados os inteiros a e b, calcular oe achar valores para x e y tais que |
| Entrada: | a, b. |
| Saída: | x, y, |
|  |  |

int **x**mdc**(**inta**,** intb**,** int&x**,** int&y**)** **{**

**if(**b **==** 0**){**

x **=** 1**;**

y **=** 0**;**

**return** a**;**

**}**

intx1**,** y1**,** mdc **=** xmdc**(**b**,** a % b**,** x1**,** y1**);**

x **=** y1**;**

y **=** x1 **-** **(**a / b**)** \* y1**;**

**return** mdc**;**

**}**

## Máximo divisor comum

intmdc**(**inta**,** intb**)** **{**

intremainder**;**

**while** **(**b **!=** 0**)** **{**

remainder **=** a % b**;**

a **=** b**;**

b **=** remainder**;**

**}**

**return** a**;**

**}**

## Algoritmo de Pollard Rho

|  |  |
| --- | --- |
| Objetivo: | Decompor um número inteiro em seus dois maiores fatores primos. |
| Entrada: | Um número inteiro. |
| Saída: | O menor dos dois fatores do número. |
|  |  |

intpollard**(**intnumber**)** **{**

x\_fixed **=** 2**,** cycle\_size **=** 2**,** x **=** 2**,** factor **=** 1**;**

**while** **(**factor **==** 1**)** **{**

**for (**intcount**=**1**;**count **<=** cycle\_size && factor **<=** 1**;**count**++)** **{**

x **=** **(**x\*x**+**1**)**%number**;**

factor **=** mdc**(**x **-** x\_fixed**,** number**);**

**}**

cycle\_size \***=** 2**;**

x\_fixed **=** x**;**

**}**

**return** factor**;**

**}**

## Mínimo múltiplo comum

int mmc**(**int a**,** int b**)** **{**

int temp **=** mdc**(**a**,** b**);**

**return** temp ? **(**a / temp \* b**)** : 0**;**

**}**

## Algoritmo Húngaro

|  |  |
| --- | --- |
| Objetivo: | Dados que temos n trabalhadores e n tarefas a serem feitos, e para cada par (trabalhador, tarefa) teremos um valor salário. O resultado será o custo de realizar todas as tarefas com um custo mínimo, onde um trabalho será realizado por apenas um trabalhador e vice-versa. |
| Entrada: | Uma matriz de custos[n][n], onde o valor armazenado em [i][j] é o salário do trabalhador i para realizar o trabalho j. |
| Saída: | Valor mínimo para a realização de todas tarefas. |

#**define** N 100

#**define** INF 100000000

intcost**[**N**][**N**],** n**,** max\_match**,** lx**[**N**],** ly**[**N**],** xy**[**N**],** yx**[**N**],** slack**[**N**],** slackx**[**N**],** prev**[**N**];**

boolS**[**N**],** T**[**N**];**

voidinit\_labels**()** **{**

memset**(**lx**,** 0**,** sizeof**(**lx**));**

memset**(**ly**,** 0**,** sizeof**(**ly**));**

**for (**intx **=** 0**;** x **<** n**;** x**++)**

**for (**inty **=** 0**;** y **<** n**;** y**++)**

lx**[**x**]** **=** max**(**lx**[**x**],** cost**[**x**][**y**]);**

**}**

voidupdate\_labels**()** **{**

intx**,** y**,** delta **=** INF**;**

**for (**y **=** 0**;** y **<** n**;** y**++)**

**if (!**T**[**y**])**

delta **=** min**(**delta**,** slack**[**y**]);**

**for (**x **=** 0**;** x **<** n**;** x**++)**

**if (**S**[**x**])**

lx**[**x**]** **-=** delta**;**

**for (**y **=** 0**;** y **<** n**;** y**++)**

**if (**T**[**y**])**

ly**[**y**]** **+=** delta**;**

**for (**y **=** 0**;** y **<** n**;** y**++)**

**if (!**T**[**y**])**

slack**[**y**]** **-=** delta**;**

**}**

voidadd\_to\_tree**(**intx**,** intprevx**)** **{**

S**[**x**]** **=** true**;**

prev**[**x**]** **=** prevx**;**

**for (**inty **=** 0**;** y **<** n**;** y**++)**

**if (**lx**[**x**]** **+** ly**[**y**]** **-** cost**[**x**][**y**]** **<** slack**[**y**])** **{**

slack**[**y**]** **=** lx**[**x**]** **+** ly**[**y**]** **-** cost**[**x**][**y**];**

slackx**[**y**]** **=** x**;**

**}**

**}**

voidaugment**()** **{**

**if (**max\_match **==** n**)**

**return;**

intx**,** y**,** root**,** q**[**N**],** wr **=** 0**,** rd **=** 0**;**

memset**(**S**,** false**,** sizeof**(**S**));**

memset**(**T**,** false**,** sizeof**(**T**));**

memset**(**prev**,** **-**1**,** sizeof**(**prev**));**

**for (**x **=** 0**;** x **<** n**;** x**++)**

**if (**xy**[**x**]** **==** **-**1**)** **{**

q**[**wr**++]** **=** root **=** x**;**

prev**[**x**]** **=** **-**2**;**

S**[**x**]** **=** true**;**

break**;**

**}**

**for (**y **=** 0**;** y **<** n**;** y**++)** **{**

slack**[**y**]** **=** lx**[**root**]** **+** ly**[**y**]** **-** cost**[**root**][**y**];**

slackx**[**y**]** **=** root**;**

**}**

**while** **(**true**)** **{**

**while** **(**rd **<** wr**)** **{**

x **=** q**[**rd**++];**

**for (**y **=** 0**;** y **<** n**;** y**++)**

**if (**cost**[**x**][**y**]** **==** lx**[**x**]** **+** ly**[**y**]** && **!**T**[**y**])** **{**

**if (**yx**[**y**]** **==** **-**1**)**

break**;**

T**[**y**]** **=** true**;**

q**[**wr**++]** **=** yx**[**y**];**

add\_to\_tree**(**yx**[**y**],** x**);**

**}**

**if (**y **<** n**)**

break**;**

**}**

**if (**y **<** n**)**

break**;**

update\_labels**();**

wr **=** rd **=** 0**;**

**for (**y **=** 0**;** y **<** n**;** y**++)**

**if (!**T**[**y**]** && slack**[**y**]** **==** 0**)** **{**

**if (**yx**[**y**]** **==** **-**1**)** **{**

x **=** slackx**[**y**];**

break**;**

**}**

**else** **{**

T**[**y**]** **=** true**;**

**if (!**S**[**yx**[**y**]])** **{**

q**[**wr**++]** **=** yx**[**y**];**

add\_to\_tree**(**yx**[**y**],** slackx**[**y**]);**

**}**

**}**

**}**

**if (**y **<** n**)**

break**;**

**}**

**if (**y **<** n**)** **{**

max\_match**++;**

**for (**intcx **=** x**,** cy **=** y**,** ty**;** cx **!=** **-**2**;** cx **=** prev**[**cx**],** cy **=** ty**)** **{**

ty **=** xy**[**cx**];**

yx**[**cy**]** **=** cx**;**

xy**[**cx**]** **=** cy**;**

**}**

augment**();**

**}**

**}**

inthungaro**()** **{**

intret **=** 0**;**

max\_match **=** 0**;**

memset**(**xy**,** **-**1**,** sizeof**(**xy**));**

memset**(**yx**,** **-**1**,** sizeof**(**yx**));**

init\_labels**();**

augment**();**

**for (**intx **=** 0**;** x **<** n**;** x**++)**

ret **+=** cost**[**x**][**xy**[**x**]];**

**return** ret**;**

**}**

## Fatoração em números primos

|  |  |
| --- | --- |
| Objetivo: | Dado um número n, obter um vetor com todos seus fatores primos. Por exemplo: entrada: 4, saída: [2, 2]. |
| Entrada: | Um número n > 0. |
| Saída: | Um vector contendo todos seus fatores primos, incluindo números repetidos. |

vector**<**int**>** primeFactors**(**int n**)** **{**

vector**<**int**>** v**;**

intsqrtn **=** sqrt**(**n**);**

**while (**n%2 **==** 0**)** **{**

v**.**push\_back**(**2**);**

n **=** n/2**;**

**}**

**for (**inti **=** 3**;** i **<=** sqrtn**;** i **=** i**+**2**)** **{**

**while (**n % i **==** 0**)** **{**

v**.**push\_back**(**i**);**

n **=** n/i**;**

**}**

**}**

**if (**n **>** 2**)**

v**.**push\_back**(**n**);**

**return** v**;**

**}**

## Matching máximo em grafo bipartido

|  |  |
| --- | --- |
| Objetivo: | Obter o matching máximo em um grafo bipartido em **.** |
| Entrada: | Um grafo com n1 vértices no seu lado direito e n2 vértices no seu lado esquerdo, com todas as arestas já adicionadas. |
| Saída: | O valor do fluxo máximo. |
| Uso: | |  |  | | --- | --- | | Inicializar: | init(n\_Esquerda, n\_Direita); | | Adicionar aresta: | addAresta(origem, destino); | |
|  |  |

**const** int MAXN1 **=** 50000**,** MAXN2 **=** 50000**,** MAXM **=** 150000**;**

int n1**,** n2**,** edges**,** last**[**MAXN1**],** prev**[**MAXM**],** head**[**MAXM**],** matching**[**MAXN2**],** dist**[**MAXN1**],** Q**[**MAXN1**],** used**[**MAXN1**],** vis**[**MAXN1**];**

void init**(**int \_n1**,** int \_n2**)** **{**

n1 **=** \_n1**;**

n2 **=** \_n2**;**

edges **=** 0**;**

fill**(**last**,** last **+** n1**,** **-**1**);**

**}**

void addAresta**(**int u**,** int v**)** **{**

head**[**edges**]** **=** v**;**

prev**[**edges**]** **=** last**[**u**];**

last**[**u**]** **=** edges**++;**

**}**

void bfs**()** **{**

fill**(**dist**,** dist **+** n1**,** **-**1**);**

int sizeQ **=** 0**;**

**for** **(**int u **=** 0**;** u **<** n1**;** **++**u**)** **{**

**if** **(!**used**[**u**])** **{**

Q**[**sizeQ**++]** **=** u**;**

dist**[**u**]** **=** 0**;**

**}**

**}**

**for** **(**int i **=** 0**;** i **<** sizeQ**;** i**++)** **{**

int u1 **=** Q**[**i**];**

**for** **(**int e **=** last**[**u1**];** e **>=** 0**;** e **=** prev**[**e**])** **{**

int u2 **=** matching**[**head**[**e**]];**

**if** **(**u2 **>=** 0 && dist**[**u2**]** **<** 0**)** **{**

dist**[**u2**]** **=** dist**[**u1**]** **+** 1**;**

Q**[**sizeQ**++]** **=** u2**;**

**}**

**}**

**}**

**}**

bool dfs**(**int u1**)** **{**

vis**[**u1**]** **=** **true;**

**for** **(**int e **=** last**[**u1**];** e **>=** 0**;** e **=** prev**[**e**])** **{**

int v **=** head**[**e**];**

int u2 **=** matching**[**v**];**

**if** **(**u2 **<** 0 || **!**vis**[**u2**]** && dist**[**u2**]** **==** dist**[**u1**]** **+** 1 && dfs**(**u2**))** **{**

matching**[**v**]** **=** u1**;**

used**[**u1**]** **=** **true;**

**return** **true;**

**}**

**}**

**return** **false;**

**}**

int maxMatching**()** **{**

fill**(**used**,** used **+** n1**,** **false);**

fill**(**matching**,** matching **+** n2**,** **-**1**);**

**for** **(**int res **=** 0**;;)** **{**

bfs**();**

fill**(**vis**,** vis **+** n1**,** **false);**

int f **=** 0**;**

**for** **(**int u **=** 0**;** u **<** n1**;** **++**u**)**

**if** **(!**used**[**u**]** && dfs**(**u**))**

**++**f**;**

**if** **(!**f**)**

**return** res**;**

res **+=** f**;**

**}**

**}**

## ModPow

|  |  |
| --- | --- |
| Objetivo: | Obter o resultado de |
| Entrada: |  |
| Saída: |  |
|  |  |

**int** modPow(**int** a, **int** b, **int** m) {  
 **int** res = **1**;  
 **for** (; b > **0**; b >>= **1**) {  
 **if** (b & **1**)  
 res = (**long long**) res \* a % m;  
 a = (**long long**) a \* a % m;  
 }  
 **return** res;  
}

## Máximo e mínimo de funções

|  |  |
| --- | --- |
| Objetivo: | Dada uma função f, que recebe um valor double, achar seu ponto de minímo ou máximo |
| Entrada: | a, b (mínimo e máximo do domínio); f(x); erro máximo. |
| Saída: | ou |
|  |  |

double gss**(**double a**,** double b**,** double **(**\*f**)(**double**),** double e **=** 1e**-**6**)** **{**

double r **=** **(**sqrt**(**5**)-**1**)**/2**;** //**=.**618**...=**golden ratio**-**1

double x1 **=** b**-**r\***(**b**-**a**),** x2 **=** a**+**r\***(**b**-**a**);**

double f1 **=** f**(**x1**),** f2 **=** f**(**x2**);**

**while** **(**b**-**a **>** e**)** **{**

**if** **(**f1 **<** f2**)** **{** //change to **>** to find maximum

b **=** x2**;** x2 **=** x1**;** f2 **=** f1**;**

x1 **=** b**-**r\***(**b**-**a**);** f1 **=** f**(**x1**);**

**}** **else** **{**

a **=** x1**;** x1 **=** x2**;** f1 **=** f2**;**

x2 **=** a**+**r\***(**b**-**a**);** f2 **=** f**(**x2**);**

**}**

**}**

**return** **(**b **+** a**)** / 2**;**

**}**

## Mergesort

**typedef** vector**<**int**>**::iterator vec\_it**;**

void merge**(**vec\_it left**,** vec\_it left\_end**,** vec\_it right**,** vec\_it right\_end**,** vec\_it numbers**)** **{**

**while** **(**left **!=** left\_end**)** **{**

**if** **(**\*left **<** \*right || right **==** right\_end**)** **{**

\*numbers **=** \*left**;**

**++**left**;**

**}** **else** **{**

\*numbers **=** \*right**;**

**++**right**;**

**}**

**++**numbers**;**

**}**

**while** **(**right **!=** right\_end**)** **{**

\*numbers **=** \*right**;**

**++**right**;**

**++**numbers**;**

**}**

**}**

void merge\_sort**(**vector**<**int**>**& numbers**)** **{**

**if** **(**numbers**.**size**()** **<=** 1**)**

**return;**

vector**<**int**>**::size\_type middle **=** numbers**.**size**()** / 2**;**

vector**<**int**>** left**(**numbers**.**begin**(),** numbers**.**begin**()** **+** middle**);**

vector**<**int**>** right**(**numbers**.**begin**()** **+** middle**,** numbers**.**end**());**

merge\_sort**(**left**);**

merge\_sort**(**right**);**

merge**(**left**.**begin**(),** left**.**end**(),** right**.**begin**(),** right**.**end**(),** numbers**.**begin**());**

**}**

## Todos divisores de um número

vector**<**int**>** divisores**(**int n**)** **{**

vector**<**int**>** div**;**

sqrtn **=** sqrt**(**n**)** **+** 1**;**

**for(**i **=** 1**;** i **<=** sqrtn**;** i**++)**

**if(!(**n % i**))**

div**.**push\_back**(**i**),** div**.**push\_back**(**n / i**);**

**return** div**;**

**}**

## Suffix Tree

//Usage:

// Fill txt with the characters of the txting**.**

// Call SuffixSort**(**n**),** where n is the length of the txting stored in txt**.**

// That's it**!**

//Output:

// SA **=** The suffix array**.**

// Contains the n suffixes of txt sorted in lexicographical order**.**

// Each suffix is represented as a single integer **(**the SAition of txt where it starts**).**

// iSA **=** The inverse of the suffix array**.** iSA**[**i**]** **=** the index of the suffix txt**[**i**..**n**)**

// in the SA array**.** **(**In other words**,** SA**[**i**]** **=** k **<==>** iSA**[**k**]** **=** i**)**

// With this array**,** you can compare two suffixes in O**(**1**)**: Suffix txt**[**i**..**n**)** is smaller

// than txt**[**j**..**n**)** **if** and only **if** iSA**[**i**]** **<** iSA**[**j**]**

**const** int MAX **=** 100010**;**

char txt**[**MAX**];** //input

int iSA**[**MAX**],** SA**[**MAX**];** //output

int cnt**[**MAX**],** next**[**MAX**];** //internal

bool bh**[**MAX**],** b2h**[**MAX**];**

// Compares two suffixes according to their first characters

bool smaller\_first\_char**(**int a**,** int b**){**

**return** txt**[**a**]** **<** txt**[**b**];**

**}**

void suffixSort**(**int n**){**

**for** **(**int i**=**0**;** i**<**n**;** **++**i**)**

SA**[**i**]** **=** i**;**

sort**(**SA**,** SA **+** n**,** smaller\_first\_char**);**

**for** **(**int i**=**0**;** i**<**n**;** **++**i**)** **{**

bh**[**i**]** **=** i **==** 0 || txt**[**SA**[**i**]]** **!=** txt**[**SA**[**i**-**1**]];**

b2h**[**i**]** **=** **false;**

**}**

**for** **(**int h **=** 1**;** h **<** n**;** h **<<=** 1**){**

int buckets **=** 0**;**

**for** **(**int i**=**0**,** j**;** i **<** n**;** i **=** j**){**

j **=** i **+** 1**;**

**while** **(**j **<** n && **!**bh**[**j**])** j**++;**

next**[**i**]** **=** j**;**

buckets**++;**

**}**

**if** **(**buckets **==** n**)** **break;**

**for** **(**int i **=** 0**;** i **<** n**;** i **=** next**[**i**]){**

cnt**[**i**]** **=** 0**;**

**for** **(**int j **=** i**;** j **<** next**[**i**];** **++**j**)**

iSA**[**SA**[**j**]]** **=** i**;**

**}**

cnt**[**iSA**[**n **-** h**]]++;**

b2h**[**iSA**[**n **-** h**]]** **=** **true;**

**for** **(**int i **=** 0**;** i **<** n**;** i **=** next**[**i**])** **{**

**for** **(**int j **=** i**;** j **<** next**[**i**];** **++**j**)** **{**

int s **=** SA**[**j**]** **-** h**;**

**if** **(**s **>=** 0**){**

int head **=** iSA**[**s**];**

iSA**[**s**]** **=** head **+** cnt**[**head**]++;**

b2h**[**iSA**[**s**]]** **=** **true;**

**}**

**}**

**for** **(**int j **=** i**;** j **<** next**[**i**];** **++**j**){**

int s **=** SA**[**j**]** **-** h**;**

**if** **(**s **>=** 0 && b2h**[**iSA**[**s**]])**

**for** **(**int k **=** iSA**[**s**]+**1**;** **!**bh**[**k**]** && b2h**[**k**];** k**++)**

b2h**[**k**]** **=** **false;**

**}**

**}**

**for** **(**int i**=**0**;** i**<**n**;** **++**i**){**

SA**[**iSA**[**i**]]** **=** i**;**

bh**[**i**]** |**=** b2h**[**i**];**

**}**

**}**

**for** **(**int i**=**0**;** i**<**n**;** **++**i**){**

iSA**[**SA**[**i**]]** **=** i**;**

**}**

**}**

// End of suffix array algorithm

// Begin of the O**(**n**)** longest common prefix algorithm

int lcp**[**MAX**];**

// lcp**[**i**]** **=** length of the longest common prefix of suffix SA**[**i**]** and suffix SA**[**i**-**1**]**

// lcp**[**0**]** **=** 0

void getlcp**(**int n**)** **{**

**for** **(**int i**=**0**;** i**<**n**;** **++**i**)**

iSA**[**SA**[**i**]]** **=** i**;**

lcp**[**0**]** **=** 0**;**

**for** **(**int i**=**0**,** h**=**0**;** i**<**n**;** **++**i**)** **{**

**if** **(**iSA**[**i**]** **>** 0**)** **{**

int j **=** SA**[**iSA**[**i**]-**1**];**

**while** **(**i **+** h **<** n && j **+** h **<** n && txt**[**i**+**h**]** **==** txt**[**j**+**h**])**

h**++;**

lcp**[**iSA**[**i**]]** **=** h**;**

**if** **(**h **>** 0**)**

h**--;**

**}**

**}**

**}**

## Crivo de Eratóstenes segmentado (para gerar primos muito grandes)

char nprimo**[**100001**]** **=** **{**0**};** // tamanho **=** sqrt**(**maximo**)**

std::vector**<**int**>** primos**;**

int main**()** **{**

int n**,** a**,** b**,** i**,** j**;**

nprimo**[**1**]** **=** 1**;**

nprimo**[**0**]** **=** 1**;**

**for(**i **=** 2**;** i **<** 320**;** i**++)** // i **[**2**,** sqrt**(**sqrt**(**maximo**))]**

**if(!**nprimo**[**i**])**

**for(**j **=** i \* i**;** j **<** 100001**;** j **+=** i**)** // j **[**i^2**,** sqrt**(**maximo**)]**

nprimo**[**j**]** **=** 1**;**

**for(**i **=** 2**;** i **<** 100001**;** i**++)**

**if(!**nprimo**[**i**])**

primos**.**push\_back**(**i**);**

scanf**(**"%d"**,** &n**);**

**while(**n**--)** **{**

scanf**(**"%d %d"**,** &a**,** &b**);**

**if(**a **>** 100000 && b **>** 100000**)** **{** // **(**a **>** sqrt**(**N**)** && b **>** sqrt**(**N**))**

**for(**i **=** a**;** i **<=** b**;** i**++)** **{**

**for(**j **=** 0**;** j **<** primos**.**size**();** j**++)**

**if(**i % primos**[**j**]** **==** 0**)**

**goto** ab**;**

printf**(**"%d\n"**,** i**);**

ab:**;**

**}**

**}**

**else** **if(**a **<** 100001 && b **<** 100001**)** **{** // (a < sqrt(N) && b < sqrt(N))

for(i = a; i <= b; i++)

if(!nprimo[i])

printf("%d\n"**,** i**);**

**}**

**else** **{**

**for(**i **=** 0**;** i **<** primos**.**size**();** i**++)**

**if(**primos**[**i**]** **>=** a**)**

break**;**

**for(;** i **<** primos**.**size**();** i**++)**

printf**(**"%d\n"**,** primos**[**i**]);**

**for(;** i **<=** b**;** i**++)** **{**

**for(**j **=** 0**;** j **<** primos**.**size**();** j**++)**

**if(**i % primos**[**j**]** **==** 0**)**

**goto** ac**;**

printf**(**"%d\n"**,** i**);**

ac:**;**

**}**

**}**

**}**

**}**

# Bibliotecas e estruturas

## Geometria computacional reduzido

**typedef** pair**<**double**,** double**>** Ponto**;**

bool cw**(**Ponto a**,** Ponto b**,** Ponto c**)** **{**

**return** **(**b**.**first **-** a**.**first**)** \* **(**c**.**second **-** a**.**second**)** **-** **(**b**.**second **-** a**.**second**)** \* **(**c**.**first **-** a**.**first**)** **<** 0**;**

**}**

// Retorna o casco convexo **do** conjunto de pontos p

vector**<**Ponto**>** convexHull**(**vector**<**Ponto**>** p**)** **{**

int n **=** p**.**size**();**

**if** **(**n **<=** 1**)**

**return** p**;**

int k **=** 0**;**

sort**(**p**.**begin**(),** p**.**end**());**

vector**<**Ponto**>** q**(**n \* 2**);**

**for** **(**int i **=** 0**;** i **<** n**;** q**[**k**++]** **=** p**[**i**++])**

**for** **(;** k **>=** 2 && **!**cw**(**q**[**k **-** 2**],** q**[**k **-** 1**],** p**[**i**]);** **--**k**)**

**;**

**for** **(**int i **=** n **-** 2**,** t **=** k**;** i **>=** 0**;** q**[**k**++]** **=** p**[**i**--])**

**for** **(;** k **>** t && **!**cw**(**q**[**k **-** 2**],** q**[**k **-** 1**],** p**[**i**]);** **--**k**)**

**;**

q**.**resize**(**k **-** 1 **-** **(**q**[**0**]** **==** q**[**1**]));**

**return** q**;**

**}**

//O dobro da área definida pelo triangulo de pontos pontos a**,** b e c **(**sem sinal**).**

double uArea2**(**Ponto a**,** Ponto b**,** Ponto c**)** **{**

**return** abs**((**b**.**first **-** a**.**first**)** \* **(**c**.**second **-** a**.**second**)** **-** **(**b**.**second **-** a**.**second**)** \* **(**c**.**first **-** a**.**first**));**

**}**

//O dobro da área definida pelo triangulo de pontos pontos a**,** b e c **(**com sinal**).**

double area2**(**Ponto a**,** Ponto b**,** Ponto c**)** **{**

**return** **(**b**.**first **-** a**.**first**)** \* **(**c**.**second **-** a**.**second**)** **-** **(**b**.**second **-** a**.**second**)** \* **(**c**.**first **-** a**.**first**);**

**}**

//Distância entre os pontos a e b

double dist**(**Ponto a**,** Ponto b**)** **{**

**return** hypot**(**a**.**first **-** b**.**first**,** a**.**second **-** b**.**second**);**

**}**

//Interseção de semi**-**retas **(**p1 **->** p2**),** **(**p3 **->** p4**)**

bool segIntercept**(**Ponto p1**,** Ponto p2**,** Ponto p3**,** Ponto p4**)** **{**

**return** cw**(**p1**,** p2**,** p3**)** **!=** cw**(**p1**,** p2**,** p4**)** && cw**(**p3**,** p4**,** p1**)** **!=** cw**(**p3**,** p4**,** p2**);**

**}**

//Retorna a área **do** polígono p

double polygonArea**(**vector**<**Ponto**>** p**)** **{**

double s **=** 0**.**0**;**

**for** **(**int i **=** 0**;** i **<** p**.**size**();** i**++** **)**

s **+=** area2**(**Ponto**(**0**,**0**),** p**[**i**],** p**[(** i **+** 1 **)** % p**.**size**()]);**

**return** fabs**(**s / 2**.**0**);**

**}**

//Retorna a área **do** polígono p definido pelos pontos p**[**i**,** f**]**

double polygonArea2**(**vector**<**Ponto**>** p**,** int i**,** int f**)** **{**

double s **=** 0**.**0**;**

Ponto primeiro **=** p**[**i**];**

**for** **(;** i **!=** f**;** i**++)**

s **+=** area2**(**Ponto**(**0**,**0**),** p**[**i**],** p**[(** i **+** 1 **)]);**

s **+=** area2**(**Ponto**(**0**,**0**),** p**[**i**],** primeiro**);**

**return** fabs**(**s / 2**.**0**);**

**}**

//Retorna a menor largura **do** conjunto de pontos p

double raio**(**vector**<**Ponto**>** p**)** **{**

vector**<**Ponto**>** h **=** convexHull**(**p**);**

int m **=** h**.**size**();**

**if** **(**m **==** 1**)**

**return** 0**;**

**if** **(**m **==** 2**)**

**return** 0**;**

int k **=** 1**;**

**while** **(**uArea2**(**h**[**m **-** 1**],** h**[**0**],** h**[(**k **+** 1**)** % m**])** **>** uArea2**(**h**[**m **-** 1**],** h**[**0**],** h**[**k**]))**

**++**k**;**

double res **=** 10000000**;**

**for** **(**int i **=** 0**,** j **=** k**;** i **<=** k && j **<** m**;** i**++)** **{**

res **=** min**(**res**,** dist**(**h**[**i**],** h**[**j**]));**

**while** **(**j **<** m && uArea2**(**h**[**i**],** h**[(**i **+** 1**)** % m**],** h**[(**j **+** 1**)** % m**])** **>** uArea2**(**h**[**i**],** h**[(**i **+** 1**)** % m**],** h**[**j**]))** **{**

res **=** min**(**res**,** dist**(**h**[**i**],** h**[(**j **+** 1**)** % m**]));**

**++**j**;**

**}**

**}**

**return** res**;**

**}**

//Retorna a maior largura **do** conjunto de pontos p

double diametro**(**vector**<**Ponto**>** p**)** **{**

vector**<**Ponto**>** h **=** convexHull**(**p**);**

int m **=** h**.**size**();**

**if** **(**m **==** 1**)**

**return** 0**;**

**if** **(**m **==** 2**)**

**return**dist**(**h**[**0**],** h**[**1**]);**

int k **=** 1**;**

**while** **(**uArea2**(**h**[**m **-** 1**],** h**[**0**],** h**[(**k **+** 1**)** % m**])** **>** uArea2**(**h**[**m **-** 1**],** h**[**0**],** h**[**k**]))**

**++**k**;**

double res **=** 0**;**

**for** **(**int i **=** 0**,** j **=** k**;** i **<=** k && j **<** m**;** i**++)** **{**

res **=** max**(**res**,** dist**(**h**[**i**],** h**[**j**]));**

**while** **(**j **<** m && uArea2**(**h**[**i**],** h**[(**i **+** 1**)** % m**],** h**[(**j **+** 1**)** % m**])** **>** uArea2**(**h**[**i**],** h**[(**i **+** 1**)** % m**],** h**[**j**]))** **{**

res **=** max**(**res**,** dist**(**h**[**i**],** h**[(**j **+** 1**)** % m**]));**

**++**j**;**

**}**

**}**

**return** res**;**

**}**

## Geometria computacional completo

**const** double EPS **=** 1e**-**10**;**

**inline** int cmp**(** double x**,** double y **=** 0**,** double tol **=** EPS **)** **{**

**return** **(**x **<=** y **+** tol **)** ? **(** x **+** tol **<** y **)** ? **-**1 : 0 : 1**;**

**}**

**struct** Point **{**

double x**,** y**;**

Point**(**double x **=** 0**,** double y **=** 0**)** : x**(**x**),** y**(**y**)** **{}**

Point operator**+(**Point q**)** **const** **{**

**return** Point**(** x **+** q**.**x**,** y **+** q**.**y **);**

**}**

Point operator**-(**Point q**)** **const** **{**

**return** Point**(** x **-** q**.**x**,** y **-** q**.**y **);**

**}**

Point operator\***(**double t**)** **const** **{**

**return** Point**(** x \* t**,** y \* t **);**

**}**

Point operator/**(**double t**)** **const** **{**

**return** Point**(** x / t**,** y / t **);**

**}**

double operator\***(**Point q**)const** **{**

**return** x \* q**.**x **+** y \* q**.**y**;**

**}**

double operator^**(** Point q **)** **const** **{**

**return** x \* q**.**y **-** y \* q**.**x**;**

**}**

int cmp**(** Point q **)** **const** **{**

**if** **(**int t **=** ::cmp**(**x**,** q**.**x**))**

**return** t**;**

**return** ::cmp**(**y**,** q**.**y**);**

**}**

bool operator**==(**Point q**)** **const** **{**

**return** cmp**(**q**)** **==** 0**;**

**}**

bool operator**!=(**Point q**)** **const** **{**

**return** cmp**(**q**)** **!=** 0**;**

**}**

bool operator**<(**Point q**)** **const** **{**

**return** cmp**(**q**)** **<** 0**;**

**}**

static Point pivot**;**

**};**

Point Point::pivot**;**

typedef vector**<**Point**>** Polygon**;**

**inline** double abs**(** Point& p **)** **{**

**return** hypot**(** p**.**x**,** p**.**y **);**

**}**

**inline** double arg**(** Point& p **)** **{**

**return** atan2**(** p**.**y**,** p**.**x **);**

**}**

//Verifica o sinal **do** produto vetorial entre os vetores **(**p**-**r**)** e **(**q **-** r**)**

**inline** int ccw**(** Point& p**,** Point& q**,** Point& r **)** **{**

**return** cmp**(** **(** p **-** r **)** ^ **(** q **-** r **)** **);**

**}**

//calcula o angulo orientado entre os vetores **(**p**-**q**)** e **(**r **-** q**)**

**inline** double angle**(** Point& p**,** Point &q**,** Point& r **)** **{**

Point u **=** p **-** q**,** w **=** r **-** q**;**

**return** atan2**(** u ^ w**,** u \* w **);**

**}**

//Decide se o ponto p esta sobre a reta que passa por p1p2**.**

bool pontoSobreReta**(** Point& p1**,** Point &p**,** Point& p2 **)** **{**

**return** ccw**(** p1**,** p2**,** p **)** **==** 0**;**

**}**

//Decide de p esta sobre o segmento p1p2

bool between**(** Point& p1**,** Point &p**,** Point& p2 **)** **{**

**return** ccw**(** p1**,** p2**,** p **)** **==** 0 && cmp**(** **(** p1 **-** p **)** \* **(** p2 **-** p **)** **)** **<=** 0**;**

**}**

//Calcula a distancia **do** ponto p a reta que passa por p1p2

double retaDistance**(** Point& p1**,** Point& p2**,** Point &p **)** **{**

Point A **=** p1 **-** p**,** B **=** p2 **-** p1**;**

**return** fabs**(** A ^ B **)** / sqrt**(** B \* B **);**

**}**

//Calcula a distancia **do** ponto p ao segmento de reta que passa por p1p2

double segDistance**(** Point& p1**,** Point& p2**,** Point &p **)** **{**

Point A **=** p1 **-** p**,** B **=** p1 **-** p2**,** C **=** p2 **-** p**;**

double a **=** A \* A**,** b **=** B \* B**,** c **=** C \* C**;**

**if** **(** cmp**(** a**,** b **+** c **)** **>=** 0 **)** **return** sqrt**(** c **);**

**if** **(** cmp**(** c**,** a **+** b **)** **>=** 0 **)return** sqrt**(** a **);**

**return** fabs**(** A ^ C **)** / sqrt**(** b **);**

**}**

//Calcula a area orientada **do** poligono T**.**

double polygonArea**(** Polygon& T **)** **{**

double s **=** 0**.**0**;**

int n **=** T**.**size**(** **);**

**for** **(** int i **=** 0**;** i **<** n**;** i**++** **)**

**{**

s **+=** T**[**i**]** ^ T**[(** i **+** 1 **)** % n**];**

**}**

**return** s / 2**.**0**;** //Retorna a area com sinal

**}**

//Classifica o ponto p em relacao ao poligono T dependendo se ele está

//na fronteira **(-**1**)** no exterior **(**0**)** ou no interior **(**1**).**

int inpoly**(** Point& p**,** Polygon& T **)** **{**

//**-**1 sobre**,** 0 fora**,** 1 dentro

double a **=** 0**.**0**;**

int n **=** T**.**size**(** **);**

**for** **(** int i **=** 0**;** i **<** n**;** i**++** **)**

**{**

**if** **(** between**(** T**[**i**],** p**,** T**[(** i **+** 1 **)** % n**]** **)** **)** **return** **-**1**;**

a **+=** angle**(** T**[**i**],** p**,** T**[(** i **+** 1 **)** % n**]** **);**

**}**

**return** cmp**(** a **)** **!=** 0**;**

**}**

//Ordenacao radial**.**

bool radialSort**(** Point p**,** Point q **)** **{**

Point P **=** p **-** Point::pivot**,** Q **=** q **-** Point::pivot**;**

double R **=** P ^ Q**;**

**if** **(** cmp**(** R **)** **)** **return** R **>** 0**;**

**return** cmp**(** P \* P**,** Q \* Q **)** **<** 0**;**

**}**

//Determina o convex hull de T**.** ATENCAO**.** A lista de pontos T e destruida**.**

Polygon convexHull**(** vector**<**Point**>**& T **)** **{**

int j **=** 0**,** k**,** n **=** T**.**size**(** **);**

Polygon U**(** n **);**

Point::pivot **=** \*min\_element**(** T**.**begin**(** **),** T**.**end**(** **)** **);**

sort**(** T**.**begin**(** **),** T**.**end**(** **),** radialSort **);**

**for** **(** k **=** n **-** 2**;** k **>=** 0 && ccw**(** T**[**0**],** T**[**n **-** 1**],** T**[**k**]** **)** **==** 0**;** k**--** **);**

reverse**(** **(** k **+** 1 **)** **+** T**.**begin**(** **),** T**.**end**(** **)** **);**

**for** **(** int i **=** 0**;** i **<** n**;** i**++** **)** **{**

// troque o **>=** por **>** para manter pontos colineares

**while** **(**j **>** 1 && ccw**(** U**[**j **-** 1**],** U**[**j **-** 2**],** T**[**i**]** **)** **>=** 0**)**

j**--;**

U**[**j**++]** **=** T**[**i**];**

**}**

U**.**resize**(** j **);**

**return** U**;**

**}**

//Interseção de semi**-**retas **(**p1 **->** p2**),** **(**p3 **->** p4**)**

bool segIntercept**(**Point p1**,** Point p2**,** Point p3**,** Point p4**)** **{**

**return** ccw**(**p1**,** p2**,** p3**)** **!=** ccw**(**p1**,** p2**,** p4**)** && ccw**(**p3**,** p4**,** p1**)** **!=** ccw**(**p3**,** p4**,** p2**);**

**}**

## Grafos

**struct** tVertice **{**

int id**,** pai**;**

long long int dist**;**

tVertice**(**int a**,** long long int b**,** int p **=** **-**1**)** **{**

id **=** a**;**

dist **=** b**;**

pai **=** p**;**

**}**

bool operator**<(**tVertice a**)** **const** **{**

**return** a**.**dist **<** dist**;**

**}**

**};**

**struct** tGrafo **{**

vector**<**tVertice**>** \*Grafo**;**

int n**,** \*pais**;**

tGrafo**(**int a**)** **{**

Grafo **=** new vector**<**tVertice**>[**a**];**

pais **=** new int**[**a**];**

n **=** a**;**

**}**

~tGrafo**()** **{**

limpa**();**

delete **[]** pais**;**

delete **[]** Grafo**;**

**}**

void operator**=(**tGrafo **const** &a**)** **{**

limpa**();**

**for(**int i **=** 0**;** i **<** n**;** i**++)**

**for(**int j **=** 0**;** j **<** a**.**Grafo**[**i**].**size**();** j**++)**

Grafo**[**i**].**push\_back**(**a**.**Grafo**[**i**][**j**]);**

**}**

void limpa**()** **{**

**for(**int i **=** 0**;** i **<** n**;** i**++)**

Grafo**[**i**].**clear**();**

**}**

void addAresta**(**int a**,** int b**,** long long int v **=** 0**)** **{**

Grafo**[**a**].**push\_back**(**tVertice**(**b**,** v**));**

**}**

void removeAresta**(**int a**,** int b**)** **{**

**for(**int i **=** 0**;** i **<** Grafo**[**a**].**size**();** i**++)**

**if(**Grafo**[**a**][**i**].**id **==** b**)** **{**

Grafo**[**a**].**erase**(**Grafo**[**a**].**begin**()** **+** i**);**

**return;**

**}**

**}**

long long int Dijkstra**(**int inicio**,** int fim**)** **{**

priority\_queue**<**tVertice**>** fila**;**

bool visitados**[**n**];**

int i**;**

**for(**i **=** 0**;** i **<** n**;** i**++)**

pais**[**i**]** **=** visitados**[**i**]** **=** 0**;**

fila**.**push**(**tVertice**(**inicio**,** 0**));**

**while(**fila**.**top**().**id **!=** fim && **!**fila**.**empty**())** **{**

**if(!**visitados**[**fila**.**top**().**id**])** **{**

**for(**i **=** 0**;** i **<** Grafo**[**fila**.**top**().**id**].**size**();** i**++)**

**if(!**visitados**[**Grafo**[**fila**.**top**().**id**][**i**].**id**])**

fila**.**push**(**tVertice**(**Grafo**[**fila**.**top**().**id**][**i**].**id**,**

fila**.**top**().**dist **+** Grafo**[**fila**.**top**().**id**][**i**].**dist**,** fila**.**top**().**id**));**

visitados**[**fila**.**top**().**id**]** **=** 1**;**

pais**[**fila**.**top**().**id**]** **=** fila**.**top**().**pai**;**

**}**

fila**.**pop**();**

**}**

**if(**fila**.**empty**())**

**return** **-**1**;**

pais**[**fila**.**top**().**id**]** **=** fila**.**top**().**pai**;**

**return** fila**.**top**().**dist**;**

**}**

// Retorna a MST**,** caso queira retornar o custo**,** basta retornar a variável custo**.**

tGrafo Prim**(**int inicio**)** **{**

priority\_queue**<**tVertice**>** fila**;**

tGrafo G2**(**n**);**

bool visitados**[**n**];**

int i**,** custo **=** 0**;**

fila**.**push**(**tVertice**(**inicio**,** 0**));**

**while(!**fila**.**empty**())** **{**

**if(!**visitados**[**fila**.**top**().**id**])** **{**

**for(**i **=** 0**;** i **<** Grafo**[**fila**.**top**().**id**].**size**();** i**++)**

**if(!**visitados**[**Grafo**[**fila**.**top**().**id**][**i**].**id**])**

fila**.**push**(**tVertice**(**Grafo**[**fila**.**top**().**id**][**i**].**id**,**

Grafo**[**fila**.**top**().**id**][**i**].**dist**,**

fila**.**top**().**id**));**

custo **+=** fila**.**top**().**dist**;**

visitados**[**fila**.**top**().**id**]** **=** 1**;**

G2**.**addAresta**(**fila**.**top**().**pai**,** fila**.**top**().**id**);**

**}**

fila**.**pop**();**

**}**

**return** G2**;**

**}**

long long int buscaLargura**(**int inicio**,** int fim**)** **{**

queue**<**tVertice**>** fila**;**

bool visitados**[**n**];**

int i**;**

**for(**i **=** 0**;** i **<** n**;** i**++)**

visitados**[**i**]** **=** pais**[**i**]** **=** 0**;**

fila**.**push**(**tVertice**(**inicio**,** 0**));**

**while(**fila**.**front**().**id **!=** fim && **!**fila**.**empty**())** **{**

**if(!**visitados**[**fila**.**front**().**id**])** **{**

**for(**i **=** 0**;** i **<** Grafo**[**fila**.**front**().**id**].**size**();** i**++)**

**if(!**visitados**[**Grafo**[**fila**.**front**().**id**][**i**].**id**])**

fila**.**push**(**tVertice**(**Grafo**[**fila**.**front**().**id**][**i**].**id**,**

fila**.**front**().**dist **+** 1**,**

fila**.**front**().**id**));**

visitados**[**fila**.**front**().**id**]** **=** 1**;**

pais**[**fila**.**front**().**id**]** **=** fila**.**front**().**pai**;**

**}**

fila**.**pop**();**

**}**

**if(**fila**.**empty**())**

**return** **-**1**;**

pais**[**fila**.**front**().**id**]** **=** fila**.**front**().**pai**;**

**return** fila**.**front**().**dist**;**

**}**

long long int fordFulkerson**(**int s**,** int t**)** **{**

int u**,** v**;**

long long int max\_flow **=** 0**;**

tGrafo G2**(**n**);**

G2 **=** \*this**;**

**while** **(**G2**.**buscaLargura**(**s**,** t**)** **>=** 0**)** **{**

long long int path\_flow **=** 10000000000**;**

**for** **(**v **=** t**;** v **!=** s**;** v **=** G2**.**pais**[**v**])** **{**

u **=** G2**.**pais**[**v**];**

path\_flow **=** min**(**path\_flow**,** G2**.**valAresta**(**u**,** v**));**

**}**

**for** **(**v **=** t**;** v **!=** s**;** v **=** G2**.**pais**[**v**])** **{**

u **=** G2**.**pais**[**v**];**

G2**.**modificaAresta**(**u**,** v**,** **-**path\_flow**);**

G2**.**modificaAresta**(**v**,** u**,** path\_flow**);**

**}**

max\_flow **+=** path\_flow**;**

**}**

**return** max\_flow**;**

**}**

long long int valAresta**(**int a**,** int b**)** **{**

**for(**int i **=** 0**;** i **<** Grafo**[**a**].**size**();** i**++)**

**if(**Grafo**[**a**][**i**].**id **==** b**)**

**return** Grafo**[**a**][**i**].**dist**;**

**return** **-**1**;**

**}**

void modificaAresta**(**int a**,** int b**,** long long int v**)** **{**

**for(**int i **=** 0**;** i **<** Grafo**[**a**].**size**();** i**++)**

**if(**Grafo**[**a**][**i**].**id **==** b**)** **{**

Grafo**[**a**][**i**].**dist **+=** v**;**

**if(**Grafo**[**a**][**i**].**dist **==** 0**)**

Grafo**[**a**].**erase**(**Grafo**[**a**].**begin**()** **+** i**);**

**return;**

**}**

addAresta**(**a**,** b**,** v**);**

**}**

bool existeAresta**(**int a**,** int b**)** **{**

**for(**int i **=** 0**;** i **<** Grafo**[**a**].**size**();** i**++)**

**if(**Grafo**[**a**][**i**].**id **==** b**)**

**return** true**;**

**return** false**;**

**}**

**};**

## Segtree com lazy propagation

#**define** MAX 1000000 // O valor aqui tem que ser **>=** 4 \* tamanho **do** maior n

#**define** ELEMENTO\_NEUTRO 0

int init**[**MAX**],** tree**[**MAX**],** lazy**[**MAX**];**

void build\_tree**(**int node**,** int a**,** int b**)** **{**

**if(**a **>** b**)**

**return;**

**if(**a **==** b**)** **{**

tree**[**node**]** **=** init**[**a**];**

lazy**[**node**]** **=** ELEMENTO\_NEUTRO**;**

**return;**

**}**

build\_tree**(**node\*2**,** a**,** **(**a**+**b**)**/2**);**

build\_tree**(**node\*2**+**1**,** 1**+(**a**+**b**)**/2**,** b**);**

//Atualização **do** pai **-** verificar operação

tree**[**node**]** **=** tree**[**node\*2**]** **+** tree**[**node\*2**+**1**];**

lazy**[**node**]** **=** ELEMENTO\_NEUTRO**;**

**}**

void update\_tree**(**int node**,** int a**,** int b**,** int i**,** int j**,** int value**)** **{**

//Atualização atrasada **-** verificar operação

**if(**lazy**[**node**]** **!=** ELEMENTO\_NEUTRO**)** **{**

tree**[**node**]** **+=** lazy**[**node**];**

**if(**a **!=** b**)** **{**

lazy**[**node\*2**]** **+=** lazy**[**node**];**

lazy**[**node\*2**+**1**]** **+=** lazy**[**node**];**

**}**

lazy**[**node**]** **=** ELEMENTO\_NEUTRO**;**

**}**

**if(**a **>** b || a **>** j || b **<** i**)**

**return;**

//Atualização **do** nó **-** verificar operação

**if(**a **>=** i && b **<=** j**)** **{**

tree**[**node**]** **+=** value**;**

**if(**a **!=** b**)** **{**

lazy**[**node\*2**]** **+=** value**;**

lazy**[**node\*2**+**1**]** **+=** value**;**

**}**

**return;**

**}**

update\_tree**(**node\*2**,** a**,** **(**a**+**b**)**/2**,** i**,** j**,** value**);**

update\_tree**(**1**+**node\*2**,** 1**+(**a**+**b**)**/2**,** b**,** i**,** j**,** value**);**

//Atualização **do** pai **-** verificar operação

tree**[**node**]** **=** tree**[**node\*2**]** **+** tree**[**node\*2**+**1**];**

**}**

int query\_tree**(**int node**,** int a**,** int b**,** int i**,** int j**)** **{**

**if(**a **>** b || a **>** j || b **<** i**)** **{**

**return** ELEMENTO\_NEUTRO**;**

**}**

//Atualização atrasada **-** verificar operação

**if(**lazy**[**node**]** **!=** ELEMENTO\_NEUTRO**)** **{**

tree**[**node**]** **+=** lazy**[**node**];**

**if(**a **!=** b**)** **{**

lazy**[**node\*2**]** **+=** lazy**[**node**];**

lazy**[**node\*2**+**1**]** **+=** lazy**[**node**];**

**}**

lazy**[**node**]** **=** ELEMENTO\_NEUTRO**;**

**}**

**if(**a **>=** i && b **<=** j**)**

**return** tree**[**node**];**

int q1 **=** query\_tree**(**node\*2**,** a**,** **(**a**+**b**)**/2**,** i**,** j**);**

int q2 **=** query\_tree**(**1**+**node\*2**,** 1**+(**a**+**b**)**/2**,** b**,** i**,** j**);**

//Retorno da arvore **-** verificar operação

**return** q1 **+** q2**;**

**}**

Uso:

Assumindo que "n" é o numero de termos que o segmento tem

Inicialize "init" com os valores iniciais**,** por exemplo:

* **for(**i **=** 0**;** i **<** n**;** scanf**(**"%d"**,** val**),** init**[**i**]** **=** val**,** i**++);**

E mande construir a arvore:

* build\_tree**(**1**,** 0**,** n**-**1**);**

Para atualizar a arvore:

* update\_tree**(**1**,** 0**,** n**-**1**,** inicio**,** fim**,** val**);**

Onde inicio é a posição inicial **do** segmento desejado e fim é a posição final **do** mesmo e val é o quanto você quer alterar os valores desse seguimento

Para fazer queries:

* query\_tree**(**1**,** 0**,** n**-**1**,** inicio**,** fim**);**

Onde inicio é a posição inicial **do** segmento desejado e fim é a posição final **do** mesmo**.** O retorno terá o mesmo tipo que os dados guardados na arvore e será o resultado **do** segmento pesquisado**.**

## Matrizes

**typedef** vector**<**int**>** vi**;**

**typedef** vector**<**vi**>** vvi**;**

**const** int mod **=** 1234567891**;**

// Retorna a matriz

vvi matrixUnit**(**int n**)** **{**

vvi res**(**n**,** vi**(**n**));**

**for** **(**int i **=** 0**;** i **<** n**;** i**++)**

res**[**i**][**i**]** **=** 1**;**

**return** res**;**

**}**

// Retorna

vvi matrixAdd**(const** vvi &a**,** **const** vvi &b**)** **{**

int n **=** a**.**size**();**

int m **=** a**[**0**].**size**();**

vvi res**(**n**,** vi**(**m**));**

**for** **(**int i **=** 0**;** i **<** n**;** i**++)**

**for** **(**int j **=** 0**;** j **<** m**;** j**++)**

res**[**i**][**j**]** **=** **(**a**[**i**][**j**]** **+** b**[**i**][**j**])** % mod**;**

**return** res**;**

**}**

// Retorna

vvi matrixMul**(const** vvi &a**,** **const** vvi &b**)** **{**

int n **=** a**.**size**();**

int m **=** a**[**0**].**size**();**

int k **=** b**[**0**].**size**();**

vvi res**(**n**,** vi**(**k**));**

**for** **(**int i **=** 0**;** i **<** n**;** i**++)**

**for** **(**int j **=** 0**;** j **<** k**;** j**++)**

**for** **(**int p **=** 0**;** p **<** m**;** p**++)**

res**[**i**][**j**]** **=** **(**res**[**i**][**j**]** **+** **(**long long**)** a**[**i**][**p**]** \* b**[**p**][**j**])** % mod**;**

**return** res**;**

**}**

// Retorna a matriz

vvi matrixPow**(const** vvi **&**a**,** int p**)** **{**

**if** **(**p **==** 0**)**

**return** matrixUnit**(**a**.**size**());**

**if** **(**p & 1**)**

**return** matrixMul**(**a**,** matrixPow**(**a**,** p **-** 1**));**

**return** matrixPow**(**matrixMul**(**a**,** a**),** p / 2**);**

**}**

// Retorna

vvi matrixPowSum**(const** vvi &a**,** int p**)** **{**

int n **=** a**.**size**();**

**if** **(**p **==** 0**)**

**return** vvi**(**n**,** vi**(**n**));**

**if** **(**p % 2 **==** 0**)**

**return** matrixMul**(**matrixPowSum**(**a**,** p / 2**),** matrixAdd**(**matrixUnit**(**n**),**

matrixPow**(**a**,** p / 2**)));**

**return** matrixAdd**(**a**,** matrixMul**(**matrixPowSum**(**a**,** p **-** 1**),** a**));**

**}**

// Criar uma matriz a**[**3**][**2**]** → vvi a**(**3**,** vi**(**2**));**

## Union-Find com ranking

Obtêm o conjunto disjunto dos pontos**,** porém**,** sempre ligando a arvore menor à arvore maior**.**

// Tamanho máximo de n

**const** int maxn **=** 200000**;**

int Rank**[**maxn**],** p**[**maxn**],** n**;**

void init**(**int \_n**)** **{**

n **=** \_n**;**

fill**(**Rank**,** Rank **+** n**,** 0**);**

**for** **(**int i **=** 0**;** i **<** n**;** i**++)** p**[**i**]** **=** i**;**

**}**

int find**(**int x**)** **{**

**return** x **==** p**[**x**]** ? x : **(**p**[**x**]** **=** find**(**p**[**x**]));**

**}**

void unir**(**int a**,** int b**)** **{**

a **=** find**(**a**);**

b **=** find**(**b**);**

**if** **(**a **==** b**)** **return;**

**if** **(**Rank**[**a**]** **<** Rank**[**b**])** swap**(**a**,** b**);**

**if** **(**Rank**[**a**]** **==** Rank**[**b**])** **++**Rank**[**a**];**

p**[**b**]** **=** a**;**

**}**

# Bibliotecas STD

## Vector

Funções:

|  |  |
| --- | --- |
| begin**()** | Retorna um iterador para o inicio **do** vector |
| end**()** | Retorna um iterador para o fim **do** vector |
| size**()** | Retorna o número de elementos contidos no vector |
| push\_back**(**T**)** | Insere o termo t no vector |
| clear**()** | Remove todos os termos **do** vector |
| pop\_back**()** | Remove o último termo **do** vector |
| swap**(**vector**)** | Troca todos elementos com o vector recebido |
| erase**(**it**)** | Remove o termo apontado pelo iterador passado |
| erase**(**it b**,** it e**)** | Remove os termos entre **[**b**,** e**)** |

## Map

Funções:

|  |  |
| --- | --- |
| begin**()** | Retorna um iterador para o inicio **do** map |
| end**()** | Retorna um iterador para o fim **do** map |
| size**()** | Retorna o número de elementos contidos no map |
| insert**(**make\_par**(**k**,** v**))** | Insere o termo v no map com a chave k |
| find**(**k**)** | Procura pelo termo com a chave k e retorna um iterador |
| lower\_bound**(**k**)** | Retorna um iterador para o menor termo até k**,** se k existir**,** retorna k |
| upper\_bound**(**k**)** | Retorna um iterador para o próximo termo após k**,** mesmo se k existir |

## Queue / Priority\_queue

Funções:

|  |  |
| --- | --- |
| front**()** | Retorna um iterador para o próximo termo da fila |
| back**()** | Retorna um iterador para o último termo da fila **(**somente queue**)** |
| push**(**T**)** | Adiciona o termo T a fila |
| pop**()** | Remove o próximo termo da fila |
| size**()** | Retorna quantos elementos existem na fila |
| empty**()** | Retorna um booleano informando se a lista está vazia |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **L** | **K** | **J** | **I** | **H** | **G** | **F** | **E** | **D** | **C** | **B** | **A** | **Problema** |
|  |  |  |  |  |  |  |  |  |  |  |  | **Quem leu** |
|  |  |  |  |  |  |  |  |  |  |  |  | **Dificuldade** |
|  |  |  |  |  |  |  |  |  |  |  |  | **Escrito** |
|  |  |  |  |  |  |  |  |  |  |  |  | **Resumo** |