1st Ed. (Beta)

Carlos Mendoza Farfán
Alfredo Granda del Águila

Universidad Peruana de Ciencias Aplicadas (UPC)

[PROGRAMMING COMPETITION COMPENDIUM]

This book contains implementations of algorithms in C++ and Java, that are frequent in programming competitions.

TABLE OF CONTENTS

CONTENTS

GENERAL INFORMATION

C++ DATA TYPES

Type	Bytes	Range	
char	1	signed	-128 to 127
		unsigned	0 to 255
int	4	signed	-2,147,483,648 to 2,147,483,647
		unsigned	0 to 4,294,967,295
long long	8	signed	-9,223,372,036,854,775,807 to 9,223,372,036,854,775,806
		unsigned	0 to 18,446,744,073,709,551,616
bool	4	true or false	
double	8	+/- 1.7e +/- 308 (~15 digits)	

IDENTIFIERS FOR PRINTF AND SCANF

d	Integer	signed decimal integer
i	Integer	signed decimal integer
o	Integer	unsigned octal integer
u	Integer	unsigned decimal integer
x	Integer	unsigned hexadecimal int (with a, b, c, d, e, f)
X	Integer	unsigned hexadecimal int (with A, B, C, D, E, F)
f	Floating point	signed value of the form [-]dddd.dddd.
e	Floating point	signed value of the form [-]d.dddd or e[+/-]ddd
g	Floating point	signed value in either e or f form, based on given value and precision. Trailing zeros and the decimal point are printed if necessary.
E	Floating point	Same as e; with E for exponent.
G	Floating point	Same as g; with E for exponent if e format used
c	Character	Single character
lc	Wide char	Single wide character (UTF-8 files)
s	String pointer	Prints characters until a null-terminator is pressed or precision is reached
ls	Wide string pointer	Prints wide characters until a null-terminator is pressed or precision is reached (UTF-8 files)
n	Pointer to int	Stores (in the location pointed to by the input argument) a count of the chars written so far.
p	Pointer	Prints the input argument as a pointer; format depends on which memory model was used. It will be either XXXX:YYYY or YYYY (offset only).

VIM

INSTALLATION

On a terminal, run the command:

```
sudo apt-get install vim-gnome
```

.VIMRC

Create a file ".vimrc" at user's home directory with this content:

```
filetype indent on
set number
set ignorecase
set smartcase
set smartindent
set tabstop=4
set shiftwidth=4
set expandtab
```

A much more complete .vimrc is available at: http://shrib.com/ybLo2aGV

COMMANDS

Edit/Compile/Run C++ code:

vim file.cpp	Creates the file *file.cpp*. If it already exists, then it is opened.
g++ file.cpp -o exe	Compiles *file.cpp* and creates executable *exe*.
./exe < in.txt > out.txt	Runs the program with *in.txt* as input y stores output in *out.txt*

Vim:

Save - Close file:	
:w	Saves the file without closing it.
:wq	Saves the file, then closes it.
:q!	Closes the file (any change is discarded).
Change Vim Mode:	
i	Switches to Insert mode.
v	Switches to Text-Selection mode.
V	Switches to Text-Selection mode (selects complete line).
ESC key	Switches to Command mode.
Moving the cursor:	
gg	Moves the cursor to the beginning of file.
G	Moves the cursor to the end of file.
*<num>*gg	Moves the cursor to line *<num>*.
w	Moves the cursor to beginning of the next word.
e	Moves the cursor to the end of current word. If it is already there, it moves the cursor to the end of the next word.
b	Moves the cursor to the beginning of current word. If it is already there, it moves the cursor to the beginning of the previous word.
$	Moves the cursor to end of the current line.
zz	Centers the screen to the position of the cursor.
Edit – Copy - Paste:	
x	Erases the character the cursor is currently pointing.
r*<car>*	Replaces the character the cursor is currently pointing for *<car>*.
dd	Cuts current line.
*<num>*dd	Cuts *<num>* lines beginning with the current one.
d$	Cuts characters starting from the cursor to the end of line.

y	Copies the selected text to local buffer.
p	Pastes the text that is on local buffer.
"+y	Copies the selected text to clipboard.
"+p	Pastes the text that is on clipboard.
Others:	
ggvG$	Selects all the text on the file.
=	Gives format to the current selected text.

TEMPLATES

LIBRARY (G++)

Use the following library to include all the rest.

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

TYPEDEF

List of typedefs used in this compendium.

```
typedef pair<int,int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
```

TRICKS

SCANF

NUMBERS

Input	Instruction	n
356	`scanf("%1d", &n);`	3
356	`scanf("%2d", &n);`	35

STRINGS

Input	Instruction	V
Hola Mundo	`scanf("%s", &V);`	Hola
Hola Mundo	`scanf("%10s", &V);`	Hola
Holamundo	`scanf("%s", &V);`	Holamundo
Holamundo	`scanf("%4s", &V);`	Hola
Hola Mundo	`scanf("%[^\n]\n", &V); // Igual a // gets`	Hola Mundo
HOLAMUndo	`scanf("%[A-Z]\n", &V);`	HOLAMU
Hola Mu n do	`scanf("%[A-Za-z ]\n", &V);`	Hola Mu n do
Hola Mu4n do	`scanf("%[A-Za-z ]\n", &V);`	Hola Mu

"Pa 34" to "La 25"	Si queremos obtener el siguiente resultado:
	V1 = Pa 34
	V2 = La 25
	`scanf("\"%[^\"]", &V1);`
	`scanf("\" to ");`
	`scanf("\"%[^\"]", &V2);`
		V1=Pa 34
		V2=La 25

LeeSoloLetras9Numeros	`scanf("%[A-Za-z]%n", &V, &cant);`	V=LeeSoloLetras Cant=12
Lee Todo excepto x	`scanf("%[^x]", &V);`	V=Lee Todo e

PRINTF

NUMBERS

Instruction	Output
`printf("%d", 35);`	35
`printf("%0.4d", 35);`	0035
`printf("%6.4d", 35);`	0035
`printf("%-6.4d", 35);`	0035
`printf("%6d%d", 35, 40);`	`    3540`
`printf("%-6d%d", 35, 40);`	`35    40`
`printf("%.0d%.0d%.0d", 12, 0, 34);` `// Does not print the "0" due to %.0d`	1234

`printf("%d%% percent", 35);`	`35% percent`
`printf("%x", 10); // Hexadecimal`	`a`
`printf("%X", 10);`	`A`
`printf("%#x", 10);`	`0xa`
`printf("%#X", 10);`	`0XA`
`printf("%#6x", 10);`	`   0xa`
`printf("%#06x", 10);`	`0x000a`

STRINGS

Instruction	Output
`printf("%.4s","Hola Mundo");`	`Hola`
`printf("%.*s", 4, "Hola Mundo");` `// The * will be replaced by the first parameter`	`Hola`
`printf("%8.4s","Hola Mundo");`	`    Hola`

`for(int i = 0; i < 10; i++)`
`   printf("%.*s\n", i+1, "0123456789");`
	`0`
	`01`
	`012`
	`0123`
	`01234`
	`012345`
	`0123456`
	`01234567`
	`012345678`
	`0123456789`

UTILITIES

SWAP TWO VARIABLES

```
int a = 10, b = 15;
swap(a, b); // a = 10, b = 15
```

DETERMINE IF A NUMBER IS ODD OR EVEN

```
if(num % 2 == 0) cout << "Even";
if(num % 2 != 0) cout << "Odd";
```

FIND THE POSITION OF A CHARACTER IN A STRING

```
char s[10] = "Friendship";
int pos = (int)strchr(s, 'r') - (int)&s;
```

OBTAIN THE FRACTIONAL PART OF A DECIMAL NUMBER

```
double num = 9.15, n;
double d = modf(num, &n); // n = 9.00, d = 0.15
```

COMPARE TWO DOUBLES

```
bool equals(double a, double b, double eps)
{
  return a < b + eps && b < a + eps;
}
```

Example:

```
double a = 9.150000005, b = 9.150000001;
cout << (equals(a, b, 0.0001)? "Equal" : "Different");
```

NUMBER ROUNDING

Round Up/Down:

```
double n = 123.54, down, up;
down = floor(n); up = ceil(n); // ceil and floor return a double
printf("%.2lf\n", down); // 123.00
printf("%.2lf\n", up);   // 124.00
```

Depending on the 0.5:

```
int r = (int)(n + 0.5); // 1.1 → 1 | 1.5 → 2 | 1.7 → 2 | 1.0 → 1
```

Considering fractional part:

```
double r = floor(n * 100.0) / 100.0; // 2.778 → 2.77  | 2.775 → 2.77
```

MEMSET

```
int v[MAX], m[MAX][MAX];
memset(v, 0, sizeof v); // Initializes array in 0
memset(m, 0, sizeof m); // Initializes matrix in 0
```

TO LOWER CASE

```
char c = 'D';
char x = (c >= 'A' && c <= 'Z')? c - 'A' + 'a' : c;   // c = 'd'
```

TO UPPER CASE

```
char c = 'd';
char x = (c >= 'a' && c <= 'z')? c - 'a' + 'A' : c;   // c = 'D'
```

CHAR TO DIGIT

```
char c = '9';
int n = c - '0'; // n = 9;
```

DIGIT TO CHAR

```
int n = 9;
char c = n + '0'; // c = '9';
```

REVERSE

```
double v[6] = { 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };
reverse(v, v + 6); // v = { 1.7, 1.6, 1.5, 1.4, 1.3, 1.2 }
char s[11] = "0123456789";
reverse(s, s + strlen(s)); // s = "9876543210"
```

ROTATE

```
double v[6] = { 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };
rotate(v, v + 2, v + 6); // v = { 1.4, 1.5, 1.6, 1.7, 1.2, 1.3 }
```

ITERATE OVER A MATRIX (ALTERNATING COLUMNS FROM LEFT TO RIGHT AND RIGHT TO LEFT)

```
int M[ROW][COL];
for(int i = 1; i <= ROW; i++)
  for(int j = 1; j <= COL; j++)
  {
    int r = i - 1;
    int c = j + (COL + 1 - 2 * j) * ((i + 1) % 2) - 1;
    M[r][c] = 1;
  }
```

ITERATE OVER A MATRIX (DIAGONAL)

```
int M[ROW][COL];
for(int d = 1; d <= (COL + ROW - 1); d++)
{
  int height = 1 + max(0, d - COL), pcount = min(d, ROW – height + 1);
  for(int j = 0; j < pcount; j++)
  {
    int r = min(COL, d) – j - 1
    int c = height + j - 1;
    M[r][c] = 1;
  }
}
```

CONVERT STRING TO NUMBER

```
char s[MAX];
sprintf(s, "%d", 798);   // s = "798"
int n = 0;
sscanf("498", "%d", &n); // n = 498
```

SPLIT STRING BY TOKENS

```
void split(char *s, char *toks, vector<string> &v)
{
  v.clear();
  char *p;
  p = strtok(s, toks); // Finds first substring free of any token
  while(p != NULL)
  {
    v.push_back(p);          // Store word
    p = strtok(NULL, toks); // Finds next word
  }
}
```

s is the string to split, **tok** contains the delimiter characters, **v** will contain the splitted strings. Example:

```
vector<string> v;
split("Hola-que tal,  medio  ,como estan", " -,", v);
// v = { "Hola", "que", "tal", "medio", "como", "estan" }
```

Using STL:

```
void split(const string &s, char tok, vector<string> &v)
{
  v.clear();
  stringstream ss(s);
  string p;
  while(getline(ss, p, tok))
    v.push_back(p);
}
```

CONVERT STRING TO UPPER CASE

```
void toUpper(char *s)
{
  for(int i = 0; s[i] != 0; i++)
    s[i] = toupper(s[i]);
}

void toLower(char *s)
{
  for(int i = 0; s[i] != 0; i++)
    s[i] = tolower(s[i]);
}
```

NUMBER OF DIGITS OF INTEGER

```
int n = 457;
int d = log10(abs(n)) + 1; // d = 3
// abs is to deal with negative numbers
```

BASE CONVERSION

DECIMAL TO BINARY

```
char* toBinary(unsigned int a)
{
  unsigned int c = 1;
  char s[33]; s[32] = '\0';
  for(int i = 31; i >= 0; i--)
  {
    s[i] = (a & c)? '1' : '0';
    c <<= 1;
  }
  return s;
}
```

Example:

```
puts(toBinary(78));
```

BINARY TO DECIMAL

```
unsigned int toInteger(char* s)
{
  unsigned int a = 0;
  for(int i = 0; i < 32; i++)
  {
    a = a | (s[i] - '0');
    if(i != 31)
      a <<= 1;
  }
  return a;
}
```

Example:

```
char s[33] = "00000000000000000000000000001010";
printf("%u", toInteger(s));
```

DECIMAL TO HEX / HEX TO DECIMAL

```
char h[100];
int n = 4095;
sprintf(h,"%X",n); // to Hex
sprintf(h,"%x",n); // to Hex
sscanf(h,"%x",&n); // to Int
```

TO ANY BASE (FROM 2 TO 36)

```
char* toBase(int n, int b = 10)
{
  char s[205];
  int pos = 0, sign = n;
  n = abs(n);
  do {
    int d = n % b;
    n /= b;
    s[pos++] = (d < 10)? (d + '0') : ('A' + d - 10);
  }while(n != 0);
  if(sign < 0)
    s[pos++] = '-';
  s[pos] = '\0';
  reverse(s, s + pos);
  return s;
}
```

Example:

```
puts(toBase(65, 2));  // 1000001
puts(toBase(10, 16)); // A
puts(toBase(98, 36)); // 2Q
```

UTF-8

INPUT

```
setlocale(LC_ALL, "en_US.utf8"); // Just once, after "int main()"
wchar_t s[100];   // Declare a wide string
scanf("%ls\n",s); // Read a wide string
```

LENGHT OF A WCHAR_T*

```
int wstrlen(wchar_t* ws) // Returns the lenght of a wchar_t*
{
  int ans = 0;
  for(; *ws; ws++, ans++);
  return ans;
}
```

WCHAR_T TO CHAR

```
char wcharToChar(wchar_t x) // wchar_t to char
{
  char c[10]; wctomb(c, x);
  return c[0];
}
```

WCHAR_T* TO CHAR*

```
void wc_str(wchar_t* ws, char *s) // Convierte wchar_t* en char*
{
  int n = wstrlen(ws);
  for(int i = 0; i < n; i++)
    s[i] = wcharToChar(ws[i]);
  s[n] = '\0';
}
```

C++ FUNCTIONS

TRIGONOMETRIC

acos()	double acos(double x)	Arc cosine
asin()	double asin(double x)	Arc sin
atan()	double atan(double x)	Arc tangent
atan2()	double atan2(double x, double y)	Arc tangent of x / y
cos()	double cos(double x)	Cosine
sin()	double sin(double x)	Sine
tan()	double tan(double x)	Tangent

EXPONENTIAL AND LOGARITHMIC

exp()	double exp(double x)	e^x where e = 2.7182818284590452354
log()	double log(double x)	Natural Logarithm
log10()	double log10(double x)	Logarithm base 10

HYPERBOLIC

cosh()	double cosh(double x)	Hiperbolic cosine
sinh()	double sinh(double x)	Hiperbolic sine
tanh()	double tanh(double x)	Hiperbolic tangent

MATH

sqrt()	double sqrt(double x)	Square root
ceil()	double ceil(double x)	Rounds x upward
floor()	double floor(double x)	Rounds x downward
abs()	int abs(int x)	Absolute value
labs()	long labs(long x)	Absolute value for long
modf()	double modf(double x, double *y)	Breaks x into integral and fractional part
pow()	double pow(double x, double y)	Raises x to the power of y

CHARACTER VALIDATION

isalnum(int c)	Is letter or digit
isalpha(int c)	Is letter
isascii(int c)	Is ASCII character (between 0 y 127)
iscntrl(int c)	Is a control character
isdigit(int c)	Is a digit
isgraph(int c)	Is a printable character (except space character)
islower(int c)	Is lower case letter
isprint(int c)	Is a printable character (including space character)
ispunct(int c)	Is a punctuation letter
isupper(int c)	Is upper case letter
isxdigit(int c)	Is hexadecimal digit (0-9, a-f, A-F)
toupper(int c)	Converts the character to its upper case equivalent
tolower(int c)	Converts the character to its lower case equivalent

STRING

size_t **strlen**(char *str);	Length of the string str
char *__strcpy__(char *destination, char *source);	Copies source to destination
char *__strncpy__(char *destination, char *source, size_t n);	Copies the first n characters of source to destination
char *__strdup__(char *source);	Duplica una cadena. Reserva su propio espacio en memoria y devuelve la copia de la cadena.
char *__strcat__(char *str1, char *str2);	Concatena str1 y str2, guardando el resultado en str1
int **strcmp**(char *str1, char *str2);	Compara dos cadenas < 0 str1 es menor que str2, == 0 son iguales, > 0 str1 es mayor que str2
int **strncmp**(char *str1, char *str2, size_t n);	Igual a strcmp pero solo compara n caracteres.
char *__strchr__(char *str, int ch);	Devuelve un puntero a la primera ocurrencia de ch en la cadena str.
size_t **strcspn**(char *str1, char *str2);	Devuelve la posición del primer carácter que se encuentre entre los caracteres que estén en str2 dentro de str1. No busca la subcadena str2, sino que busca algún carácter de str2 dentro de str1. Si ningún carácter de str2 se encuentra en str1 devuelve strlen.
char *__strstr__(char *str1, char *str2);	Devuelve un puntero a la primera subcadena str2 que encuentre en str1. Si no la encuentra devuelve NULL.
int **atoi**(char *ptr);	Convierte una cadena a un número entero. Convierte hasta encontrar un carácter inválido.
long **atol**(char *ptr);	Convierte una cadena a long. Convierte hasta encontrar un carácter inválido.
double **atof**(char *str);	Convierte una cadena a double. Convierte hasta encontrar un carácter inválido. Acepta cadenas como 123E+3

DATA STRUCTURES

PAIR

```
typedef pair<int,int> ii;
typedef pair<int,ii> iii;
// Access values
ii p;
p.first = 4;
p.second = 7;
// Asignment
iii r = iii(1, ii(2,3));
```

VECTOR

```
typedef vector<int> vi; // Shortcut
vi v(10, 0);          // Creates an array v of size 10, every cell as 0
int n = v.size();     // Gets number of elements
v[3] = 45;            // Sets value of v[3]
v.push_back(10);      // Inserts a new number (v[11] = 10)
vi::iterator it;      // Iterator
it = v.begin();       // Moves iterator to 1st element (points to v[0])
it += 4;              // Moves 4 times the iterator (points to v[5])
it = v.insert(it, 5); // Inserts 5 before iterator and updates it
v.insert(it, 2, 6);   // Inserts 6 two times before iterator()
v.erase(it, it + 4);  // Erases 4 elements starting at iterator
v.pop_back();         // Erases last element
v.clear();            // Clears the array (new size is 0)
// Print
for(it = v.begin(); it != v.end(); it++)
  cout << (*it);
// Print in reverse order
for(vi::reverse_iterator rit = v.rbegin(); rit != v.rend(); rit++)
  cout << (*rit);
```

STACK

```
stack<pair<int,double> > st;      // Careful not to put ">>"
st.push(make_pair(4, 3.5));       // Inserts a pair
bool f = st.empty();              // Validates if stack is empty
int n = st.size();                // Gets number of elements
pair<int,double> par = st.top(); // Retrieves top of the stack
st.pop();                         // Erases top of the stack
```

QUEUE

```
queue<pair<int,double> > q;         // Careful not to put ">>"
q.push(make_pair(4, 3.5));          // Pushes a pair
bool f = q.empty();                 // Validates if queue is empty
int n = (int)q.size();              // Gets number of elements
pair<int, double> par = q.front(); // Retrieves front of the queue
q.pop();                            // Erases front of the queue
```

PRIORITY QUEUE

```
priority_queue<int> pq;                              // Max Heap
priority_queue<int, vector<int>, greater<int> > pq; // Min Heap
// Using our own struct
struct group { int a, b; };
bool operator < (const group &x, const group &y)
{
  return (x.a != y.a)? (x.a < y.a) : (x.b < y.b)  // Sort by "a", then by "b"
}
priority_queue<group> pq; // Max Heap
```

MAP

Container that stores elements formed by <*key value*, *mapped value*>. It is implemented as a red-black tree so insertions and lookups are guaranteed to be in O(lg N).

```
map<string,int> mapa; // Associates a string (Month) to an integer (Days)
mapa["Jan"] = 31;     // If "Jan" does not exist, its key value is set.
mapa["Feb"] = 28;     // If "Feb" does exist, its key value is updated.
// Finding a key value
string s = "Mar";
if(mapa.find(s) != mapa.end())
  cout << s << " exists and has " << mapa[s] << "days.";
// Prints the map. Elements are sorted by key value
for(map<string,int>::iterator it = mapa.begin(); it != mapa.end(); it++)
  cout << "Month " << (*it).first << " has " << (*it).second << " days";
```

SET

Container that stores unique elements. It is implemented as a red-black tree.

```
set<ii> s;
s.insert(ii(10,5));
s.insert(ii(18,3));
// Finding an element
if(s.find(ii(10,5)) != s.end())
  cout << "Pair (10, 5) was found";
// Inorder Traversal
for(set<group>::iterator it = s.begin(); it != s.end(); it++)
  cout << (*it).a + (*it).b << " = " << (*it).a << " + " << (*it).b;
// Postorder Traversal
for(set<group>::reverse_iterator it = s.rbegin(); it != s.rend(); it++)
  cout << (*it).a + (*it).b << " = " << (*it).a << " + " << (*it).b;
```

MULTISET

A set that allows duplicate elements.

```
multiset<int> s;                    // Stores increasingly
multiset<int, greater<int> > t;    // Stores decreasingly
multiset<int>::iterator it;        // Iterator
s.insert(4);                        // Insert an element
```

```
int cont = s.count(5); // Count number of appearances of element
bool f = s.find(3);    // Find an element
s.erase(4);            // Erases all elements equivalent to number 4
s.erase(it);           // Erases number which is pointed by iterator
// Print numbers increasingly
for(it = s.begin(); it != s.end(); it++)
  cout << " " << (*it);
// Find max number
it = s.begin();
int maxi = (*it);
// Find min number
it = S.end(); it--;
int mini = (*it);
```

UNION-FIND DISJOINT SET

```
vi pset;   // pset[i]: Boss of node i
vi sset;   // sset[i]: Number of nodes that depend on node i
int nSets; // Total number of sets

void init(int n)   // n: Number of nodes
{
  nSets = n;       // There are n sets
  sset = vi(n, 1); // Each set has size one
  pset = vi(n, 0); // Assign capacity for n nodes
  for(int i = 0; i < n; i++) // For each node i..
    pset[i] = i;             // Node i is its own boss
}

int findSet(int i) // Returns the final boss of node i
{
  return (pset[i] == i)? i : (pset[i] = findSet(pset[i]));
}

bool isSameSet(int i, int j) // Checks if two nodes belong to the same set
{
  return findSet(i) == findSet(j); // Check if nodes have the same boss
}

void unionSet(int i, int j)  // Joins the sets of node i and node j
{
  if(!isSameSet(i, j)) // If the nodes belong to different sets..
  {
    nSets--; // After merging two sets, the total number decreases in one
    // Keep boss of j as the main one...
    sset[findSet(j)] += sset[findSet(i)]; // Increase set where j
    pset[findSet(i)] = findSet(j);        // Boss of i changes to boss of j
  }
}
```

EVALUATING PROPERTIES IN CONNECTED COMPONENTS

To check a property in a connected component, evaluate the boss of the set.

1. Size of the set to which node i belongs

```
cout << "Size of the set containing node " << i << ": " << sset[findSet(i)];
```

2. Size of the largest connected component

```
int ans = 0;
for(int i = 0; i < n; i++)
  ans = max(ans, sset[findSet(i)]);
cout << "Size of the largest connected component: " << ans;
```

3. Number of connected components that have even number of nodes.

```
int ans = 0;
for(int i = 0; i < n; i++)
  if(findSet(i) == i && sset[findSet(i)] % 2 == 0)
    ans++;
cout << "Connected componentes that have even number of nodes : " << ans;
```

4. Check if a component has nodes only with even degree.

```
vi deg;                        // Degree of each node
vb evendeg = vb(n, true);    // True is the neutral value of && operator
for(int i = 0; i < n; i++)   // For each node..
  evendeg[findSet(i)] &&= (deg[i] % 2 == 0); // Only update the boss
bool f = evendeg[findSet(x)]; // To evaluate the property in any node x
```

BINARY INDEXED TREE / FENWICK TREE

```
#define LSOne(S) (S & (-S))  // Least Significant One
vi t;  // Fenwick Tree
int n; // Number of elements

void inc(int i, int val) // Increases v[i] by "val"
{
  for(i++; i <= n; i += LSOne(i))
    t[i] += val;
}

int rsq(int i) // Range Sum Query in range [0, x]
{
  int sum = 0;
  for(i++; i; i -= LSOne(i))
    sum += t[i];
  return sum;
}

int rsq(int l, int r) // Range Sum Query in range [l, r]
{
  return rsq(r) - rsq(l - 1);
}
```

How to use:

```
cin >> n;          // Read number of elements
t = vi(n + 1, 0); // Initialize tree for n elements (all in zero)
for(int i = 0; i < n; i++) // For each element..
{
  int v; cin >> v; // Read number at position i
  inc(i, v);        // Update tree
}
cout << "RSQ(0,3) = " << rsq(0,3); // Range Sum Query in range [0,3]
```

BIT 2D

```
#define LSOne(S) (S & (-S))  // Least Significant One
const int MAX = 1025; // Max number of elements
int t[MAX][MAX];      // Initialize it with memset(t, 0, sizeof t)
int r,c;              // r: Rows, c: Columns

void inc(int x, int y, int val) // Increases value at position (x,y) by "val"
{
  int py = y;
  for(x++; x <= r; x += LSOne(x))
    for(y = py + 1; y <= c; y += LSOne(y))
      t[x][y] += val;
}

int rsq(int x, int y) // Range Sum Query on range [(0,0); (x,y)]
{
  int ans = 0, py = y;
  for(x++; x; x -= LSOne(x))
    for(y = py + 1; y; y -= LSOne(y))
      ans += t[x][y];
  return ans;
}

int rsq(int sx, int sy, int tx, int ty) // RSQ on range [(sx,sy); (tx,ty)]
{
  return rsq(tx, ty) - rsq(sx-1, ty) - rsq(tx, sy-1) + rsq(sx-1, sy-1);
}
```

It is used in a similar way to BIT 1D.

SPARSE TABLE

```
const int MAX = 100005; // Max number of elements
int t[MAX][17]; // t[i][k] covers range [i, i + 2^k - 1]. (2^17 ~ 100K nodes)
int n;          // Number of elements

void build()
{
  // Key idea: We can cover the range 2^k with two segments of 2^(k-1)
  for(int k = 1; (1 << k) <= n; k++) // For each range k until 2^k <= n..
    for(int i = 0; i + (1 << k) - 1 < n; i++) // For each element i..
      t[i][k] = max(t[i][k-1], t[i + (1 << (k-1))][k-1]); // Two segments
}
```

```
int rmq(int l, int r) // Query in range [l, r]
{
  int k = 0;                   // max k such that 2^k covers the range [l,r]
  while((1 << k) <= r - l + 1) // While 2^k covers the range..
    k++;                       // Increase k
  k--;                         // Fixed the value of k
  return max(t[l][k], t[r - (1 << k) + 1][k]); // [l,l+2^k-1] && [r-2^k+1,r]
}
```

It is required to read the elements of the array in the following way:

```
for(int i = 0; i < n; i++)
  cin >> t[i][0]; // Caso base: Solo 1 elemento
build();
cout << "RMQ(0,3) = " << rmq(0,3); // Query in range [x, y]
```

SEGMENT TREE

Supports Range Sum Query

```
const int MAX = 1e5; // Max number of elements
int t[MAX*2];        // Segment Tree (Root is t[1])
int n;               // Number of elements of the array

void build() // Builds the segment tree
{
  for(int i = n - 1; i > 0; i--)       // For each non-leaf node..
    t[i] = t[i << 1] + t[i << 1 | 1]; // Update according to both children
}

void update(int i, int val) // Sets array[i] to val (i in range [0, N-1])
{
  for(t[i += n] = val; i >>= 1; )      // Update leaf node, then go up..
    t[i] = t[i << 1] + t[i << 1 | 1]; // Update according to children
}

int query(int l, int r) // Range Sum Query in range [l,r]
{
  int ans = 0;
  for(l += n, r += n; l <= r; l = (l + 1) >> 1, r = (r - 1) >> 1)
  {
    if(  l & 1 ) ans = ans + t[l]; // For left pointer, odd nodes matter
    if(!(r & 1)) ans = t[r] + ans; // For right pointer, even nodes matter
  }
  return ans;
}
```

Read the elements of the array in the following way:

```
for(int i = 0; i < n; i++) // For each element i..
  cin >> t[n + i];         // Read i at position n + i
```

RANGE INCREASE QUERY + SINGLE POINT QUERY

```
const int MAX = 1e5; // Max number of elements
int t[MAX*2];        // Segment Tree (Root is t[1])
int n;               // Number of elements of the array

void inc(int l, int r, int val) // Increases range [l,r] by val
{
  for(l += n, r += n; l <= r; l = (l + 1) >> 1, r = (r - 1) >> 1)
  {
    if(  l & 1 ) t[l] += val; // For left pointer, odd nodes matter
    if(!(r & 1)) t[r] += val; // For right pointer, even nodes matter
  }
}

int query(int i) // Get value of i-th element in the array
{
  int ans = 0;
  for(i += n; i > 0; i >>= 1)
    ans += t[i];
  return ans;
}

void push() // Push all modifications to leaf nodes in O(n)
{
  for(int i = 1; i < n; i++)
    t[i << 1] += t[i],
    t[i << 1 | 1] += t[i],
    t[i] = 0;
}
```

WITH ARRAYS

Find number of elements greater/lower than a fixed value x in O(lg^2 n)

```
#define all(x) (x).begin(), (x).end()
const int MAX = 1e5; // Max number of elements
vi t[MAX*2];         // Segment Tree (Root is t[1])
int n;               // Number of elements of the array

void build() // Builds the segment tree
{
  for(int i = n - 1; i > 0; i--)
  {
    t[i].resize(t[i << 1].size() + t[i << 1 | 1].size());    // Prepare size
    merge(all(t[i << 1]), all(t[i << 1 | 1]), t[i].begin()); // Merge
  }
}

int query(int l, int r, int x) // Number of elements less than x in [l,r]
{
  int ans = 0;
  for(l += n, r += n; l <= r; l = (l + 1) >> 1, r = (r - 1) >> 1)
  {
    if(  l & 1 ) ans += lower_bound(all(t[l]), x) - t[l].begin();
    if(!(r & 1)) ans += lower_bound(all(t[r]), x) - t[r].begin();
  }
```

```
  return ans;
}

int query(int l, int r, int x) // Number of elements greater than x in [l,r]
{
  int ans = 0;
  for(l += n, r += n; l <= r; l = (l + 1) >> 1, r = (r - 1) >> 1)
  {
    if(  l & 1 ) ans += t[l].end() - upper_bound(all(t[l]), x);
    if(!(r & 1)) ans += t[r].end() - upper_bound(all(t[r]), x);
  }
  return ans;
}
```

Read the elements of the array in the following way:

```
for(int i = 0; i < n; i++) // For each element i..
{
  cin >> x;
  t[n + i] = vi(1, x); // Read i at position n + i
}
```

SEGMENT TREE 2D

```
const int MAX = 505; // Max number of elements
int t[MAX*2][MAX*2]; // Segment Tree (Root is t[1][1])
int r,c;             // r: Rows, c: Columns

void build()
{
  for(int x = 2 * r - 1; x > 0; x--)    // For each row..
    for(int y = 2 * c - 1; y > 0; y--) // For each col..
      if(x >= r && y < c)      // Expand on y axis
        t[x][y] = max(t[x][y << 1], t[x][y << 1 | 1]);
      else if(x < r && y < c) // Expand on x axis
        t[x][y] = max(t[x << 1][y], t[x << 1 | 1][y]);
}

void update(int x, int y, int val)
{
  for(t[x += r][y += c] = val; x > 0; x >>= 1)
    for(int i = y; i > 0; i >>= 1)
      if(x >= r && i < c)
        t[x][i] = max(t[x][i << 1], t[x][i << 1 | 1]);
      else if(x < r && i < c)
        t[x][i] = max(t[x << 1][i], t[x << 1 | 1][i]);
}

int query(int sx, int sy, int tx, int ty)
{
  int ans = 0;
  for(sx += r, x2 += r; sx <= x2; sx = (sx+1) >> 1, x2 = (x2-1) >> 1)
    for(int i1=sy+c, i2=ty+c; i1 <= i2; i1 = (i1+1) >> 1, i2 = (i2-1) >> 1)
    {
      if( (sx & 1) &&  (i1 & 1)) ans = max(ans, t[sx][i1]);
```

```
      if( (sx & 1) && !(i2 & 1)) ans = max(ans, t[sx][i2]);
      if(!(tx & 1) &&  (i1 & 1)) ans = max(ans, t[tx][i1]);
      if(!(tx & 1) && !(i2 & 1)) ans = max(ans, t[tx][i2]);
    }
  return ans;
}
```

Read the elements of the array in the following way:

```
for(int i = 0; i < r; i++)   // For each row i
  for(int j = 0; j < c; j++) // For each column j..
    cin >> M[r + i][c + j];  // Read (i,j) at position (r+i, c+j)
```

LAZY PROPAGATION

Supports Range Increment Query and Range Sum Query.

```
const int MAX = 1e5; // Max number of elements
int t[MAX*2];        // Segment Tree (Root is index 1)
int n;               // Number of elements of the array
int d[MAX];          // d[i]: Value that node i has to propagate
int q[MAX*2];        // q[i]: Size of the array covered by node i
int h;               // h: Height of the tree

void build() // Builds the segment tree
{
  for(int i = n; i < 2 * n; i++)
    q[i] = 1;
  for(int i = n - 1; i > 0; i--)
    q[i] = q[i << 1] + q[i << 1 | 1],
    d[i] = 0,
    t[i] = t[i << 1] + t[i << 1 | 1];
  h = sizeof(int) * 8 - __builtin_clz(n);
}

void apply(int i, int val) // Increments i-th node by val
{
  t[i] += val * q[i];
  if(i < n)
    d[i] = d[i] + val;
}

void pull(int i) // Updates the path from node i to root
{
  while(i >>= 1)
    t[i] = t[i << 1] + t[i << 1 | 1] + d[i] * q[i];
}

void push(int i) // Propagates the path from root to node i
{
  for(int s = h; s > 0; s--)
  {
    int p = i >> s;
    if(d[p] != 0)
    {
```

```
      apply(p << 1, d[p]);
      apply(p << 1 | 1, d[p]);
      d[p] = 0;
    }
  }
}

void inc(int l, int r, int val) // Increment Range [l,r] by val
{
  int l0 = l + n, r0 = r + n;
  push(l0), push(r0);
  for(l += n, r += n; l <= r; l = (l + 1) >> 1, r = (r - 1) >> 1)
  {
    if(  l & 1 ) apply(l, val);
    if(!(r & 1)) apply(r, val);
  }
  pull(l0), pull(r0);
}

int query(int l, int r) // Range Sum Query in [l,r]
{
  int ans = 0;
  push(l + n), push(r + n);
  for(l += n, r += n; l <= r; l = (l + 1) >> 1, r = (r - 1) >> 1)
  {
    if(  l & 1 ) ans = ans + t[l];
    if(!(r & 1)) ans = t[r] + ans;
  }
  return ans;
}
```

Read the elements of the array in the following way:

```
for(int i = 0; i < n; i++) // For each element i..
  cin >> t[n + i];         // Read i at position n + i
```

HEAVY-LIGHT DECOMPOSITION

WEIGHTED EDGES

```
const int INF = 200000000;
const int MAX = 10005; // Max number of nodes
vvii g; // g: Tree
int n;  // n: Number of nodes in graph
// Segment Tree
int t[MAX*2]; // Segment Tree (Root is index 1)

void update(int i, int val) // Segment Tree update
{
  for(t[i += n] = val; i >>= 1; )
    t[i] = max(t[i << 1], t[i << 1 | 1]);
}

int query(int l, int r) // Segment Tree query
```

```
{
  int ans = -INF;
  for(l += n, r += n; l <= r; l = (l + 1) >> 1, r = (r - 1) >> 1)
  {
    if (  l & 1 ) ans = max(ans, t[l]);
    if (!(r & 1)) ans = max(t[r], ans);
  }
  return ans;
}

// HLD
int nxt[MAX],parent[MAX],depth[MAX],chain[MAX],stPos[MAX];
// nxt[i]: Next node after i in the chain (-1 if none)
// parent[i]: Parent of node i
// depth[i]: Height of node i
// chain[i]: First node in the chain where node i belongs
// stPos[i]: Position of node i in the segment tree

int dfs(int v, int p = -1) // Returns size of subtree of v
{
  parent[v] = p;
  depth[v] = (p != -1)? depth[p] + 1 : 0;
  int size = 1, maxi = 0;
  for(int i = 0; i < (int)g[v].size(); i++)
  {
    int u = g[v][i].second;
    if(u != p)
    {
      int subtree = dfs(u, v); // Go to neighbor u
      if(subtree > maxi) // If node u is the 'heaviest' son..
        nxt[v] = u, maxi = subtree; // Node u will be the next one in chain
      size += subtree;   // Increase current subtree size
    }
  }
  return size;
}

void init(int r = 0) // Initializes heavy-light with root r
{
  memset(nxt, -1, sizeof nxt); // Clear chain information
  dfs(r); // DFS over the root to build chains
  for(int i = 0, cont = 0; i < n; i++)
    if(parent[i] == -1 || nxt[parent[i]] != i)
      for(int j = i; j != -1; j = nxt[j])
      {
        chain[j] = i;
        stPos[j] = cont++;
      }
}

// Set weight of edge ending in 'v' to val
void updateEdge(int v, int val) { update(stPos[v], val); }

int queryPath(int p, int q) // Get max edge in path from node p to node q
{
  int ans = -INF;
```

```
  for(; chain[p] != chain[q]; q = parent[chain[q]])
  {
    if(depth[chain[p]] > depth[chain[q]])
      swap(p, q);
    ans = max(query(stPos[chain[q]], stPos[q]), ans);
  }
  if(depth[p] > depth[q])
    swap(p, q);
  ans = max(query(stPos[p] + 1, stPos[q]), ans);
  return ans;
}

// Set initial values
int otherEnd[MAX];
vvi edgeIdx;

void dfsTree(int v = 0, int p = -1)
{
  for(int i = 0; i < (int)g[v].size(); i++)
  {
    int u = g[v][i].second, w = g[v][i].first;
    int e = edgeIdx[v][i];
    if(u != p)
    {
      otherEnd[e] = u;
      updateEdge(u, w);
      dfsTree(u, v);
    }
  }
}
```

Ejemplo:

```
g = vvii(n); edgeIdx = vvi(n);
for(int i = 0; i < n - 1; i++)
{
  cin >> x >> y >> w; --x; --y;
  g[x].push_back(ii(w,y)); edgeIdx[x].push_back(i);
  g[y].push_back(ii(w,x)); edgeIdx[y].push_back(i);
}
init();
dfsTree();
while(cin >> opc, opc[0] != 'D')
{
  cin >> x >> y;
  if(opc[0] == 'C')
    updateEdge(otherEnd[--x], y);
  else
    cout << queryPath(--x, --y) << '\n';
}
```

WEIGHTED NODES

```
const int MAX = 100005;
const int INF = 20000000;
```

```
vvi g;
int n;
// Segment Tree
int t[MAX*2];

void build()
{
  for(int i = n - 1; i > 0; i--)
    t[i] = max(t[i << 1], t[i << 1 | 1]);
}

void update(int i, int val) // Segment Tree update
{
  for(t[i += n] = val; i >>= 1; )
    t[i] = max(t[i << 1], t[i << 1 | 1]);
}

int query(int l, int r)
{
  int ans = -INF;
  for(l += n, r += n; l <= r; l = (l + 1) >> 1, r = (r - 1) >> 1)
  {
    if(  l & 1 ) ans = max(ans, t[l]);
    if(!(r & 1)) ans = max(t[r], ans);
  }
  return ans;
}

// HLD
int nxt[MAX],chain[MAX],depth[MAX],parent[MAX],stPos[MAX];

int dfs(int v, int p = -1)
{
    parent[v] = p;
    depth[v] = (p != -1)? depth[p] + 1 : 0;
    int size = 1, maxi = 0;
    for(int i = 0; i < (int)g[v].size(); i++)
    {
        int u = g[v][i];
        if(u != p)
        {
            int subtree = dfs(u, v);
            if(subtree > maxi)
                nxt[v] = u, maxi = subtree;
            size += subtree;
        }
    }
    return size;
}

void init(int r = 0)
{
    memset(nxt, -1, sizeof nxt);
    dfs(r);
    for(int i = 0, cont = 0; i < n; i++)
        if(parent[i] == -1 || nxt[parent[i]] != i)
```

```
            for(int j = i; j != -1; j = nxt[j])
            {
                chain[j] = i;
                t[cont + n] = j; // Initial value of j-th element
                stPos[j] = cont++;
            }
    build();
}

void updateNode(int v) { update(stPos[v]); }

int queryPath(int p, int q)
{
    int ans = -INF;
    for(; chain[p] != chain[q]; q = parent[chain[q]])
    {
        if(depth[chain[p]] > depth[chain[q]])
            swap(p, q);
        ans = max(query(stPos[chain[q]], stPos[q]), ans);
    }
    if(depth[p] > depth[q])
        swap(p, q);
    ans = max(query(stPos[p], stPos[q]), ans);
    return ans;
}
```

Example:

```
g = vvi(n);
for(int i = 0; i < n - 1; i++)
{
  cin >> x >> y; --x; --y;
  g[x].push_back(y);
  g[y].push_back(x);
}
init();
updateNode(nodeIdx, weight);
queryPath(x, y);
```

POLICY-BASED STRUCTURES

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
```

ORDER STATISTICS SET

```
#include <ext/pb_ds/tree_policy.hpp>

tree<int,     // Key type
null_type,    // Mapped value
less<int>,    // Key comparison
rb_tree_tag, // Data structure to use
tree_order_statistics_node_update> // Policy for updating nodes
t;
```

Operations:

```
t.find(x) == t.end(); // Evaluates if element x is inserted
t.insert(x);          // Inserts element x
t.erase(x);           // Erases element x
cout << *t.find_by_order(k); // Find K-th smallest element (0-indexed)
cout << t.order_of_key(x);   // Number of elements smaller than x
```

TREAPS

SPLIT/MERGE

```
struct node {
  int x, y, sz; // x: Key, y: Priority, sz: Subtree size
  node *l, *r;  // l: Left subtree, r: Right Subtree
  node(int x) : x(x), y(rand()), sz(1), l(NULL), r(NULL) {}
};

typedef node* pnode;

int sz(pnode t) { return t? t->sz : 0; } // Subtree size of node t

void upd(pnode t) { if(t) t->sz = 1 + sz(t->l) + sz(t->r); } // Updates size

void split(pnode t, int x, pnode &l, pnode &r)
{
  if(!t)
    l = r = NULL;
  else if(x < t->x)
    split(t->l, x, l, t->l), r = t;
  else
    split(t->r, x, t->r, r), l = t;
  upd(t);
}

void merge(pnode &t, pnode l, pnode r)
{
  if(!l || !r)
    t = l? l : r;
  else if(l->y > r->y)
    merge(l->r, l->r, r), t = l;
  else
    merge(r->l, l, r->l), t = r;
  upd(t);
}

bool find(pnode t, int x) // t: Root node, x: Key to find
{
  if(!t)
    return false;
  if(t->x == x)
    return true;
  return find(x < t->x? t->l : t->r, x);
}
```

```
void insert(pnode &t, pnode it) // t: Root node, it: Node to insert
{
  if(!t)
    t = it;
  else if(it->y > t->y)
    split(t, it->x, it->l, it->r), t = it;
  else
    insert(it->x < t->x? t->l : t->r, it);
  upd(t);
}

void erase(pnode &t, int x) // t: Root node, x: Key to delete
{
  if(!t)
    return;
  else if(t->x == x)
    merge(t, t->l, t->r);
  else
    erase(x < t->x? t->l : t->r, x);
  upd(t);
}
```

Example:

```
/// Declare treap
pnode t = NULL;
/// Insert key 5
insert(t, new node(5));
/// Busqueda y eliminacion
pnode f = find(t, 5);
if(!f)
  cout << "Not found";
else
  erase(t, 5);
```

FIND K-TH ELEMENT [0, N-1]

```
pnode kth(pnode t, int k) // t: Root node, k: Index [0, N-1]
{
  if(k >= sz(t))    // If k is greater that num of elements..
    return NULL;    //  No answer
  int s = sz(t->l); // Left substree size
  if(k == s)        // The index is the same as the elements in the left..
    return t;       //  Node t is the kth element
  else if(k < s)    // The index is lower..
    return kth(t->l, k); // Find kth index in left subtree
  else              // The index is higher..
    return kth(t->r, k - s - 1); // Find (k-s-1)th index in right subtree
}
```

CONTAR ELEMENTOS MENORES A UNA CLAVE

```
int count(pnode t, int x) // Cuenta los elementos menores a x
{
  if(!t)
    return 0;
  if(x > t->x)
    return 1 + sz(t->l) + count(t->r, x);
  else
    return count(t->l, x);
}
```

ALGORITHMS IN STL

```
// Inicializacion de vector y arreglo
int arr[5] = {1, 2, 3, 4, 5};
vector<int> v(arr, arr + 5);

// Recorrer
void for_each_function(int val) { printf("%d\n", val); }
void for_each_function_modify(int &val) { val++; }

for_each(arr, arr + 5, for_each_function);
for_each(v.begin(), v.end(), for_each_function);
for_each(v.begin(), v.end(), for_each_function_modify);

// Buscar en arreglo
int *p = find(arr, arr + 5, 3);
if(p) printf("Se encontro 3 con valor %d\n", *p);
else  printf("No se encontro el 3\n");

// Buscar en vector
vector<int>::iterator it = find(v.begin(), v.end(), 3);
if(it != v.end()) printf("Se encontro 3 con valor %d\n", *it);
else printf("No se encontro el 3\n");

// Buscar si
bool esPar(int i) { return (i % 2) == 0; }
// Primer numero par. Se accede con *it
vector<int>::iterator it = find_if(v.begin(), v.end(), esPar);
// Ultimo numero par. Se accede con *rit
vector<int>::reverse_iterator rit = find_if(v.rbegin(), v.rend(), esPar);

// Cuenta cuantos numeros 4 hay en el vector o arreglo
int cant = (int)count(arr, arr + 5, 4);

// Cuenta cuantos numeros pares hay en el vector o arreglo
int cantPar = count_if(vec.begin(), vec.end(), esPar);

// Busca un subgrupo
int sub[3] = {2, 3, 4};
vector<int>:: iterator it = search(v.begin(), v.end(), sub, sub + 3);
if(it != v.end()) printf("Posicion: %d\n", (int)(it - vec.begin()));
else printf("No se encontro el subgrupo");
```

```
// Busca 2 numeros 30 en el vector
vector<int>::iterator it = search_n(v.begin(), v.end(), 2, 30);

// Compara si son iguales -- puede ser list.begin() en lugar de arr
if(equal(vec.begin(), vec.end(), arr)) printf("Son iguales");

// Primeros elementos que difieren
pair<vector<int>::iterator, int*> par;
par = mismatch(vec.begin(), vec.end(), arr);
printf("Primeros diferentes: %d y %d\n", *par.first, *par.second);
par.first++; par.second++;
par = mismatch(par.first, vec.end(), par.second);
printf("Segundos diferentes: %d y %d\n", *par.first, *par.second);

// Invertir
reverse(arr, arr + 5); // arr = { 5, 4, 3, 2, 1 }

// Rotar
rotate(arr, arr + 1, arr + 5); // arr = { 4, 3, 2, 1, 5 }

// Barajar al azar
random_shuffle (arr, arr + 5);

// Minimo y Maximo
int a = min(3, 2);
int b = max(4, 8);

// Minimo y Maximo valor
int *c = min_element(arr, arr + 5);
int *d = max_element(arr, arr + 5);
printf("Maximo: %d\n", *max_element(arr, arr + 5));

// Comparacion lexicografica
char uno[] = "Azzzz";      // 5 letras
char dos[] = "azaaaaaab";  // 9 letras

if(lexicographical_compare(uno, uno + 5, dos, dos + 9))
   printf("%s es menor que %s\n", uno, dos);
else if(lexicographical_compare(dos, dos + 9, uno, uno + 5))
   printf("%s es mayor que %s\n", uno, dos);
else printf("%s y %s son iguales\n", uno, dos);

// Comparacion lexicografica case insensitive
bool miComp(char c1, char c2) { return tolower(c1) < tolower(c2); }

char uno[] = "Azzzz";      // 5 letras
char dos[] = "azaaaaaab";  // 9 letras

if(lexicographical_compare(uno, uno + 5, dos, dos + 9, miComp))
   printf("%s es menor que %s\n", uno, dos);
else if(lexicographical_compare(dos, dos + 9, uno, uno + 5, miComp))
   printf("%s es mayor que %s\n", uno, dos);
else printf("%s y %s son iguales\n", uno, dos);

// Generar todas las permutaciones
char cadena[6] = "abcde";
```

```
int len = strlen(cadena);
sort(cadena, cadena + len);
do {
   puts(cadena);
}while(next_permutation(cadena, cadena + len));

//Elimina los elementos duplicados
v.erase(unique(v.begin(), v.end()),v.end());

// Llena un vector
vector<int> v(8, 0);                  // v: 0 0 0 0 0 0 0 0
fill(v.begin(), v.end(), 2);          // v: 2 2 2 2 2 2 2 2
fill(v.begin(), v.begin() + 4, 5);    // v: 5 5 5 5 2 2 2 2
fill(v.begin() + 3, v.end() - 2, 8); // v: 5 5 5 8 8 8 2 2

// Copiar map a vector
map<string, int> M;
vector<pair<string, int> > V(M.begin(), M.end());
```

SET_SYMMETRIC_DIFFERENCE

```
int A[] = { 5, 10, 15, 20, 25};
int B[] = {50, 40, 30, 20, 10};
vector<int> v(10, 0);  // 0  0  0  0  0  0  0  0  0  0
vector<int>::iterator it;
sort(A, A + 5);        //  5 10 15 20 25
sort(B, B + 5);        // 10 20 30 40 50
it = set_symmetric_difference(A, A + 5, B, B + 5, v.begin());
                       //  5 15 25 30 40 50  0  0  0  0
cout << "sym. difference has " << int(it - v.begin()) << " elements.\n";
```

SET_UNION

```
int A[] = { 5, 10, 15, 20, 25};
int B[] = {50, 40, 30, 20, 10};
vector<int> v(10, 0);  // 0  0  0  0  0  0  0  0  0  0
vector<int>::iterator it;
sort(A, A + 5);        //  5 10 15 20 25
sort(B, B + 5);        // 10 20 30 40 50
it = set_union(A, A + 5, B, B + 5, v.begin());
                       // 5 10 15 20 25 30 40 50  0  0
cout << "union has " << int(it - v.begin()) << " elements.\n";
```

SET_INTERSECTION

```
int A[] = {5, 10, 15, 20, 25};
int B[] = {50, 40, 30, 20, 10};
vector<int> v(10, 0);  // 0  0  0  0  0  0  0  0  0  0
vector<int>::iterator it;
sort(A, A + 5);        //  5 10 15 20 25
sort(B, B + 5);        // 10 20 30 40 50
it = set_intersection(A, A + 5, B, B + 5, v.begin());
                       // 10 20 0  0  0  0  0  0  0  0
```

```
cout << "intersection has " << int(it - v.begin()) << " elements.\n";
```

SET_DIFFERENCE

```
int A[] = {5, 10, 15, 20, 25};
int B[] = {50, 40, 30, 20, 10};
vector<int> v(10, 0);  // 0  0  0  0  0  0  0  0  0  0
vector<int>::iterator it;
sort(A, A + 5);        //  5 10 15 20 25
sort(B, B + 5);        // 10 20 30 40 50
it = set_difference(A, A + 5, B, B + 5, v.begin());
                       //  5 15 25  0  0  0  0  0  0  0
cout << "difference has " << int(it - v.begin()) << " elements.\n";
```

MERGE

```
int A[] = {5, 10, 15, 20, 25};
int B[] = {50, 40, 30, 20, 10};
vector<int> v(10, 0);
vector<int>::iterator it;
sort(A, A + 5);
sort(B, B + 5);
merge(A, A + 5, B, B + 5, v.begin());
cout << "The resulting vector contains:";
for(it = v.begin(); it != v.end(); ++it) cout << " " << *it;
cout << endl;
```

SORTINGS

BUBBLE SORT

```
void sort(int v[], int n)
{
  bool sorted;
  for(int i = 0; i < n - 1; i++)
  {
    sorted = true;
    for(int k = 0; k < n - (i + 1); k++)
      if(v[k] > v[k + 1])
      {
        swap(v[j], v[j + 1]);
        sorted = false;
      }
    if(sorted) break;
  }
}
```

SELECTION SORT

```
void sort(int v[], int n)
{
  int h, menor;
  for(int i = 0; i < n - 1; i++)
  {
    menor = v[i]; h = i;
    for(int k = i + 1; k < n; k++)
      if(v[k] < menor)
      {
        menor = v[k]; h = k;
      }
    v[h] = v[i]; v[i] = menor;
  }
}
```

INSERTION SORT

```
void sort(int v[], int n)
{
  int aux, k;
  for(int i = 1; i < n; i++)
  {
    aux = v[i];
    k = i - 1;
    while(k >= 0 && aux < v[k])
    {
      v[k + 1] = v[k]; k--;
    }
    v[k + 1] = aux;
  }
}
```

COUNTING SORT

```
#define MAX 150

void sort(int v[], int n)
{
  int k = v[0], cop[MAX], frec[MAX]; // Variables auxiliares
  for(int i = 1; i < n; i++)
    k = max(k, v[i]);
  memset(aux, 0, sizeof aux);
  for(int i = 0; i < n; i++)
    frec[v[i]]++;
  for(int i = 1; i < k + 1; i++)
    frec[i] += frec[i - 1];
  for(int i = n; i >= 1; i--)
  {
    cop[frec[v[i - 1]] - 1] = v[i - 1];
    frec[v[i - 1]]--;
  }
  for(int i = 0; i < N; i++)
    v[i] = cop[i];
}
```

STL SORT

ARRAY

```
int v[10] = { 4, 12, 6, 78, 3, 0, 66, 74, 2, 14 };
sort(v, v + 10);
```

VECTOR

```
struct grupo
{
  int v;
  string s;
  grupo(int pv, const string &ps) : v(pv), s(ps) { }
};

bool operator < (const grupo &x, const grupo &y)
{
  if(x.v != y.v)
    return x.v < y.v; // Ascendente según val
  else
    return x.s < y.s; // Lexicograficamente
};
```

En el main:

```
vector<grupo> v;
sort(v.begin(), v.end());
```

PAIR

```
vector<pair<int,string> > v(10); // El operador < del pair es implicito
sort(v.begin(), v.end());
```

Tambien existe un sort estable:

```
stable_sort(vec.begin(), vec.end());
```

INDEX INVERSION COUTING

Cantidad de swaps de elementos consecutivos que se deben hacer para ordenar un vector ascedentemente.

ALGORITHM O(N^2)

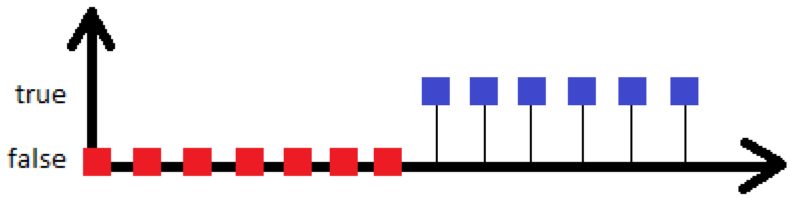
```
ll solve(const vector<int> &v)
{
   int n = (int)v.size();
   ll ans = 0;
   for(int i = 0; i < n - 1; i++)
      for(int j = i + 1; j < n; j++)
         if(v[i] > v[j])
            ans++;
   return ans;
}
```

ALGORITHM O(N LOG N)

```
ll solve(vector<int> &v)
{
  int n = (int)v.size(); // n: Number of elements to sort
  if(n < 2) return 0;    // There is nothing to swap
  // Split in two arrays
  int n1 = n/2, n2 = n - n/2;                 // Split in two arrays
  vector<int> l(v.begin(), v.begin() + n1); // L: 1st Array
  vector<int> r(v.begin() + n1, v.end());   // R: 2nd Array
  // Solve each array
  ll ans = solve(l) + solve(r);
  // Merge answers
  int i1 = 0, i2 = 0;
  while(i1 < n1 && i2 < n2) // While there are two arrays to merge
  {
    if(l[i1] <= r[i2]) {   v[i1 + i2] = l[i1]; ++i1; } // Minimum is at L
    else { ans += n1 - i1; v[i1 + i2] = r[i2]; ++i2; } // Minimum is at R
  }
  while(i1 < n1) { v[i1 + i2] = l[i1]; ++i1; } // Copy the rest of L
  while(i2 < n2) { v[i1 + i2] = r[i2]; ++i2; } // Copy the rest of R
  return ans;
}
```

BINARY SEARCH

DISCRETE BINARY SEARCH



LEAST X THAT MAKES F(X) = TRUE

```
int lo = MIN, hi = MAX;
while(lo < hi)
{
  int mid = lo + (hi-lo)/2; // Find middle element (ROUNDED DOWN)
  if(f(mid))       // If mid satisfies the property..
    hi = mid;      // Search in: [lo, mid], we are minimizing the value
  else             // Else if mid does not satisfy the property..
    lo = mid + 1;  // Search in: [mid+1, hi], we want a value that satisfies
}
// "lo" is now the first number x for which f(x) is true
```

GREATEST X THAT MAKES F(X) = FALSE

```
int lo = MIN, hi = MAX;
while(lo < hi)
{
  int mid = lo + (hi-lo+1)/2; // Find middle element (ROUNDED UP)
  if(f(mid))       // If mid satisfies the property..
    hi = mid - 1;  // Search in: [lo, mid-1], we want a value that ¬satisfies
  else             // If mid does not satisfy the property..
    lo = mid;      // Search in: [mid, hi], we are maximizing the value
}
// "lo" is now the last number x for which f(x) is false
```

CONTINOUS BINARY SEARCH

```
double lo = MIN, hi = MAX, ans;
for(int k = 0; k < 50; k++) // Fixed iterations to avoid an infinite loop..
{
  double mid = (lo + hi) * 0.5; // Find middle element
  if(f(mid))         // If mid is a valid answer..
    ans = hi = mid;  // Search in: [lo, mid]. Save the answer
  else               // If mid is not a valid answer..
    lo = mid;        // Search in: [mid, hi]
}
// "ans" is now the lowest valid answer that satisfies function f
```

STL

BINARY_SEARCH

Returns true if there is an element in range [*first*,*last*) equal to *value*.

```
cout << (binary_search(v.begin(), v.end(), 3)? "Found" : "Not found");
```

LOWER_BOUND

Returns an iterator that points to the first element of the range [*first*,*last*) that is **greater or equal** that *value*.

```
vector<int>::iterator low;                   // 10 10 10 20 20 20 30 30
low = lower_bound(v.begin(), v.end(), 20);   //          ^
cout << "lower_bound at pos << int(low - v.begin());
```

UPPER_BOUND

Returns an iterator that points to the first element of the range [*first*,*last*) that is **greater** that *value*.

```
vector<int>::iterator up;                   // 10 10 10 20 20 20 30 30
up = upper_bound(v.begin(), v.end(), 20); //                   ^
cout << "upper_bound at pos << int(up - v.begin());
```

TERNARY SEARCH

We will assume that the functions are *unimodal* (first strictly increasing, then strictly decreasing) and we want to find the max value. However, the code can be adapted to handle the opposite case (follow **highlighted** comments).

DISCRETE TERNARY SEARCH

```
int lo = MIN, hi = MAX;
while(lo < hi)
{
  int mid = (lo + hi) * 0.5;
  if(v[mid] > v[mid+1]) // If v[mid] is greater.. (Change to "<" to minimize)
    hi = mid;           // Search in [lo, mid]
  else                  // If v[mid+1] is greater..
    lo = mid + 1;       // Search in [mid+1, hi]
}
// "lo" is now the index of the array that has the max value
```

CONTINOUS TERNARY SEARCH

```
#define EPS 1e-7

double lo = -1000000, hi = 1000000;
while(lo + EPS < hi)
{
  double mid1 = (2*lo + hi) / 3;
  double mid2 = (lo + 2*hi) / 3;
  if(f(mid1) > f(mid2)) // If f(mid1) is greater..(Change to "<" to minimize)
    hi = mid2;          // Search in [lo, mid2]
  else                  // If f(mid2) is greater..
    lo = mid1;          // Search in [mid1, hi]
}
// "lo" is now the value that maximizes f(lo)
```

ON 2-VARIABLE FUNCTIONS

```
double best(double x) // Given X, search for Y that maximizes f(X,Y)
{
  double lo = -1000000, hi = 1000000;
  while(lo + EPS < hi)
  {
    double mid1 = (2*lo + hi) / 3;
    double mid2 = (lo + 2*hi) / 3;
    if(f(x,mid1) > f(x, mid2)) // Change to "<" to minimize)
      hi = mid2;  // Search in [lo, mid2]
    else          // If f(x,mid2) is greater..
      lo = mid1;  // Search in [mid1, hi]
  }
  return f(x,lo); // "lo" is now the value that maximizes f(X,lo)
}
```

```
double solve() // Finds X and Y that maximizes f(X,Y)
{
  double lo = -1000000, hi = 1000000;
  while(lo + EPS < hi)
  {
    double mid1 = (2*lo + hi) / 3;
    double mid2 = (lo + 2*hi) / 3;
    if(best(mid1) > best(mid2)) // (Change to "<" to minimize)
      hi = mid2; // Search in [lo, mid2]
    else
      lo = mid1; // Search in [mid1, hi]
  }
  return best(lo); // Now find the best value for "lo"
}
```

SQRT DECOMPOSITION

MO ALGORITHM – O(N * SQRT(N) * O(X))

```
const int MAX = 200005;              // Max number of queries
struct query { int i,l,r; } q[MAX]; // i: Query index, l: Left, r: Right
int m;        // m: Number of queries
int block;    // block: Number of buckets ~ sqrt(Number of elements in array)
ll ans[MAX]; // ans[i]: Answer for query with original index i

bool mo(const query &a, const query &b)
{
  if(a.l / block != b.l / block)
    return a.l / block < b.l / block;
  else
    return a.r < b.r;
}

vi v;       // v: Array of elements
int n;      // n: Number of elements
int answer; // answer: Global current answer
void add(int i) { // Add v[i] to "answer" in O(X) }
void remove(int i) { // Remove v[i] from "answer" in O(X) }
```

To process queries:

```
// Read array of elements and queries
for(int i = 0; i < n; i++)
  cin >> v[i];
for(int i = 0; i < m; i++)
{
  q[i].i = i;
  cin >> q[i].l >> q[i].r;
}
// Calculate size of block and sort queries using new order
block = sqrt(n);
sort(q, q + m, mo);
// Process the queries linearly following new order
int l = 0, r = 0; answer = 0;
for(int i = 0; i < m; i++)
{
  while(l < q[i].l)
    remove(l++);
  while(l > q[i].l)
    add(--l);
  while(r <= q[i].r)
    add(r++);
  while(r > q[i].r + 1)
    remove(--r);
  ans[q[i].i] = answer;
}
// Print answers
for(int i = 0; i < m; i++)
  cout << ans[i] << '\n';
```

BACKTRACKING

TOWER OF HANOI

```
stack<int> t[3]; // Each tower will be represented by a stack

void solve(int n, int a, int c, int b) // a: Source, c: Goal, b: Middle
{
  if(n == 1)                // If there is only one left disc..
  {
    t[c].push(t[a].top());  // Move the disc from source to goal
    t[a].pop();             // Remove the disc from source
    return;
  }
  solve(n-1, a, b, c); // Move n-1 discs from source to middle
  solve(  1, a, c, b); // Move   1 disc  from source to goal
  solve(n-1, b, c, a); // Move n-1 discs from middle to goal
}

void init(int n) // Initializes the towers
{
  while(!t[0].empty()) t[0].pop(); // Cleans 1st tower
  while(!t[1].empty()) t[1].pop(); // Cleans 2nd tower
  while(!t[2].empty()) t[2].pop(); // Cleans 3rd tower
  for(int i = n; i > 0; i--)
    t[0].push(i);
}
```

Example:

```
init(n);            // Initialize first tower with n discs
solve(n, 0, 2, 1); // Solve for n discs
```

Important: When moving a tower of size *n*, the largest disc will never go to the middle stack.

DYNAMIC PROGRAMMING

LONGEST INCREASING SUBSEQUENCE – LIS

O(N^2)

Dado el arreglo: { -7, 10, 9, 2, 3, 8, 8, 1 }, su LIS es: { -7, 2, 3, 8 }

idx	0	1	2	3	4	5	6	7
v	-7	10	9	2	3	8	8	1
LIS[i]	1	2	2	2	3	4	4	2
prev[i]	-1	0	0	0	3	4	4	0

- v: Vector de números
- LIS[i]: Tamaño del LIS que se puede lograr desde la posición 0 hasta la posición i
- prev[i]: Posición previa por la que pasé para llegar a la posición i (el -1 indica inicio del LIS)

```
vi getLIS(const vi &v)
{
  int n = (int)v.size();
  vi prev(n), LIS(n), ans;
  /// Armar vector LIS y prev
  for(int i = 0; i < n; i++) // Para cada elemento..
  {
    LIS[i] = 1; prev[i] = -1;  // Asumir que es el inicio del LIS
    for(int j = 0; j < i; j++) // Recorremos todos los anteriores..
      if(v[j] < v[i] && LIS[j] + 1 > LIS[i]) // Se puede colocar despues de j
      {
        LIS[i] = LIS[j] + 1; prev[i] = j;
      }
  }
  /// Hallar la subsequencia
  int mayor = LIS[0], pos = 0;
  for(int i = 1; i < n; i++) // Buscamos la posicion con mayor tamaño
    if(LIS[i] > mayor)
    {
      mayor = LIS[i]; pos = i;
    }
  while(pos != -1) { ans.push_back(v[pos]); pos = prev[pos]; } // Retroceder
  reverse(ans.begin(),ans.end());
  return res;
}
```

En el main:

```
vi v; // Secuencia original, añadir los elementos
vi ans = getLIS(v);
printf("Longitud del LIS: %d\n", (int)ans.size());
for(int i = 0; i < (int)ans.size(); i++) printf("%d\n",ans[i]);
```

ALGORITMO O(N LOG N)

idx	0	1	2	3	4	5	6	7
v	-7	10	9	2	3	8	8	1
LIS[i]	1	2	2	2	3	4	4	2
ans[i]	0	3	4	5	-	-	-	-

- v: Vector de números
- LIS[i]: Tamaño del LIS que se puede lograr desde la posición 0 hasta la posición i
- ans: Contiene los indices de un increasing subsequence válido.

```
vector<int> getLIS(const vector<int> &v) // v no debe estar vacio
{
   int n = (int)v.size(), lo, hi, mid;
   vector<int> prev(n), LIS(n), ans;
   LIS[0] = 1; ans.push_back(0); // Asumir que el elemento 0 es parte del LIS
   for(int i = 1; i < n; i++) // Para cada elemento i ...
   {
      if(v[ans.back()] < v[i]) // Puedo colocarlo a la derecha del actual
      {
         LIS[i] = LIS[ans.back()] + 1;
         prev[i] = ans.back(); ans.push_back(i);
         continue; // Pasar a la siguiente iteracion
      }
      lo = 0; hi = (int)ans.size() - 1;
      while(lo < hi) // Busqueda binaria
      {
         mid = (lo + hi) * 0.5;
         if(v[ans[mid]] < v[i]) lo = mid + 1; // Buscar en [mid+1, hi]
         else hi = mid; // Buscar en [lo, mid]
      }
      if(v[i] < v[ans[lo]]) // v[i] debe reemplazar a v[lo] en la respuesta
      {
         if(lo > 0) prev[i] = ans[lo - 1]; // El prev[i] sera el izq de lo
         LIS[i] = (lo == 0)? 1 : (LIS[prev[i]] + 1);
         ans[lo] = i; // El elemento i reemplaza al elemento lo
      }
      else if(v[i] == v[ans[lo]]) LIS[i] = LIS[ans[lo]]; // Actualizar LIS[i]
   }
   for(int sz = (int)ans.size(), pos = ans.back(); sz--; pos = prev[pos])
      ans[sz] = v[pos]; // Reemplazamos los indices por los valores numericos
   return ans;
}
```

En el main:

```
vector<int> v; // Secuencia original, añadir los elementos
vector<int> ans = getLIS(v);
printf("Longitud del LIS: %d\n", (int)ans.size());
for(int i = 0; i < (int)res.size(); i++) printf("%d\n",ans[i]);
```

ALGORITMO O(N LOG N)

Código más compacto que da la respuesta sin armar el vector LIS.

```
vector<int> getLIS(const vector<int> &v) // v no debe estar vacio
{
   int n = (int)v.size(), lo, hi, mid;
   vector<int> prev(n), ans;
   ans.push_back(0); // Asumir que el elemento 0 es parte del LIS
   for(int i = 1; i < n; i++) // Para cada elemento i ...
   {
      if(v[ans.back()] < v[i]) // Puedo colocarlo a la derecha del actual
      {
         prev[i] = ans.back(); ans.push_back(i);
         continue; // Pasar a la siguiente iteracion
      }
      lo = 0; hi = (int)ans.size() - 1;
      while(lo < hi) // Busqueda binaria
      {
         mid = (lo + hi) * 0.5;
         if(v[ans[mid]] < v[i])
            lo = mid + 1; // Buscar en [mid+1,hi]
         else
            hi = mid;     // Buscar en [lo, mid]
      }
      if(v[i] < v[ans[lo]]) // v[i] debe reemplazar a v[lo] en la respuesta
      {
         if(lo > 0) prev[i] = ans[lo - 1]; // El prev[i] sera el izq de lo
         ans[lo] = i; // El elemento i reemplaza al elemento lo
      }
   }
   for(int sz = (int)ans.size(), pos = ans.back(); sz--; pos = prev[pos])
      ans[sz] = v[pos]; // Reemplazamos los indices por los valores numericos
   return ans;
}
```

En el main:

```
vector<int> v; // Secuencia original, añadir los elementos
vector<int> ans = getLIS(v);
printf("Longitud del LIS: %d\n", (int)ans.size());
for(int i = 0; i < (int)ans.size(); i++) printf("%d\n",ans[i]);
```

HEAVIEST INCREASING SUBSEQUENCE – HIS

Sea: v = [[a_1, w_1], [a_2, w_2], ..., [a_N, w_N]]. Se desea encontrar una subsecuencia que sea **ascendente en *a*** y que **maximize la sumatoria de *w***. El algoritmo devuelve la sumatoria de pesos de la subsecuencia óptima.

ALGORITMO O(N LOG N)

```
#define MAX 200005
typedef pair<int, int> ii;
```

```
int HIS(int v[][2], int N)
{
   set<ii> st; st.insert(ii(0, 0));
   set<ii>::iterator it;
   vector<ii>::iterator vit;
   for(int i = 0; i < N; i++)
   {
      it = st.lower_bound(ii(v[i][0], 0)); it--;
      ii nuevo(v[i][0], it->second + v[i][1]); it++;
      bool valid = true;
      vector<ii> erase_list;
      while(it != st.end())
         if(it->first == v[i][0] && it->second >= nuevo.second)
         {
            valid = false; break;
         }
         else if(it->second <= nuevo.second)
         {
            erase_list.push_back(*it); it++;
         }
         else break;
      for(vit = erase_list.begin(); vit != erase_list.end(); vit++)
         st.erase(*vit);
      if(valid) st.insert(nuevo);
   }
   return st.rbegin()->second; // peso total del HIS
}
```

Para mostrar el resultado:

```
int N, v[MAX][2];
scanf("%d", &N); // N: Numero de elementos
for(int i = 0; i < N; i++) scanf("%d", &v[i][0]); // index
for(int i = 0; i < N; i++) scanf("%d", &v[i][1]); // weight
printf("Max weight: %d\n", HIS(v, N));
```

LONGEST COMMON INCREASING SUBSEQUENCE – LCIS

Dadas dos secuencias de números A y B. Se pide hallar la subsecuencia ascendente **común** más larga.

HALLAR TAMAÑO DEL LCIS – O(N * M)

```
int LCIS_size(const vector<int> &A, const vector<int> &B)
{
   int N = (int)A.size(), M = (int)B.size(), len = 0;
   vector<int> C(M, 0); // C[i]: Tamaño del LCIS en la columna i
   for(int i = 0; i < N; i++)
      for(int cur = 0, j = 0; j < M; j++) // cur: Mayor tamaño hasta ahora
         if(A[i] == B[j] && cur + 1 > C[j]) C[j] = cur + 1;
         else if(C[j] > cur && B[j] < A[i]) cur = C[j];
   for(int i = 0; i < M; i++) // Buscamos la columna con mayor valor
      len = max(len, C[i]);
   return len;
}
```

RECUPERAR LCIS – O(N * M)

```
vector<int> LCIS(const vector<int> &A, const vector<int> &B)
{
   int N = (int)A.size(), M = (int)B.size(), len = 0, idx = -1;
   vector<int> C(M, 0), ans; // C[i]: Tamaño del LCIS en la columna i
   vector<vector<pair<int,int> > > posible(M);
   for(int i = 0; i < N; i++)
      for(int cur = 0, last = -1, j = 0; j < M; j++)
         if(A[i] == B[j])
         {
            posible[j].push_back(make_pair(cur + 1, last)); // Nuevo candidato
            if(cur + 1 > C[j]) C[j] = cur + 1;
         }
         else if(B[j] < A[i] && C[j] > cur) { cur = C[j]; last = j; }
   for(int i = 0; i < M; i++) // Buscamos la columna con mayor valor
      if(C[i] > len) { len = C[i]; idx = i; }
   while(len) // Armar subsequencia de respuesta
   {
      ans.push_back(B[idx]);
      for(int i = 0; i < (int)posible[idx].size(); i++) // Ver candidatos
         if(posible[idx][i].first == len)
         {
            idx = posible[idx][i].second;
            break;
         }
      len--;
   }
   reverse(ans.begin(),ans.end());
   return res;
}
```

FIBONACCI

```
int fib[MAXN + 1]; // Memoization

int calcFibonacci()
{
  fib[1] = fib[2] = 1;
  for(int i = 3; i <= MAXN; i++) fib[i] = fib[i - 1] + fib[i - 2];
}
```

KNAPSACK 1-0

```
// N: Cantidad de items, M: Maxima capacidad de la mochila
// W[i]: Pesos del item i, V[i]: Valor del item i
// C[i][w]: Mejor ganancia con los i primeros items en la mochila de tam. w
for(int i = 0; i <= N; i++) C[i][0] = 0;
for(int w = 0; w <= M; w++) C[0][w] = 0;
// DP
for(int i = 1; i <= N; i++) // Para cada item i
  for(int w = 1; w <= M; w++) // Para cada mochila de capacidad w
     if(W[i] > w) C[i][w] = C[i - 1][w]; // Este item no entra en la mochila
     else C[i][w] = max(C[i - 1][w] , C[i - 1][w - W[i]] + V[i]);
```

SUBSET SUM

Dado un arreglo de enteros V (cuya sumatoria es *suma)*, verificar si existe un subconjunto que sume K.

```
int dp[MAX];
void subsetSum(int v[], int n, int suma)
{
   memset(dp, 0, sizeof dp); dp[0] = 1;
   for(int i = 0; i < n; i++) // Para cada elemento i
      for(int j = suma; j >= v[i]; j--)
         dp[j] |= dp[j - v[i]]; // Si podemos llegar a j - v[i], tambien a j
}
printf("%s\n", dp[K]? "YES" : "NO"); // dp[i] es true si hay subconjunto
```

CAMBIO DE MONEDAS

MINIMA CANTIDAD DE MONEDAS PARA OBTENER UN VALOR DETERMINADO

change[*x*] es la minima cantidad de monedas necesarias para cambiar un valor *x*.

```
#define MAX 10000
#define INF 20000000
int change[MAX + 1], N = 5; // N: Cantidad de monedas
int monedas[5] = {50, 25, 10, 5, 1}; // monedas[i]: Valor de moneda i

void makeChange()
{
   memset(change, 0, sizeof change);
   for(int i = 1; i <= MAX; i++) // Para cada valor i a cambiar
   {
      change[i] = INF; // INF indica que no es posible llegar al valor i
      for(int j = 0; j < N; j++) // Para cada moneda j que sea <= a i
         if(monedas[j] <= i && 1 + change[i - monedas[j]] < change[i])
            change[i] = 1 + change[i - monedas[j]];
   }
}
```

CANTIDAD DE FORMAS DE OBTENER UN VALOR DETERMINADO

nways[*x*] es la cantidad de formas de obtener valor *x* con las monedas dadas.

```
#define MAX 10000
long long nways[MAX + 1];
int N = 5, monedas[5] = {50, 25, 10, 5, 1}; // N: Cantidad de monedas

void countWays()
{
   memset(nways, 0, sizeof nways); nways[0] = 1;
   for(int i = 0; i < N; i++) // Para cada moneda
      for(int j = monedas[i]; j <= MAX; j++)
         nways[j] += nways[j - monedas[i]];
}
```

MINIMA CANTIDAD DE MONEDAS PARA OBTENER UN VALOR USANDO SOLO 1 DE CADA TIPO

change[*x*] es la minima cantidad de monedas necesarias para cambiar un valor *x*.

```
#define MAX 10005
#define INF 20000000
int change[MAX + 1]; // INF indica que no es posible llegar al valor i
int N = 5, monedas[5] = {50, 25, 10, 5, 1}; // N: Cantidad de monedas

void makeChange()
{
   memset(change, 0, sizeof change);
   for(int i = 1; i <= MAX; i++) change[i] = INF;
   for(int i = 0; i < N; i++)
      for(int j = MAX; j >= 0; j--)
         if(j + monedas[i] < MAX && 1 + change[j] < change[j + monedas[i]])
            change[j + monedas[i]] = 1 + change[j];
}
```

MAXIMUM SUM

KADANE'S ALGORITHM 1D - O(N)

```
// x: Posicion de inicio, y: Posicion de fin
int maxSum1D(int v[], int n, int &x, int &y)
{
   int sum = 0, ki = 0, maxsum = -1; x = y = -1;
   for(int i = 0; i < n; i++)
   {
      sum = sum + v[i];
      if(sum > maxsum) { maxsum = sum; x = ki; y = i; } // [ki, y]
      if(sum < 0) { sum = 0; ki = i + 1; } // Si es negativo, actualizar ki
   }
   return maxsum;
}
```

KADANE'S ALGORITHM 2D - O(N^3)

```
#define MAXR 105
#define MAXC 105

int fx1, fy1, fx2, fy2; //(fx1,fy1): Posicion inicio, (fx2,fy2): Posicion fin

int maxSum2D(int v[][MAXC], int R, int C)
{
    int tmp[MAXC], y1, y2, cur, maxsum = -1; fx1 = fx2 = fy1 = fy2 = -1;
    for(int i = 0; i < R; i++)
    {
       memset(tmp, 0, sizeof tmp); // tmp[i]: Suma acumulada de la columna i
       for(int j = i; j < R; j++)  // Para cada j en el rango [i, R]..
       {
          for(int k = 0; k < C; k++) tmp[k] += v[j][k]; // Acumular fila j
          // El acumulado abarca las filas [i, j]
```

```
            cur = maxSum1D(tmp, C, y1, y2); // Maxsum1D en el acumulado
            if(cur > maxsum)
            {
               fx1 = i; fy1 = y1;
               fx2 = j; fy2 = y2;
               maxsum = cur;
            }
         }
    }
    return maxsum;
}
```

KADANE'S ALGORITHM 3D - O(N^4)

```
#define MAX 25

int maxSum3D(ll v[][MAX][MAX], int R, int C, int H)
{
   int tmp[MAX][MAX], cur, maxsum = -1;
   for(int h = 0; h < H; h++)
   {
      memset(tmp, 0, sizeof tmp);
      for(int k = h; k < H; k++) // Para cada h en el rango [h, H]..
      {
         for(int i = 0; i < R; i++)
            for(int j = 0; j < C; j++)
               tmp[i][j] += v[i][j][k]; // Acumular el nivel k
         // El acumulado abarca el bloque desde la altura h hasta altura k
         cur = maxSum2D(tmp, R, C); // Maxsum2D en el acumulado
         if(cur > maxsum) maxsum = cur;
      }
   }
   return maxsum;
}
```

EN UNA MATRIZ: SUMAS ACUMULADAS - O(N^4)

```
#define MAX 100
#define INF 200000000

int maxSum2D(int v[][MAX], int n) // v: matriz de sumas acumuladas
{
   int maxSubRect = -(INF - 1);
   for(int i = 0; i < n; i++)
      for(int j = 0; j < n; j++)
         for(int k = i; k < n; k++)
            for(int l = j; l < n; l++) {
               int subRect = v[k][l];
               if(i > 0) subRect -= v[i - 1][l];
               if(j > 0) subRect -= v[k][j - 1];
               if(i > 0 && j > 0) subRect += v[i - 1][j - 1];
               if(maxSubRect < subRect) maxSubRect = subRect;
            }
   return maxSubRect;
```

```
}
```

En el main:

```
int v[MAX][MAX], n;
for(int i = 0; i < n ; i++)
   for(int j = 0; j < n ; j++)
   {
      scanf("%d",&v[i][j]);
      if(i > 0) v[i][j] += v[i - 1][j];
      if(j > 0) v[i][j] += v[i][j - 1];
      if(i > 0 && j > 0) v[i][j] -= v[i - 1][j - 1];
   }
printf("%d\n", maximumSum2D(v, n));
```

TRAVELING SALESMAN PROBLEM (TSP)

USANDO BITMASK – O(2^N * N)

Hallar el menor costo de realizar un circuito que pase por cada nodo una sola vez.

```
int dp[MAX][1 << MAX], N; // N: Cantidad de nodos
int dist[MAX][MAX]; // dist[i][j]: costo de ir de i a j (-1 si no hay ruta)

int tsp(int pos, int mask) // pos: Nodo actual, mask: Estado de nodos
{
   if(mask == (1 << N) - 1) return dist[pos][0]; // Terminado
   if(dp[pos][mask] != -1) return dp[pos][mask]; // Ya calculado
   int ans = INF;
   for(int nxt = 0; nxt < N; nxt++) // Probar ir al resto de ciudades
      if(nxt != pos && !(mask & (1 << nxt)) && dist[pos][nxt] != -1)
         ans = min(ans, dist[pos][nxt] + tsp(nxt, mask | (1 << nxt)));
   return dp[pos][mask] = ans;
}
```

En el main:

```
memset(dp, -1, sizeof dp);
printf("El costo de hacer el circuito es: %d\n",tsp(0, 1));
```

RECONSTRUIR LA RUTA

```
int descendant[MAX][1 << MAX];
int dp[MAX][1 << MAX], N; // N: Cantidad de nodos
int dist[MAX][MAX]; // dist[i][j]: costo de ir de i a j (-1 si no hay ruta)

int tsp(int pos, int mask) // pos: Nodo actual, mask: Estado de nodos
{
   if(mask == (1 << N) - 1) return dist[pos][0]; // Terminado
   if(dp[pos][mask] != -1) return dp[pos][mask]; // Ya calculado
   int ans = INF;
   for(int nxt = 0; nxt < N; nxt++) // Probar ir al resto de ciudades
      if(nxt != pos && !(mask & (1 << nxt)) && dist[pos][nxt] != -1)
```

```
      {
         int val = dist[pos][nxt] + tsp(nxt, mask | (1 << nxt));
         if(val < ans) { ans = val; descendant[pos][mask] = nxt; }
      }
   return dp[pos][mask] = ans;
}

void printRoute(int pos, int mask) // Usamos los descendientes almacenados
{
   if(descendant[pos][mask] == -1) return;
   int nxt = descendant[pos][mask];
   printf("Go from %d to %d with cost %d\n", pos, nxt, dist[pos][nxt]);
   printRoute(nxt, mask | (1 << nxt));
}
```

En el main:

```
memset(dp, -1, sizeof dp);
printf("El costo de hacer el circuito es: %d\n", tsp(0,1));
printf("El circuito es:\n"); printRoute(0,1);
```

BITONIC TSP – O(N^2)

```
int dp[MAX][MAX]; // dp[i][j]: Menor dist. de las rutas, la 1ra acaba en i,
                  // la 2da acaba en j (abarca todos los numeros <= max(i,j))
double dist[MAX][MAX]; // dist[i][j]: costo de ir de i a j

double bitonicTSP(int N) // N: Cantidad de nodos
{
   if(N <= 1) return 0.0; // Caso trivial
   // Limpiar matriz DP
   for(int i = 0; i < N; i++)
      for(int j = 0; j < N; j++)
         dp[i][j] = INF;
   dp[0][0] = 0;
   // Procesar
   double ans = INF;
   for(int j = 0; j < N; j++)
      for(int i = max(0, j - 1); i >= 0; i--)
      {
         if(j == N - 1) ans = min(ans, dp[i][j] + dist[i][j]);
         else
         {
            // Colocar el nodo j + 1 en la 2da ruta (extender j)
            dp[i][j + 1] = min(dp[i][j + 1], dp[i][j] + dist[j][j + 1]);
            // Colocar el nodo j + 1 en la 1ra ruta (extender i)
            dp[j][j + 1] = min(dp[j][j + 1], dp[i][j] + dist[i][j + 1]);
         }
      }
   return ans;
}
```

INTEGER PARTITION

Hallar la cantidad de formas de alcanzar un número X usando Y sumandos.

CONSIDERANDO QUE (A + B) ES DIFERENTE QUE (B + A)

```
int dp[MAX][MAX]; // dp[i][j]: formas de alcanzar el numero i con j sumandos

void integerPartition()
{
   memset(dp, 0, sizeof dp);
   for(int i = 1; i < MAX; i++) { dp[0][i] = dp[i][1] = 1; }
   for(int j = 2; j < MAX; j++)
      for(int i = 1; i < MAX; i++)
         dp[i][j] = dp[i][j - 1] + dp[i - 1][j];
}
```

La respuesta se encontraría en: dp[numero a buscar][numero de sumandos]

CONSIDERANDO QUE (A + B) == (B + A)

```
int dp[MAX][MAX]; // dp[i][j]: formas de alcanzar el numero i con j sumandos

void integerPartition()
{
   memset(dp, 0, sizeof dp); dp[0][0] = 1;
   for(int i = 1; i < MAX; i++)
   {
      dp[i][1] = 1;
      for(int j = 2; j <= i; j++) dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];
   }
}
```

La respuesta se encontraría en: dp[numero a buscar][numero de sumandos]

LARGEST STACK

Cada caja tiene una capacidad de soporte C y un peso W. Hallar la cantidad máxima de cajas que se puede poner una sobre otra sin que ninguna exceda su capacidad de soporte

```
#define INF 2000000000

struct grupo
{
   int w, c; // w: weight, c: capacity
   grupo(){}
   grupo(int pw, int pc) { w = pw; c = pc; }
};

bool operator < (grupo a, grupo b) { return a.w + a.c < b.w + b.c; }

int largestStack(vector<grupo> v)
{
   int n = (int)v.size(), mayor = 0;
   sort(v.begin(), v.end());
```

```
   vector<int> best(n + 1, INF); best[0] = 0;
   // best[i]: menor peso que se forma con i cajas
   for(int i = 0; i < n; i++) // Poniendo la caja i en la base...
      for(int j = mayor + 1; j > 0; j--) // Formamos una pila de tamaño j
         if(v[i].c >= best[j - 1] && best[j - 1] + v[i].w < best[j])
         {
            best[j] = best[j - 1] + v[i].w;
            mayor = max(mayor, j);
         }
   return mayor;
}
```

MATRIX CHAIN MULTIPLICATION

```
#define MAX 15
#define INF (1LL << 32)
typedef long long ll;

ll dp[MAX][MAX];
int R[MAX][MAX];

void matrixChain(const vector<int> &v)
{
   int N = (int)v.size() - 1; // N: Cantidad de matrices
   for(int i = 0; i <= N; i++) dp[i][i] = 0; // Trivial: Solo 1 matriz
   for(int l = 2; l <= N; l++) // Por cada chain-length [2, N]...
      for(int i = 1; i <= N - l + 1; i++) // Considerar cada inicio...
      {
         int j = i + l - 1; // Calculamos el final usando la longitud l
         dp[i][j] = INF;    // Al inicio este valor es INF
         for(int k = i; k <= j - 1; k++) // Intentar cada punto de corte...
         {
            ll val = dp[i][k] + dp[k + 1][j] + v[i - 1] * v[k] * v[j];
            if(val < dp[i][j])
            {
               dp[i][j] = val;
               R[i][j] = k;
            }
         }
      }
}

void print(int i, int j) // Llamar print(1, N)
{
   if(i == j) printf("A%d",i);
   else
   {
      printf("(");
      print(i, R[i][j]); printf(" x "); print(R[i][j] + 1, j);
      printf(")");
   }
}
```

En el main, para leer el arreglo v:

```
vector<int> v;
int N; // N: Cantidad de matrices
// Leer tamanios de matrices
v.clear(); v.resize(N + 1);
for(int i = 0; i < N; i++) scanf("%d %d\n",&v[i],&v[i + 1]); // [A,B];[B,C]
```

GREEDY

ACTIVITY SELECTION

MAXIMIZE NUMBER OF SCHEDULED ACTIVITIES

The time of each activity is inclusive, ie. [start, end]

```
struct act
{
  int s,t; // s: start time, t: end time
  bool operator < (const act &a) const // First the ones that finish first
  {
    return (t != a.t)? (t < a.t) : (s > a.s);
  }
};

int solve(vector<act> &v)
{
  sort(v.begin(), v.end()); // Sort the activities
  int ans = 1, last = 0;    // Select activity with idx 0
  for(int i = 1; i < (int)v.size(); i++) // For the rest of the activities..
    if(v[i].s > v[last].t) // If activity i starts later than the last one..
    {
      ans++;    // Select activity i
      last = i; // Update the last selected activity
    }
  return ans;
}
```

MATH

SPECIAL NUMBERS

CATALAN NUMBERS

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452

CARMICHAEL NUMBERS

561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657, 52633, 62745, 63973

FACTORIAL NUMBERS

0!	1	7!	5,040	14!	87,178,291,200
1!	1	8!	40,320	15!	1,307,674,368,000
2!	2	9!	362,880	16!	20,922,789,888,000
3!	6	10!	3,628,800	17!	355,687,428,096,000
4!	24	11!	39,916,800	18!	6,402,373,705,728,000
5!	120	12!	479,001,600	19!	121,645,100,408,832,000
6!	720	13!	6,227,020,800	20!	2,432,902,008,176,640,000

PRIME NUMBERS (FIRST 340)

2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113	127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229	233	239	241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337	347	349	353	359	367	373	379	383	389	397	401	409
419	421	431	433	439	443	449	457	461	463	467	479	487	491	499	503	509	521	523	541
547	557	563	569	571	577	587	593	599	601	607	613	617	619	631	641	643	647	653	659
661	673	677	683	691	701	709	719	727	733	739	743	751	757	761	769	773	787	797	809
811	821	823	827	829	839	853	857	859	863	877	881	883	887	907	911	919	929	937	941
947	953	967	971	977	983	991	997	1009	1013	1019	1021	1031	1033	1039	1049	1051	1061	1063	1069
1087	1091	1093	1097	1103	1109	1117	1123	1129	1151	1153	1163	1171	1181	1187	1193	1201	1213	1217	1223
1229	1231	1237	1249	1259	1277	1279	1283	1289	1291	1297	1301	1303	1307	1319	1321	1327	1361	1367	1373
1381	1399	1409	1423	1427	1429	1433	1439	1447	1451	1453	1459	1471	1481	1483	1487	1489	1493	1499	1511
1523	1531	1543	1549	1553	1559	1567	1571	1579	1583	1597	1601	1607	1609	1613	1619	1621	1627	1637	1657
1663	1667	1669	1693	1697	1699	1709	1721	1723	1733	1741	1747	1753	1759	1777	1783	1787	1789	1801	1811
1823	1831	1847	1861	1867	1871	1873	1877	1879	1889	1901	1907	1913	1931	1933	1949	1951	1973	1979	1987
1993	1997	1999	2003	2011	2017	2027	2029	2039	2053	2063	2069	2081	2083	2087	2089	2099	2111	2113	2129
2131	2137	2141	2143	2153	2161	2179	2203	2207	2213	2221	2237	2239	2243	2251	2267	2269	2273	2281	2287

DIVISIBILITY CRITERIA

#	Criteria	Example
2	La última cifra es par (0 incluido)	378: porque la última cifra (8) es par
3	La suma de sus cifras es un múltiplo de 3	480: porque 4+ 8+ 0 = 12 es múltiplo de 3
4	El número formado por las dos últimas cifras es un múltiplo de 4 ó termina en doble cero	7324: porque 24 es múltiplo de 4 8200: porque termina en doble 00
5	La última cifra es 0 ó 5	485: porque acaba en 5
6	El número es divisible por 2 y por 3	24: Ver criterios anteriores
7	Al separar la última cifra de la derecha, multiplicarla por 2 y restarla de las cifras restantes la diferencia obtenida es igual a 0 ó es un múltiplo de 7	34349: Separamos el 9. Al multiplicarlo por 2 y restarlo de las cifras restantes tenemos 3416 (3434 - 18). Separamos el 6. Al multiplicarlo por 2 y restarlo de las cifras restantes tenemos 329 (341 - 12). Repetimos el proceso, 9*2=18, entonces 32-18=14; por lo tanto, 34349 es divisible entre 7 porque 14 es múltiplo de 7
8	Las tres últimas cifras forman un múltiplo de 8	27280: porque 280 es múltiplo de 8
9	La suma de sus cifras es múltiplo de 9	3744: porque 3+7+4+4=18 es múltiplo de 9
10	La última cifra es 0	470: porque termina en 0
11	Sumar las cifras en posición impar por un lado y las de posición par por otro. Luego restar el resultado de ambas sumas. Si el resultado es 0 ó un múltiplo de 11, el número es divisible por 11.	42702: impares 4+7+2=13, pares 2+0=2 diferencia 13-2=11, entonces 42702 es múltiplo de 11.
12	El número es divisible por 3 y 4	528: Ver criterios anteriores
13	Al separar la última cifra de la derecha, multiplicarla por 9 y restarla de las cifras restantes la diferencia es igual a 0 ó es un múltiplo de 13	3822: separamos el 2 y lo multiplicamos por 9, 2*9=18, entonces 382-18=364. Separamos el 4 y lo multiplicamos por 9, 4*9=36, entonces 36-36=0; por lo tanto, 3822 es divisible entre 13
14	El número es par y divisible entre 7	546: separamos el 6 y lo multiplicamos por 2, 6*2=12, entonces 54-12=42. 42 es múltiplo de 7 y 546 es par; por lo tanto, 546 es divisible entre 14
15	El número es divisible entre 3 y 5	225: termina en 5 y la suma de sus cifras es múltiplo de 3; por lo tanto, 225 es divisible entre 15
17	Al separar la última cifra de la derecha, multiplicarla por 5 y restarla de las cifras restantes la diferencia es igual a 0 ó es un múltiplo de 17	2142: porque 2*5=10, entonces 214-10=204, de nuevo, 4*5=20, entonces 20-20=0; por lo tanto, 2142 es divisible entre 17
18	El número es par y divisible por 9	9702: Es par y la suma de sus cifras (9+7+0+2=18) también es divisible entre 9.

SUMMATIONS

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Sumatoria de pares = 2 + 4+ 6 +... + n = n/2 * (n/2 + 1)
Sumatoria de impares = 1 + 3 + 5 + ... + n = ((n+1) / 2)^2

$$\sum_{i=m}^{n} i = \frac{n(n+1) - m(m-1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

$$\sum_{i=1}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

Progresión geométrica: $\sum_{i=1}^{n} a_i = a_1 \frac{r^n - 1}{r - 1}$

Progresión aritmética: $\sum_{i=1}^{n} a_i = n \frac{a_1 + a_n}{2}$

SERIES

Sucesión	Fórmula	Comentario
0, 1, 5, 13, 27, 48, 78, 118, 170, 235, 315, 411, 525	floor(n(n+2)(2n+1)/8)	Number of triangles in triangular matchstick arrangement of side n.
1, 0, 1, 2, 9, 44, 265, 1854, 14833, 133496, 1334961, 14684570, 176214841, 2290792932, 32071101049,...	$f(n) = n \times f(n-1) + (-1)^n$	Cantidad de permutaciones de n elementos sin punto fijos (el '1' no debe estar en la posición '1', el '2' no debe estar en la posición '2', etc)
0, 1, 3, 8, 21, 55, 144, 377, 987, 2584, 6765, 17711, 46368, ...	$f(n) = fib(2n)$	Bisección de fibonacci
6, 28, 496, 8128, 33550336, 8589869056, 137438691328, 2305843008139952128, ...	$f(p) = 2^{p-1}(2^p - 1)$ Solo cumple si p y $(2^p - 1)$ son primos.	Números perfectos

NUMBER THEORY

ASPECTOS BÁSICOS

- El número primo más grande que entra en una variable **int** es 2,147,483,647.
- **Teorema Pequeño de Fermat**: $a^p \equiv a (\mathrm{mod}\ p)$, si p es primo, y (a, p) son coprimos. También se expresa como: $a^{p-1} \equiv 1 (\mathrm{mod}\ p)$
- Si la factorización prima de un número es de la forma $p_1^{e1} p_2^{e2} \ldots p_k^{ek}$, entonces su cantidad de divisores será $(e_1+1) \times (e_2+1) \times \ldots \times (e_k+1)$

TEMPLATE

```
typedef long long ll;
```

SIEVE OF ERATOSTHENES

```
ll sieve_size;        // Tamaño de la criba
bitset<10000010> bs; // 10^7 + espacio extra
vector<ll> primes;   // Lista de números primos

void sieve(ll n) // n: Tamaño de la criba
{
  sieve_size = n;      // Guardar el tamaño de la criba
  bs.set();            // Marcar todos los números como primos..
  bs[0] = bs[1] = 0;   // Excepto el 0 y 1
  for(ll i = 2; i <= n; i++) // Para cada número..
    if(bs[i]) // Si está marcado como primo..
    {
      for(ll j = i * i; j <= n; j += i) // Para c/uno de sus múltiplos..
        bs[j] = 0;                      // Marcarlo como no primo
      primes.push_back(i); // Agregarlo a la lista de números primos
    }
}
```

Nota: Las funciones que dependan de la criba funcionarán para n <= (último primo en vector primes)^2

VERIFICAR SI UN NÚMERO ES PRIMO

```
bool isPrime(ll n) // n: Número a verificar
{
   if(n <= sieve_size) return bs.test(n); // O(1) para números en la criba
   for(int i = 0; i < (int)primes.size(); i++) // Para cada número primo..
   {
      ll p = primes[i];
      if(n % p == 0) return false; // Si el primo lo divide, no es primo
      if(p * p > n)  return true;  // Si excedimos la raiz de n, es primo
   }
   return true;
}
```

FACTORES PRIMOS

```
vector<ll> primeFactors(ll N)
{
   vector<ll> factors; // Lista de factores primos
   ll idx = 0, PF = primes[idx];
   while(N != 1 && PF * PF <= N) // Probar todos los primos <= sqrt(N)
   {
      while(N % PF == 0) { N /= PF; factors.push_back(PF); }
      PF = primes[++idx]; // Siguiente primo
   }
   if(N != 1) factors.push_back(N);
   return factors;
}
```

EXPONENTE DE UN PRIMO P EN LA FACTORIZACIÓN PRIMA DE N!

```
ll getPower(ll p, ll N) // Fórmula de Polignac. Exponente de p en N!
{
   ll ans = 0;
   for(ll power = p; power <= N; power *= p) ans += N / power;
   return ans;
}
```

NÚMERO DE FACTORES PRIMOS

```
ll numPF(ll N)
{
   ll idx = 0, PF = primes[idx], ans = 0;
   while(N != 1 && PF * PF <= N) // Probar todos los primos <= sqrt(N)
   {
      while(N % PF == 0) { N /= PF; ans++; }
      PF = primes[++idx]; // Siguiente primo
   }
   if(N != 1) ans++;
   return ans;
}
```

NÚMERO DE FACTORES PRIMOS *DIFERENTES*

```
ll numDiffPF(ll N)
{
   ll idx = 0, PF = primes[idx], ans = 0;
   while(N != 1 && PF * PF <= N) // Probar todos los primos <= sqrt(N)
   {
      if(N % PF == 0) ans++;
      while(N % PF == 0) N /= PF;
      PF = primes[++idx]; // Siguiente primo
   }
   if(N != 1) ans++;
   return ans;
}
```

SUMA DE FACTORES PRIMOS

```
ll sumPF(ll N)
{
   ll idx = 0, PF = primes[idx], ans = 0;
   while(N != 1 && PF * PF <= N) // Probar todos los primos <= sqrt(N)
   {
      while(N % PF == 0) { N /= PF; ans += PF; }
      PF = primes[++idx]; // Siguiente primo
   }
   if(N != 1) ans += N;
   return ans;
}
```

NÚMERO DE DIVISORES

```
ll numDiv(ll N)
{
   ll idx = 0, PF = primes[idx], ans = 1;
   while(N != 1 && PF * PF <= N) // Probar todos los primos <= sqrt(N)
   {
      ll power = 0;
      while(N % PF == 0) { N /= PF; power++; }
      ans *= (power + 1);
      PF = primes[++idx]; // Siguiente primo
   }
   if(N != 1) ans *= 2;
   return ans;
}
```

SUMA DE DIVISORES

```
ll sumDiv(ll N)
{
    ll idx = 0, PF = primes[idx], ans = 1;
    while(N != 1 && PF * PF <= N) // Probar todos los primos <= sqrt(N)
    {
        ll power = 0;
        while(N % PF == 0) { N /= PF; power++; }
        ans *= ((ll)pow((double)N, 2.0) - 1) / (N - 1);
        PF = primes[++idx]; // Siguiente primo
    }
    if (N != 1) ans *= ((ll)pow((double)N,2.0) - 1) / (N - 1);
    return ans;
}
```

GREATEST COMMON DIVISOR (GCD) / LOWEST COMMON MULTIPLE (LCM)

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

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

Non-recursive implementation:

```
int gcd(int a, int b)
{
  while(b != 0)
  {
    a = a % b;
    swap(a, b);
  }
  return a;
}
```

EULER PHI FUNCTION

ϕ(N): Cantidad de números enteros positivos menores o iguales que N son coprimos con N.

Si los factores primos de un número n son $p_1, p_2, \ldots, p_k$; se puede usar la siguiente fórmula:

$$\varphi(n) = n \times (1 - \frac{1}{p_1}) \times (1 - \frac{1}{p_2}) \times \ldots \times (1 - \frac{1}{p_k})$$

$\varphi(n)$	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0+		1	1	2	2	4	2	6	4	6
10+	4	10	4	12	6	8	8	16	6	18
20+	8	12	10	22	8	20	12	18	12	28
30+	8	30	16	20	16	24	12	36	18	24
40+	16	40	12	42	20	24	22	46	16	42
50+	20	32	24	52	18	40	24	36	28	58
60+	16	60	30	36	32	48	20	66	32	44
70+	24	70	24	72	36	40	36	60	24	78
80+	32	54	40	82	24	64	42	56	40	88
90+	24	72	44	60	46	72	32	96	42	60

Tabla con los 100 primeros números de la función ϕ.

Ejemplo:

ϕ(36) = 12.

Los 12 números son: 1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31 y 35

PARA UN NÚMERO

```
ll phi(ll n)
{
   vector<ll> factors = primeFactors(n); // Lista de factores primos de n
   vector<ll>::iterator new_end = unique(factors.begin(), factors.end());
   ll ans = n; // Aplicar la formula
   for(vector<ll>::iterator i = factors.begin(); i != new_end; i++)
      ans = ans - ans / *i;
   return ans;
}
```

SIEVE

Observaciones:

- Para n = 1: $\varphi(n) = n$
- Para el resto de números: $\varphi(n) < n$

```
vector<int> phi;

void sieve(int n) // n: Cantidad de números
{
   phi.assign(n + 1, 0); // Reservar espacio en el arreglo
   // Comenzar asumiendo que phi[i] = i para todos los números
   for(int i = 1; i <= n; i++)
      phi[i] = i;
   // Criba
   for(int i = 2; i <= n; i++) // Recorrer todos los números..
      if(phi[i] == i)          // Si phi(i) = i, entonces i es primo
         for(int j = i; j <= n; j += i) // Actualizar cada múltiplo de i..
            phi[j] = (phi[j] / i) * (i - 1); // Multiplicar por (1 - 1/i)
}
```

ARITMÉTICA MODULAR

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

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

$$(a \times b) \bmod n = ((a \bmod n) \times (b \bmod n)) \bmod n$$

EXPONENCIACION RÁPIDA (B^P % M)

```
ll square(ll a) { return a * a; }

ll modPow(ll b, ll p, ll m)
{
  if(p == 0)
    return 1; // b^0 = 1
  if(p % 2 == 0)
    return square(modPow(b, p / 2, m)) % m;      // modPow(b,p/2)^2
  else
    return ((b % m) * modPow(b, p - 1, m)) % m; // modPow(b,p-1)*b
}
```

MODULO DE NUMEROS NEGATIVOS

En C++, si tratamos de calcular "-5 % 2", el resultado será -1. Para solucionarlo, podemos usar:

```
int safeMod(int a, int m) { return ((a % m) + m) % m; }
```

TEOREMA DE EULER

Sean a y n números enteros positivos coprimos, es decir gcd(a, n) = 1, entonces se cumple:

$$a^{\varphi(n)} \equiv 1 \pmod n$$

Ejemplo: Hallar el último dígito de 7^{222}.

Piden hallar 7^{222} (mod 10). Se sabe que 7 y 10 son coprimos y que ϕ (10) = 4. El teorema de Euler dice que $7^4 \equiv 1$ (mod 10), así obtenemos que $7^{222} \equiv 7^{4x55+2} \equiv (7^4)^{55}x7^2 \equiv 1^{55}x7^2 \equiv 49 \equiv 9$ (mod 10).

EXTENDED EUCLID

Resuelve la ecuación: aX + bY = gcd(a,b). Es decir, dados a y b, halla un X e Y que cumplen la ecuación. Además, halla d = gcd(a, b)

```
typedef pair<int,int> ii;
typedef pair<int,ii> iii;

// Devuelve gcd(a,b) seguido del par (x,y) que cumple: ax + by = gcd(a,b)
iii egcd(int a, int b)
{
   if(b == 0) return iii(a, ii(1, 0)); // Caso base: Residuo 0
   iii p = egcd(b, a % b);             // Logica del algoritmo de Euclides
   int x = p.second.first, y = p.second.second;
   return iii(p.first, ii(y, x - (a / b) * y));
}
```

Es útil para resolver este problema: aX + bY = c. ¿Cómo? Le enviamos a y b, luego verificamos si c % d == 0. Si es así, podemos hallar un número z (z = c / d) que usaremos para multiplicar ambos términos de la ecuación.

a.X.z + b.Y.z = d.z

Con esto ya hallamos una solución (x_0 = X.z; y_0 = Y.z). Las demás soluciones se hallan así:

$x = x_0 + (b / d) . t$; $y= y_0 - (a / d) . t$; donde t = ..., -3, -2, -1, 0, 1, 2, 3, ...

ECUACIONES DIOFANTICAS

Hallar valores de X e Y (x,y >= 0), de manera que aX + bY = c; minimizando($X * c_1 + Y * c_2$)

Explicación de la solución:

```
De la sección anterior, sabemos que:
    x = x0 + (b / d) * t
    y = y0 - (a / d) * t
Como x >= 0 && y >= 0:
    x0 + (b / d) * t >= 0
    y0 - (a / d) * t >= 0
Despejando:      -c * x / b  <= t <= c * y / a
Rango de t: ceil(-c * x / b) <= t <= floor(c * y / a)
```

```
Para minimizar el costo:
   C(x, y) = c1 * x + c2 * y
   C(t)    = c1 * (x0 + (b / d) * t) + c2 * (y0 - (a / d) * t)
Basta con verificar los valores que tiene t en sus dos extremos
```

Algoritmo:

```
ii solve(int a, int b, int c, int c1, int c2)
{
   iii egcd = extendedEuclid(a, b); // Resolver con Euclides Extendido
   int x = egcd.second.first, y = egcd.second.second, d = egcd.first;

   if(c % d != 0) return ii(-1,-1); // No hay solucion
   // Buscar los dos extremos de t y hallar los valores (x,y)
   int t1 = ceil(-c * x / b), t2 = floor(c * y / a);
   int x1 = (x * c / d) + (b / d) * t1, y1 = (y * c / d) - (a / d) * t1;
   int x2 = (x * c / d) + (b / d) * t2, y2 = (y * c / d) - (a / d) * t2;
   // Validar si los numeros son positivos
   if((x1 < 0 || y1 < 0) && (x2 < 0 || y2 < 0)) return ii(-1,-1);
   // Nos quedamos con el par que minimize la segunda ecuacion
   if(x1 * c1 + y1 * c2 < x2 * c1 + y2 * c2) return ii(x1,y1);
   return ii(x2,y2);
}
```

INVERSO MODULAR

El inverso modular de un entero *a* modulo *m* es el entero *x* que cumple: $ax \equiv 1(mod\ m)$, también se representa como: $a^{-1} \equiv x(mod\ m)$

El número *x* solo existe si *a* y *m* son coprimos, es decir, si gcd(*a*, *m*) = 1

USANDO EUCLIDES EXTENDIDO – O(LOG(M)^2)

```
int inversoModular(int a, int m)
{
    return (extendedEuclid(a, m).second.first + m) % m;
}
```

USANDO EL TEOREMA DE EULER– O(LOG(M))

```
int inversoModular(int a, int m)
{
    return modPow(a, EulerPhi(m) - 1, m);
}
```

TEOREMA CHINO DEL RESTO

Sean las ecuaciones:

Hay un único *x* que satisface las ecuaciones dadas:

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

Donde: $n_1, n_2, \ldots, n_k$; son primos entre sí

$$x := \sum_i a_i \frac{N}{n_i} \left[\left(\frac{N}{n_i} \right)^{-1} \right]_{n_i}$$

Donde: $N = n_1 \times n_2 \times \ldots n_k$

```
int chineseRemainder(vector<int> a, vector<int> n, int k)
{
   int tmp, mod = 1, ans = 0;
   for(int i = 0; i < k; i++) mod *= n[i];
   // Aplicar la formula
   for(int i = 0; i < k; i++)
   {
      tmp = mod / n[i];
      tmp *= inversoModular(tmp, n[i]);
      ans += (tmp * a[i]) % mod;
   }
   return ans % mod;
}
```

CRIBAS MODIFICADAS

FACTORES PRIMOS

```
vector<int> numDiffPF;

// Halla la cantidad de Factores Primos diferentes
void sieveNumDiffPF(int n)
{
   numDiffPF.assign(n + 1, 0);
   for(int i = 2; i <= n; i++)
      if(numDiffPF[i] == 0)
         for(int j = i; j <= n; j += i)
            numDiffPF[j]++;
}
```

INVERSO MODULAR

```
vector<int> modInverse;

// Halla el inverso de los primeros n numeros modulo m
void sieveModInverse(int n, int m)
{
   modInverse.assign(n + 1, 0); modInverse[1] = 1;
   for(int i = 2; i <= n; i++)
      modInverse[i] = (-(m / i) * modInverse[m % i]) % m + m;
}
```

DISCRETE LOGARITHM

Teniendo la siguiente ecuación: $X^{Y}\%Z = K$, dados X, Z y K, hallar el valor de Y

BABY STEP – GIANT STEP O(SQRT(Z))

```
#define MAX 100000

struct hashtable
{
   int key[MAX], value[MAX];

   void init()
   {
      for(int i = 0; i < MAX; i++)
         key[i] = value[i] = -1;
   }

   void insert(int k, int v)
   {
      int kk = k % MAX;
      while(key[kk] != -1 && key[kk] != k) kk = (kk + 1) % N;
      key[kk] = k, value[kk] = v;
   }

   int find(int k)
   {
      int kk = k % MAX;
      while(key[kk] != -1 && key[kk] != k) kk = (kk + 1) % N;
      return value[kk];
   }
}h;

int babystep(int x, int k, int z) // (X^Y) mod Z = K, devuelve el valor de Y
{
   x %= z, k %= z;
   int m = (int)ceil(sqrt(1.0 * z)), xj = 1;
   h.init(); h.insert(k, 0);            // Insertar en hash table: (K, 0)
   for(int j = 1; j <= m; j++)          // Para todo j <= sqrt(z)
   {
      xj = (1ll * xj * x) % z;          // Calcular x^j
      h.insert((1ll * xj * k) % z, j); // Insertar en hashtable: (K * x^j, j)
   }
   for(int i = 0, xm = xj, xim = 1; i * m <= z; i++)
   {
      int j = h.find(xim);          // Buscar x^(i*m) en hash table
      if(j >= 0 && i * m - j > 0)   // Si esta en la tabla e Y es mayor que 0
         return i * m - j;
      else if(j >= 0 && i * m - j == 0)
      {
         if(k == 1) return 0;      // Si Y es 0, entonces K debe ser 1
         else return -1;           // Sino no hay solucion
      }
      xim = (1ll * xim * xm) % z; // x^((i+1)*m) = x^(i*m) * x^m
```

```
    }
    return -1; // No hay solucion
}
```

DIGIT COUNTING

Cuenta la cantidad de digitos para escribir los numeros en un rango [X, Y], donde 1 <= X <= Y <= N

ALGORITMO

```
#define MAX 12
#define DIG 10
typedef long long ll;

ll digitos[MAX], digzero[MAX];

void initDigit() // Inicia contadores
{
   digitos[0] = digzero[0] = 0;
   for(int i = 1, k = 1; i < MAX; i++, k *= 10)
   {
      digitos[i] = i * k;
      digzero[i] = 9 * digitos[i - 1] + digzero[i - 1];
   }
   //Si se incluye el rango [0 a N]:
   //digzero[1] = 1;
}

vector<ll> digitCount(ll A) // Query: [1, A]
{
   vector<ll> dig(DIG, 0);
   ll base10 = 1, n = 0;
   int x = 1;
   while(A > 0)
   {
      int d = A % 10;
      for(int i = 0; i < DIG; i++) // Contar por cada digito
      {
         if(i < d) dig[i] += (base10 + digitos[x - 1] * d);
         else if(i == d) dig[i] += ((n + 1) + digitos[x - 1] * d);
         else dig[i] += (digitos[x - 1] * d);
      }
      n += base10 * d; base10 *= 10; A /= 10; x++;
   }
   dig[0] -= digitos[x - 1]; dig[0] += digzero[x - 1];
   return dig;
}

vector<ll> digitCount(ll A, ll B) // Query [A, B]
{
   vector<ll> vecHelp = digitCount(A);
   vector<ll> vec = digitCount(B);
   for(int i = 0; i < DIG; i++) vec[i] -= vecHelp[i];
   while(A > 0) { vec[A % 10]++; A /= 10; }
```

```
    return vec;
}
```

SUMA DE DIGITOS

```
ll digitSum(ll A, ll B) // Query: [A, B]
{
    vector<ll> vec = digitCount(A, B);
    ll suma = 0;
    for(int i = 1; i < DIG; i++) suma += vec[i] * i;
    return suma;
}

ll digitSum(ll A) // Query: [1, A]
{
    return digitSum(1, A);
}
```

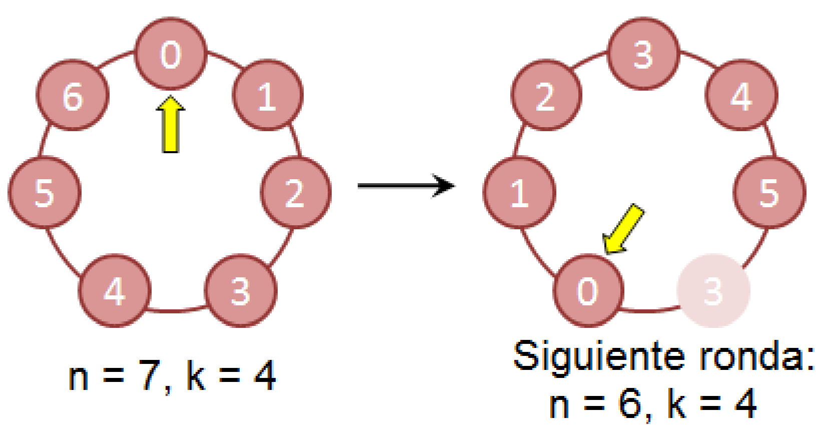
EJEMPLO

```
initDigit(); // Inicia los contadores
vector<ll> res = digitCount(A, B); // Cuenta digitos en un rango [A, B]
for(int i = 0; i < 10; i++) printf(" %lld",res[i]); // Imprime resultados
```

JOSEPHUS

ENCONTRAR AL SOBREVIVIENTE

Se colocan "n" personas en un círculo y se elimina a una cada k personas.



Se puede usar programación dinámica para encontrar al sobreviviente:

- Caso base: f(1, k) = 0;
- Caso general: f(n, k) = (f(n-1, k) + k) % n

```
int findSurvivor(int n, int k) // n personas, se elimina cada k personas
{
```

```
    int ans = 0;
    for(int i = 1; i <= n; i++) ans = (ans + k) % i;
    return ans; // (+1 si la numeración de las personas comenzara en 1)
}
```

SIMULACION

Usar treaps para hallar el orden en que los soldados seran ejecutados en O(N lg N)

```
// Colocar los N candidatos
for(int i = 0; i < N; i++) treap = insert(treap, new Node(i + 1));
// Simular
int pos = 0;
for(int i = 0; i < N; i++)
{
   pos += K - 1; pos %= N - i;      // Buscar posicion del siguiente K-esimo
   aux = find_kth(treap, pos + 1); // Buscar en el treap
   printf(" %d",aux->key);          // Imprimirlo
   treap = erase(treap, aux->key); // Sacarlo del treap
}
clear(treap);
treap = NULL;
```

NÚMEROS GRANDES – JAVA

FUNCIONES DENTRO DE LA CLASE BIGINTEGER

Se usa para el manejo de números grandes (tiene funciones de cambio de base). Las más importantes:

`BigInteger b = new BigInteger(myString, 16);`	Crea un BigInteger usando un string y la base. Ejemplo: ("1A",16) crea un BigInteger con valor 26. Por más que se le mande la base, la clase primero lo pasa a base 10 y lo guarda (por eso es 26)
`System.out.println(b.toString(16).toUpperCase());`	b.toString(16) imprime el número almacenado en b en base 16. Si b1=26, imprime "1a". toUpperCase es un método de la clase String que pasa todo a mayús.
`p.multiply(m)`	p y m son BigInteger. **Devuelve** un BigInteger con el resultado de p * m
`p.divide(m)`	p y m son BigInteger. **Devuelve** un BigInteger con el resultado de p / m (división entera)
`p.add(m)`	p y m son BigInteger. **Devuelve** un BigInteger con el resultado de p + m
`p.substract(m)`	p y m son BigInteger. **Devuelve** un BigInteger con el resultado de p –

	m
`p.mod(m)`	p y m son BigInteger. **Devuelve** un BigInteger con el resultado de p % m
`p.gcd(m)`	p y m son BigInteger. **Devuelve** un BigInteger con el resultado de el gcd de p y m
`BigInteger aux = sc.nextBigInteger();`	sc es un Scanner. Lee un BigInteger de la entrada.
`aux.compareTo(otroBigInteger)`	Compara aux con otro BigInteger. Si es 0, son iguales. Si es mayor que 0. Aux es mayor. Si es menor que 0. Aux es menor.

Ejemplo 1 (lo comentado es para hacer **debug**)

```
import java.io.*;
import java.util.*;
import java.math.*;

class Main{
   public static void main(String[] args){
      //BufferedInputStream in = null;
      //PrintStream out = null;
      //try{
      //   in = new BufferedInputStream(new FileInputStream("C:\\in.txt"));
      //   out = new PrintStream(new BufferedOutputStream(
      //                          new FileOutputStream("C:\\out.txt")));
      //}catch (FileNotFoundException ex){}
      //System.setIn(in); System.setOut(out); System.setErr(out);
      Scanner sc = new Scanner(System.in);
      while(sc.hasNext()) {
         BigInteger b1 = sc.nextBigInteger();
         BigInteger b2 = sc.nextBigInteger();
         System.out.println(b1.multiply(b2).toString());
      }
      //System.out.close();
   }
}
```

Ejemplo 2:

```
import java.io.*;
import java.util.*;
import java.math.*;

class Main {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      while(sc.hasNext()) {
         String aux = sc.next();
         if(aux.charAt(0) != '-') {
           if(aux.length() > 1 && aux.charAt(0)=='0' && aux.charAt(1)=='x') {
              BigInteger b1 = new BigInteger(aux.substring(2),16);
              System.out.println(b1.toString());
```

```
            }
            else {
               BigInteger b1 = new BigInteger(aux);
               System.out.println("0x".concat(b1.toString(16).toUpperCase()));
            }
          }
          else break;
        }
     }
}
```

NÚMEROS GRANDES – JAVA – FUNCIONES ÚTILES

RAIZ CUADRADA (ALGORITMO NEWTON – RAPHSON)

```
static boolean didWork; // didWork me permite saber si la raiz es exacta
public static BigInteger sqrt(BigInteger A) // Para BigInteger
{
   BigInteger temp, result = null;
   temp = A.shiftRight(BigInteger.valueOf(A.bitLength()).shiftRight(1).intValue());
   while(true) {
      result = temp.add(A.divide(temp)).shiftRight(1);
      if(!temp.equals(result)) temp = result;
      else break;
   }
   didWork = false;
   if(result.multiply(result).equals(A)) didWork = true ;
   return result;
}
```

Para BigDecimals

```
public static BigDecimal sqrt(BigDecimal x, int scale) // scale: precision
{
   BigInteger A = x.movePointRight(scale << 1).toBigInteger(), result = null;
   BigInteger temp;
   temp = A.shiftRight(BigInteger.valueOf(A.bitLength()).shiftRight(1).intValue());
   while(true) {
      result = temp.add(A.divide(temp)).shiftRight(1);
      if(!temp.equals(result)) temp = result;
      else break;
   }
   return new BigDecimal(result, scale);
}
```

BIGDECIMAL TO BINARY BASE

```
public static BigDecimal bigDecimaltoBinary(BigDecimal x, int scale)
{
   String numero = x.toString(); // Convertimos el Decimal en String
   String parteEntera = numero.substring(0, numero.indexOf('.'));
   String parteDecimal = "0" + numero.substring(numero.indexOf('.'));
   BigDecimal bgDecimal = new BigDecimal(parteDecimal); parteDecimal = "";
```

```
   for(int i = 0; i < scale; i++) {
      bgDecimal = bgDecimal.multiply(BigDecimal.valueOf(2)); // Por 2
      if(bgDecimal.compareTo(BigDecimal.ONE) > -1) // Si es >= 1
      {
         parteDecimal += "1"; bgDecimal = bgDecimal.subtract(BigDecimal.ONE);
      }
      else parteDecimal += "0";
   }
   String ans = (new BigInteger(parteEntera)).toString(2) +"."+ parteDecimal;
   return new BigDecimal(ans);
}
```

CALENDARIO GREGORIANO – JAVA

Java permite el manejo de fechas de calendario. Algunas de las funciones más importantes son:

`GregorianCalendar calendar = new GregorianCalendar();`	Crea un calendario
`calendar.set(year, month, day);`	Setea el calendario a una fecha determinada. **El campo month debe ser un entero entre 0 y 11.** (0: Enero, 11: Diciembre)
`calendar.add(GregorianCalendar.DATE, ndays);`	Añade cierta cantidad de días a la fecha seteada en el calendario.
`calendar.get(GregorianCalendar.MONTH)`	Obtiene el valor entero del campo especificado. Para los meses, devuelve un entero entre 0 y 11.

EJEMPLO DE USO

```
Scanner sc = new Scanner(System.in);
int N, day, month, year;
N = sc.nextInt(); day = sc.nextInt(); month = sc.nextInt(); year = sc.nextInt();
month -= 1; // RESTAR UNO AL MES!
GregorianCalendar calendar = new GregorianCalendar(); // Crear calendar
calendar.set(year, month, day); // Setear Calendar
calendar.add(GregorianCalendar.DATE, N); // Sumar N días a la fecha
int nuevomes = calendar.get(GregorianCalendar.MONTH) + 1;
// Imprimir en format DD/MM/AAAA
System.out.println(calendar.get(GregorianCalendar.DAY_OF_MONTH) + "/" +
                   nuevomes + "/" +
                   calendar.get(GregorianCalendar.YEAR));
```

HALLAR CANTIDAD DE AÑOS ENTRE 2 FECHAS

```
public final static int YEAR = GregorianCalendar.YEAR;
public final static int MONTH = GregorianCalendar.MONTH;
public final static int DAY = GregorianCalendar.DAY_OF_MONTH;

public static int getDiffYear(GregorianCalendar lo, GregorianCalendar hi)
```

```
{
   if(lo.after(hi)) return -1; // Rango de fechas incorrecto
   int ans = hi.get(YEAR) - lo.get(YEAR);
   if( hi.get(MONTH) < lo.get(MONTH) ||
      (hi.get(MONTH) == lo.get(MONTH) && hi.get(DAY) < lo.get(DAY)))
      ans--;
   return ans;
}
```

COMBINATORIAS

PROPIEDADES

1. $C_k^n = C_{n-k}^n$
2. $C_k^n = \frac{n}{k} \times (C_{k-1}^{n-1})$
3. $C_0^n = 1, \forall n \geq 0$
4. $C_k^0 = 0, \forall k > 0$

COMBINACIONES SIMPLES

```
int comb(int n, int k)
{
   double res = 1;
   if(k > n - k) k = n - k;                            // Propiedad 1
   while(k != 0) { res *= 1.0 * n / k; n--; k--; } // Propiedad 2
   return (int)res;
}
```

CALCULAR TODAS LAS COMBINACIONES

Triángulo de Pascal:

1	0	0	0	0
1	**1**	0	0	0
1	**2**	**1**	0	0
1	**3**	**3**	**1**	0
1	**4**	**6**	**4**	**1**

```
#define MOD 1000000007
#define MAX 10005

ll C[MAX][MAX];

void init() // Armar triángulo de Pascal
{
   memset(C, 0, sizeof C);
   for(int i = 0; i < MAX; i++) // Para cada fila..
   {
      C[i][0] = 1; // La primera columna es 1
```

```
      for(int j = 1; j <= i; j++) // Para el resto de columnas..
         C[i][j] = (C[i-1][j] + C[i-1][j-1]) % MOD; // DP
   }
}
```

TEOREMA DE LUCAS

Sean n y m números enteros no negativos, p un número primo y sean:

- $n = \overline{n_0 n_1 n_2 \ldots n_k}_{\,p}$ (número n en base p)
- $m = \overline{m_0 m_1 m_2 \ldots m_k}_{\,p}$ (número m en base p).

Entonces: $$C_m^n = \prod_{i=0}^{k} \binom{n_i}{m_i} (\bmod p)$$

```
#define MOD 3571
int C[MOD][MOD];

void FillLucasTable()
{
   memset(C, 0, sizeof C);
   for(int i = 0; i < MOD; i++) C[i][0] = 1;
   for(int i = 1; i < MOD; i++) C[i][i] = 1;
   for(int i = 2; i < MOD; i++)
      for(int j = 1; j < i; j++)
         C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % MOD;
}

int comb(int n, int k)
{
   ll ans = 1;
   while(n != 0)
   {
      int ni = n % MOD, n /= MOD; // Extraer último digito de n
      int ki = k % MOD; k /= MOD; // Extraer último digito de k
      ans = (ans * C[ni][ki]) % MOD;
   }
   return (int)ans;
}
```

COMBINACIONES CON REPETICIONES

Algoritmo para calcular en cuantas formas puedo reordenar una palabra usando sus letras.

```
#define MAX 30

double wordPermutations(char *str)
{
   int de[MAX] = {0}, ss[300] = {0}, len = strlen(str), j = 0;
```

```
   double c = 1.0, d = 1.0;
   for(int i = 0; i < len; i++)
   {
      ss[str[i]]++;
      if(ss[str[i]] > 1) de[j++] = ss[str[i]];
   }
   for(int i = 2; i <= len; i++)
   {
      c *= i;
      if(j > 0) d *= de[--j];
      if(d != 1 && !fmod(c, d)) { c /= d; d = 1; }
   }
   return c;
}
```

Ejemplo: printf("Cant. permutaciones de la palabra: %.0lf\n", WordPermutations(cad));

NÚMEROS CATALANES

Los primeros números catalanes son: 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452

Los números catalanes siguen la fórmula: $Cat(0) = 1, \quad Cat(n+1) = \dfrac{4n+2}{n+2} \times Cat(n)$

CÓDIGO C++

```
int Catalan[21];

void init()
{
   Catalan[0] = 1;
   for(int n = 0; n < 20; n++)
      Catalan[n + 1] = Catalan[n] * (4 * n + 2) / (n + 2);
}
```

CÓDIGO JAVA (BIGINTEGER)

```
BigInteger Catalan[] = new BigInteger[21];
for(int n = 0; n < 21; n++) Catalan[n] = BigInteger.ONE;
for(int n = 0; n < 20; n++)
   Catalan[n + 1] = Catalan[n].
                     multiply(BigInteger.valueOf(4 * n + 2)).
                     divide(BigInteger.valueOf(n + 2));
```

APLICACIONES

- Cat(n) es el número de “Dyck words” de longitud 2n. Un “Dyck word” es una cadena de n X's y n Y's de modo que ningún segmento inicial tiene más Y's que X's. Ejemplo: “Dyck words” de longitud 6:

XXXYYY   XYXXYY   XYXYXY   XXYYXY   XXYXYY.

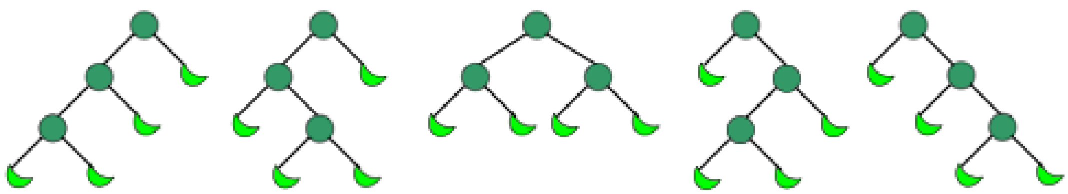
- Si cambiamos las X por un paréntesis abierto, y la Y por un paréntesis cerrado, Cat(n) es la cantidad de expresiones de n pares de paréntesis que están correctamente balanceadas:

((()))   ()(())   ()()()   (())()   (()())

- Cat(n) representa la cantidad que n + 1 factores pueden ser asociados con un operador binario. Para n = 3, tenemos:

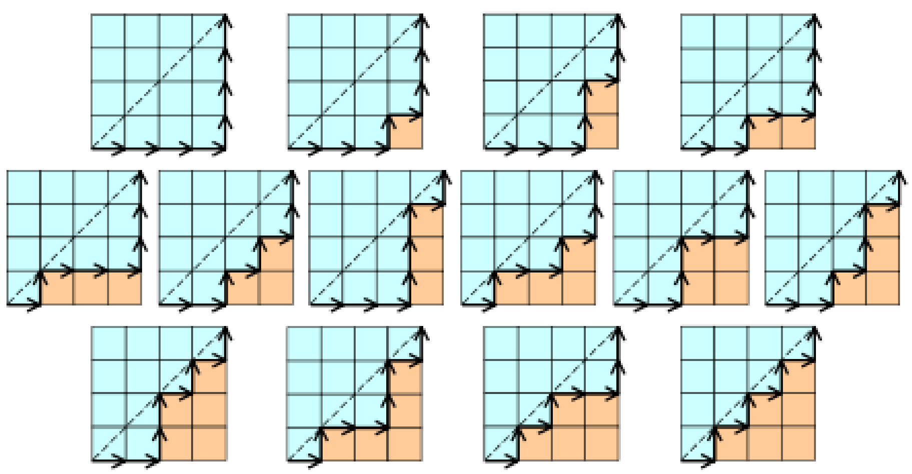
$$((ab)c)d \quad (a(bc))d \quad (ab)(cd) \quad a((bc)d) \quad a(b(cd))$$

- Cat(n) es el número de árboles binarios que se puede formar con n nodos. Para n = 3, tenemos:



Si el orden de los nodos importa, entonces la respuesta es: Cat(n) x n!.

- Cat(n) es el número de caminos que van de un extremo a otro de un cuadrado de n x n, que no pasan por la diagonal. Esto es equivalente a contar "Dyck words", donde X es "derecha" e Y es "arriba". Para n = 4, tenemos:



NÚMEROS SUPER CATALANES

Los primeros números super catalanes son: 1, 1, 3, 11, 45, 197, 903, 4279, 20793, 103049, 518859, 2646723, 13648869, 71039373, 372693519, 1968801519, 10463578353, 55909013009, 300159426963, 1618362158587, 8759309660445, 47574827600981, 259215937709463, 1416461675464871

Los números catalanes siguen la fórmula:

$$SCat(1) = 1,\quad SCat(2) = 1,\quad SCat(n) = \frac{(6n-9)\times SCat(n-1) - (n-3)\times SCat(n-2)}{n}$$

CÓDIGO JAVA (BIGINTEGERS)

```
BigInteger SuperCatalan[] = new BigInteger[20];
for(int n = 0; n < 21; n++) SuperCatalan[n] = BigInteger.ONE;
for(int n = 3; n < 21; n++)
   SuperCatalan[n] = SuperCatalan.[n - 1].
                       multiply(BigInteger.valueOf(6 * n - 9)).
                       subtract(SuperCatalan[n - 2].
                                multiply(BigInteger.valueOf(n - 3))).
                       divide(BigInteger.valueOf(n)));
```

APLICACIONES

- SCat(n) es el número de formas de insertar paréntesis en una secuencia de n símbolos. Los paréntesis pueden agrupar dos o más símbolos. Para n = 4, tenemos 11 formas:

  xxxx, (xx)xx, x(xx)x, xx(xx), (xxx)x, x(xxx), ((xx)x)x, (x(xx))x, (xx)(xx), x((xx)x), x(x(xx)).

PROBABILIDADES

SUBFACTORIALES

El subfactorial de un número n (escrito como !n) es el número de posibles desarreglos (permutación donde **ninguno** de sus elementos aparece en la posición original) de un conjunto con n elementos.

$$!n = (n-1)\ (!(n-1)+!(n-2)) \qquad \text{for } n \geq 2\ ,$$

```
#define MAX 20

int subfac[MAX];

void calc()
{
   subfac[0] = 1; subfac[1] = 0;
   for(int i = 2; i < MAX; i++) subfac[i] = (i - 1) * (subfac[i - 1] + subfac[i - 2]);
}
```

VALOR ESPERADO

$$\mathrm{E}[X] = x_1p_1 + x_2p_2 + \cdots + x_kp_k\ .$$

VALOR ESPERADO RECURSIVO

Si intentamos realizar con éxito un evento que tiene probabilidad p, el valor esperado de veces que se intentará hasta obtener un éxito será

$$\begin{aligned} E(X) &= p \cdot 1 + (1-p) \cdot (E(X)+1) \\ E(X) \cdot (1-(1-p)) &= p + 1 - p \\ E(X) &= \frac{1}{p} \end{aligned}$$

CYCLE FINDING

Teniendo una secuencia $\{x_0, x_1 = f(x_0), x_2 = f(x_1), \ldots, x_i = f(x_{i-1}), \ldots\}$, cuya función *f* trabaja en un conjunto finito de números, entonces $\exists i \neq j$ tal que $x_i \neq x_j$. El algoritmo devuelve un par (μ, λ) donde μ es el índice *i* de inicio del ciclo y λ es la extensión del ciclo.

Complejidad: O(μ + λ)

ALGORITMO

```
typedef pair<int,int> ii;

ii floydCycleFinding(int x0)
{
 int tortuga = f(x0), liebre = f(f(x0)), mu = 0, lambda = 1;
 while(tortuga != liebre) { tortuga = f(tortuga); liebre = f(f(liebre)); }
 liebre = x0;
 while(tortuga != liebre) { tortuga = f(tortuga); liebre = f(liebre); mu++; }
 liebre = f(tortuga);
 while(tortuga != liebre) { liebre = f(liebre); lambda++; }
 return ii(mu, lambda); // mu: id de inicio, lambda: extensión
}
```

CANTIDAD DE NUMEROS DIFERENTES GENERADOS DESDE X_0

```
En el codigo anterior retornar "mu + lambda"
```

ENCONTRAR EL MAXIMO NUMERO DEL CICLO

```
int floydCycleFinding(int x0)
{
  int tortoise = f(x0), rabbit = f(f(x0)), maximo = x0;
  while(tortoise != rabbit){
    tortoise = f(tortoise); rabbit = f(f(rabbit)); }
  rabbit = x0;
  while(tortoise != rabbit){
    tortoise = f(tortoise); rabbit = f(rabbit); maximo = max(maximo, rabbit);
  }
  rabbit = f(tortoise); maximo = max(maximo, rabbit);
  while(tortoise != rabbit) {
    rabbit = f(rabbit); maximo = max(maximo, rabbit);
  }
```

```
  return maximo;
}
```

ROOT FINDING

El problema de root finding consiste en hallar un valor *x* de tal manera que *f(x)* = 0. Cualquier ecuación del tipo *f*(*x*) = *g*(*x*), se convertirá en una ecuación del tipo *f(x)* – *g(x)* = 0 antes de resolverla.

Para saber si una función *f(x)* = 0 en un rango [*a*,*b*] tiene solución, se multiplica *f(a)* y *f(b)*, si el resultado es mayor que 0 se puede afirmar que no hay solución.

MÉTODO DE LA BISECCIÓN (DIVIDE Y VENCERÁS)

```
#define EPS 1e-7

double f(double x) { // Ej: Si f(x) = 3 * x, pondriamos: return 3 * x; }

double bisection(double lo, double hi)
{
   double mid;
   while(lo + EPS < hi)
   {
      mid = (lo + hi) * 0.5;
      if(f(lo) * f(mid) <= 0) hi = mid;
      else lo = mid;
   }
   return (lo + hi) * 0.5;
}
```

Ejemplo:

```
double a = 0, b = 1; // rango [a,b]
if(f(a) * f(b) > 0)  printf("No solution\n");
else  printf("%.4lf\n", bisection(a, b));
```

MÉTODO DE LA SECANTE (MÁS EFECTIVO QUE EL ANTERIOR)

```
#define EPS 1e-7

double secant(double lo, double hi)
{
   double x0 = lo, x1 = hi, delta;
   while(true)
   {
      delta = f(x1) * (x1 - x0) / (f(x1) - f(x0));
      if(fabs(delta) < EPS) return x1;
      x0 = x1; x1 = x1 - delta;
   }
}
```

MÉTODO DE NEWTON (EL MÁS EFECTIVO)

```
#define EPS 1e-7

double fd(double x) { // definimos la derivada de la función f(x) }

double newton(double lo, double hi)
{
   if(f(lo) == 0) return lo;
   double x = (lo + hi) * 0.5, x1;
   while(true)
   {
      x1 = x - f(x) / fd(x);
      if (fabs(x1 - x) < EPS) return x;
      x = x1;
   }
}
```

MATRICES

MATRIX EXPONENTIATION

```
#define MAX 2
const int MOD = 1e9 + 5;

struct Matrix
{
  int M[MAX][MAX];
  Matrix operator * (Matrix &a) // Multiplication: O(MAX^3)
  {
    Matrix ans;
    memset(ans.M, 0, sizeof ans.M);
    for(int i = 0; i < MAX; i++)
      for(int j = 0; j < MAX; j++)
        for(int k = 0; k < MAX; k++)
          ans.M[i][j] = (ans.M[i][j] + (M[i][k] * a.M[k][j]) % MOD) % MOD;
          // TRANSITIVE CLOSURE:
          // ans.M[i][j] = ans.M[i][j] | (M[i][k] & a.M[k][j]);
    return ans;
  }
  // Exponentiation: O(MAX^3 lg p)
  Matrix pow(int p) // p: Exponent
  {
    Matrix ans, b = *this;
    for(int i = 0; i < MAX; i++)
      for(int j = 0; j < MAX; j++)
        ans.M[i][j] = (i == j); // Identity Matrix
    while(p)
    {
      if(p & 1)
        ans = ans * b;
      p >>= 1;
      b = b * b;
    }
```

```
      return ans;
   }
};
```

APLICATIONS

1. ***Calculate F(n) in O(log N), where:*** $F(n) = a_1 \times F(n-1) + a_2 \times F(n-2) + \ldots + a_d F(n-d)$

$$\begin{bmatrix} F(n) \\ F(n-1) \\ F(n-2) \\ \ldots \\ F(n-d+1) \end{bmatrix}^{n} = \begin{bmatrix} a_1 & a_2 & \ldots & a_{d-1} & a_d \\ 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & \ldots & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & 1 & 0 \end{bmatrix}^{n-d} \times \begin{bmatrix} F(d) \\ F(d-1) \\ F(d-2) \\ \ldots \\ F(1) \end{bmatrix}$$

2. ***Calculate the n-th fibonacci number in O(log n) use the matrix:***

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n} = \begin{bmatrix} fibo(n+1) & fibo(n) \\ fibo(n) & fibo(n-1) \end{bmatrix}$$

DETERMINANT

```
#define MAX_SIZE 500
#define EPS 1e-7

struct Matrix
{
   double X[MAX_SIZE][MAX_SIZE];
   Matrix() {}
};

double determinant(Matrix M0, int size)
{
   double ans = 1, aux;
   bool found;
   for(int i = 0, r = 0; i < size; i++)
   {
      found = false;
      for(int j = r; j < size; j++)
         if(fabs(M0.X[j][i]) > EPS)
         {
            found = true;
            if(j > r) ans = -ans; else break;
            for(int k = 0; k < size; k++) swap(M0.X[r][k], M0.X[j][k]);
            break;
         }
      if(found)
      {
         for(int j = r + 1; j < size; j++)
```

```
         {
            aux = M0.X[j][i] / M0.X[r][i];
            for(int k = i; k < size; k++) M0.X[j][k] -= aux * M0.X[r][k];
         }
         r++;
      }
      else return 0;
   }
   for(int i = 0; i < size; i++) ans *= M0.X[i][i];
   return ans;
}

bool DeterminantIsZero(Matrix M0, int size)
{
   double aux; bool found;
   for(int i = 0;i < size; i++)
   {
      if(fabs(M0.X[i][i]) > EPS) found = true;
      else
      {
         found = false;
         for(int j = i + 1; j < size; j++)
            if(fabs(M0.X[j][i])>EPS)
            {
               found = true;
               for(int k = 0;k < size; k++) swap(M0.X[i][k], M0.X[j][k]);
               break;
            }
      }
      if(found)
      {
         for(int j = i + 1;j < size; j++)
         {
            aux = M0.X[j][i] / M0.X[i][i];
            for(int k = i; k < size; k++) M0.X[j][k] -= aux * M0.X[i][k];
         }
      }
      else return true;
   }
   return false;
}
```

SYLVESTER MATRIX

Dados polinomios p y q:

$$p(z) = p_0 + p_1 z + p_2 z^2 + \cdots + p_m z^m, \; q(z) = q_0 + q_1 z + q_2 z^2 + \cdots + q_n z^n.$$

La matriz de Sylvester para ambos polinomios, es una matriz (n+m) x (n+m), que se forma con los coeficientes de ambos polinomios. Por ejemplo, para m = 4 y n = 3:

$$S_{p,q} = \begin{pmatrix} p_4 & p_3 & p_2 & p_1 & p_0 & 0 & 0 \\ 0 & p_4 & p_3 & p_2 & p_1 & p_0 & 0 \\ 0 & 0 & p_4 & p_3 & p_2 & p_1 & p_0 \\ q_3 & q_2 & q_1 & q_0 & 0 & 0 & 0 \\ 0 & q_3 & q_2 & q_1 & q_0 & 0 & 0 \\ 0 & 0 & q_3 & q_2 & q_1 & q_0 & 0 \\ 0 & 0 & 0 & q_3 & q_2 & q_1 & q_0 \end{pmatrix}.$$

Entonces:

- Dos polinomios tienen una raíz en común si el determinante de su matriz de Sylvester es cero.
- Si se quiere saber si un polinomio tiene raíces múltiples se puede tomar al polinomio, junto con su derivada y verificar que la determinante sea distinta de cero.

POLYNOMIALS

OPERATIONS

```
vi add(const vi &a, const vi &b)
{
  int n = (int)a.size(), m = (int)b.size(), sz = max(n,m);
  vector<int> c(sz,0);
  for(int i = 0; i < n; i++)
    c[i] += a[i];
  for(int i = 0; i < m; i++)
    c[i] += b[i];
  while(sz > 1 && c[sz - 1] == 0)
  {
    c.pop_back();
    sz--;
  }
  return c;
}

vi mul(const vi &a, const vi &b)
{
  int n = (int)a.size(), m = (int)b.size(), sz = n + m - 1;
  vector<int> c(sz, 0);
  for(int i = 0; i < n; i++)
    for(int j = 0; j < m; j++)
     c[i + j] += a[i] * b[j];
  while(sz > 1 && c[sz - 1] == 0)
  {
    c.pop_back();
    --sz;
  }
  return c;
}

bool is_root(const vi &p, int r)
```

```
{
  int n = (int)p.size();
  long long y = 0;
  for(int i = 0; i < n; i++)
  {
    if(labs(y - p[i]) % r != 0)
      return false;
    y = (y - p[i]) / r;
  }
  return y == 0;
}
```

OTHERS

SAFE MULTIPLICATION

```
ll mult(ll a, ll b) // Tiempo logaritmico
{
  ll ans = 0;
  while(b)
  {
    if(b & 1) ans += a;
    if(ans >= MOD) ans -= MOD;
    a <<= 1;
    if(a >= MOD) a -= MOD;
    b >>= 1;
  }
  return ans;
}
```

FAST FOURIER TRANSFORM

```
#define PI (2*acos(0.0))

typedef complex<double> base;

void fft(vector<base> &a, bool invert)
{
  int n = (int)a.size();
  for(int i = 1, j = 0; i < n; i++)
  {
    int bit = n >> 1;
    for(; j >= bit; bit >>= 1)
      j -= bit;
    j += bit;
    if(i < j)
      swap(a[i], a[j]);
  }
  for(int len = 2; len <= n; len <<= 1)
  {
    double ang = 2 * PI / len * (invert? -1 : 1);
    base wlen(cos(ang), sin(ang));
    for(int i = 0; i < n; i += len)
    {
```

```
      base w(1);
      for(int j = 0; j < len / 2; j++)
      {
        base u = a[i + j], v = a[i + j + len/2] * w;
        a[i + j] = u + v;
        a[i + j + len/2] = u - v;
        w *= wlen;
      }
    }
  }
  if(invert)
    for(int i = 0; i < n; i++)
      a[i] /= n;
}

void mul(const vi &a, const vi &b, vi &res)
{
  vector<base> fa(a.begin(), a.end()), fb(b.begin(), b.end());
  int n = 1;
  while(n < max(a.size(), b.size()))
    n <<= 1;
  n <<= 1;
  fa.resize(n), fb.resize(n);
  fft(fa, false), fft(fb, false);
  for(int i = 0; i < n; i++)
    fa[i] *= fb[i];
  fft(fa, true);
  res.resize(n);
  for(int i = 0; i < n; i++)
    res[i] = int(fa[i].real() + 0.5);
}
```

GAME THEORY

NIM

REGULAR NIM

- Winner: the player that makes the last move.
- Loser: the player that can't make a move.

The 1st player has a winning strategy iff the XOR of all heap sizes is non zero. Otherwise the 2nd player has a winning strategy.

```
int ans = 0;
for(int i = 0; i < n; i++) // n: Number of heaps
{
  cin >> num; // num: Size of the heap i
  ans ^= num; // XOR the size of the heaps
}
cout << ((ans != 0)? "1st wins" : "2nd wins");
```

MISERE NIM

- Winner: the player that can't make a move.
- Loser: the player that makes the last move.

When all the heaps are of size 1, the 1st player has a winning strategy iff the number of heaps is even. Othewise, apply conditions of a normal nim.

```
int ans = 0;
bool ones = true;
for(int i = 0; i < n; i++) // n: Number of heaps
{
  cin >> num;                 // num: Size of the heap i
  ans ^= num;                 // XOR the size of the heaps
  ones = ones && (num == 1); // Verify if heaps only have size 1
}
if(ones) // If all heaps are of size one..
  cout << ((n % 2 == 0)? "1st wins" : "2nd wins"); // Depends on parity of n
else
  cout << ((ans != 0)? "1st wins" : "2nd wins");    // Normal nim
```

GRAPHS

GRAPH THEORY

HAVEL-HAKIMI ALGORITHM

Determines if a sequence of natural numbers could be the degree sequence of a simple graph.

```
vi v;  // Degree sequence to evaluate
int n; // Number of elements

bool solve(int s, int m) // s: Sum of numbers in v, m: Max element in v
{
  if(s == 0)                 // If the sequence contains only zeros..
    return true;             // It is valid
  if(m >= n || s % 2 != 0)   // No node can be adjacent to more than n-1 nodes
    return false;            // And sum needs to be even, otherwise no answer
  sort(v.rbegin(), v.rend()); // Sort non-increasingly
  for(int i = 0; i < n; i++)  // For each i..
  {
    for(int j = i+1; j < min(i+1+v[i], n); j++) // For the next v[i] terms..
    {
      v[j]--;          // Substract one
      if(v[j] < 0)     // If we get a negative number..
        return false; // We cannot build the graph
    }
    sort(v.rbegin(), v.rend()); // Reorder the sequence
  }
  return true;
}
```

LANDAU THEOREM

Given a sequence of numbers: [s_0,s_1,s_2,..., s_{n-1}], this sequence represent a tournament graph iff:

1. $0 \leq s_0 \leq s_1 \leq \cdots \leq s_{n-1}$
2. $s_0 + s_1 + \cdots + s_i \geq \frac{i(i+1)}{2}$; for all i ϵ [0, n – 1]
3. $s_0 + s_1 + \cdots + s_{n-1} = \frac{n(n-1)}{2}$

Then $\forall i$: $s_i \in [\max\left(s_{i-1}, \frac{i(i+1)}{2} - (s_0 + s_1 + \cdots + s_{i-1})\right), \frac{\frac{n(n-1)}{2} - (s_0+s_1+\cdots+s_{i-1})}{n-i}]$

UNWEIGHTED GRAPHS

IMPLEMENTATION

```
typedef vector<int> vi; // Integer array
typedef vector<vi> vvi; // Adjacency List
vvi g;                  // g: Graph
int n = 5;              // n: Number of nodes
g = vvi(n);             // Creates a graph g with n nodes
```

INSERT EDGE

```
int x = 0, y = 1;  // x, y: Node index [0, n-1]
g[x].push_back(y); // Inserts directed edge x -> y
```

BFS

```
vi dist;  // dist[x]: Distance from source to node x
vi prev;  // prev[x]: Parent of node x
vi color; // 0: Not discovered, 1: Discovered, 2: Processed

void init() // Cleans the arrays
{
  dist = vi(n, INF); // INF : The node is unreachable
  prev = vi(n, -1);  // -1  : The node has no parent
  color = vi(n, 0);  //  0  : The node has not been discovered
}

void bfs(int s) // s: Source node
{
  queue<int> q; q.push(s); // Queue the source node
  dist[s] = 0;             // Distance from the source to itself is 0
  color[s] = 1;            // Mark the source node as discovered
  while(!q.empty())        // While the queue has nodes..
  {
    int v = q.front(); q.pop(); // Get node v from queue
    for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor of v..
    {
      int u = g[v][i];  // Neighbor u. Edge v -> u
      if(color[u] == 0) // If node u has not been discovered..
      {
        q.push(u);             // Queue node u
        dist[u] = dist[v] + 1; // Update distance
        prev[u] = v;           // Update parent
        color[u] = 1;          // Mark node as discovered
      }
    }
    color[v] = 2; // Node v is now processed
  }
}
```

BFS IN A MATRIX

```
#define MAX 1005
typedef pair<int,int> ii; // Pair: (Row, Column)
// 4 Movements: Right, Left, Down, Up
int aux_x[4] = {0, 0, 1, -1};
int aux_y[4] = {1, -1, 0, 0};
int r,c;             // r: Number of rows, c: Number of columns
int dist[MAX][MAX];  // dist[x][y]: Distance from source to node (x, y)
int color[MAX][MAX]; // 0: Not discovered, 1: Discovered, 2: Processed

void init() // Cleans the matrices
{
  for(int i = 0; i < r; i++)
    for(int j = 0; j < c; j++)
    {
      dist[i][j] = INF; // INF : The node is unreachable
      color[i][j] = 0;  //  0  : The node has not been discovered
    }
}

void bfs(int px, int py) // (px, py): Source node
{
  queue<ii> q; q.push(ii(px, py)); // Queue the source node
  dist[px][py] = 0;  // Distance from source to itself is 0
  color[px][py] = 1; // Mark the source node as discovered
  while(!q.empty())  // While the queue has nodes..
  {
    int x = q.front().first;   // Get x from queue
    int y = q.front().second;  // Get y from queue
    q.pop();                   // Remove node from queue
    for(int i = 0; i < 4; i++) // For each neighbor of (x, y)..
    {
      int nx = x + aux_x[i]; // Neighbor nx
      int ny = y + aux_y[i]; // Neighbor ny
      // (nx, ny) must be inside the matrix and not visited before
      // One can also validate that (nx,ny) is not an invalid position (wall)
      if(nx > -1 && ny > -1 && nx < r && ny < c && color[nx][ny] == 0)
      {
        q.push(ii(nx,ny));              // Queue node (nx,ny)
        dist[nx][ny] = dist[x][y] + 1; // Update distance
        color[nx][ny] = 1;              // Mark node as discovered
      }
    }
    color[x][y] = 2;  // Node (x,y) is now processed
  }
}
```

MULTI-SOURCE BFS

```
vi color; // 0: Not discovered, 1: Discovered, 2: Processed
vi prev;  // prev[x]: Parent of node x
vi dist;  // dist[x]: Distance from source to node x
```

```
void init() // Cleans the arrays
{
  dist = vi(n, INF); // INF : The node is unreachable
  prev = vi(n, -1);  // -1  : The node has no parent
  color = vi(n, 0);  //  0  : The node has not been discovered
}

void bfs(const vector<int>& s) // s: Array of source nodes
{
  queue<int> q; // q: Queue of nodes
  for(int i = 0; i < (int)s.size(); i++) // For each source..
  {
    q.push(s[i]);     // Queue the source
    dist[s[i]] = 0;   // Distance from source to itself is 0
    color[s[i]] = 1;  // Mark the source node as discovered
  }
  while(!q.empty())  // While the queue has nodes..
  {
    int v = q.front(); q.pop(); // Get node v from queue
    for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor of v..
    {
      int u = g[v][i];  // Neighbor u. Edge v -> u
      if(color[u] == 0) // If node u has not been discovered..
      {
        q.push(u);              // Queue node u
        dist[u] = dist[v] + 1;  // Update distance
        prev[u] = v;            // Update parent
        color[u] = 1;           // Mark node as discovered
      }
    }
    color[v] = 2; // Node v is now processed
  }
}
```

PRINT SHORTEST PATH

```
void printPath(int x, int y)
{
  // If y == -1, there is no path from x to y
  if(x == y || y == -1) { cout << y; return; }
  printPath(x, prev[y]);
  cout << " " << y;
}
```

BIPARTITE GRAPH CHECK

```
vi color;   // 0: Not discovered, 1: Belongs to set #1, 2: Belongs to set #2
vi setSize; // setSize[i]: Number of nodes in set i

void init() // Cleans the arrays
{
  color = vi(n, 0);   // 0 : The node has not been discovered
  setSize = vi(2, 0); // Both sets starts with 0 nodes
}
```

```
bool isBipartite(int s) // s: Source node
{
  queue<int> q; q.push(s); // Queue the source node
  color[s] = 1;            // Assign to set #1
  setSize[0]++;            // Set #1 contains 1 node
  while(!q.empty())        // While the queue has nodes..
  {
    int v = q.front(); q.pop(); // Get node v from queue
    for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor of v..
    {
      int u = g[v][i];  // Neighbor u. Edge v -> u
      if(color[u] == 0) // If node u has not been discovered..
      {
        q.push(u);                        // Queue node u
        color[u] = (color[v] == 1)? 2 : 1; // Assign a different color
        setSize[color[u] - 1]++;          // Increase the size of the set
      }
      else if(color[u] == color[v]) // If v and u have the same color..
        return false;               // Graph is not bipartite
    }
  }
  return true; // Graph is bipartite
}
```

DFS

```
vi color; // 0: Not discovered, 1: Discovered, 2: Processed

void dfs_visit(int v) // v: Current node
{
  color[v] = 1; // Mark node v as discovered
  for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor of v..
  {
    int u = g[v][i];  // Neighbor u. Edge v -> u
    if(color[u] == 0) // If node u has not been discovered..
      dfs_visit(u);   // Visit node u
  }
  color[v] = 2; // Node v is now processed
}

void dfs()
{
  color = vi(n, 0);          // No node has been discovered
  for(int i = 0; i < n; i++) // For each node i..
    if(color[i] == 0)        // If the node i has not been discovered..
      dfs_visit(i);          // Visit the connected component
}
```

DFS IN A MATRIX

```
int aux_x[4] = {0, 0, 1, -1}; // 4 Movements: Right, Left, Down, Up
int aux_y[4] = {1, -1, 0, 0}; // 4 Movements: Right, Left, Down, Up
int r,c;              // r: Number of rows, c: Number of columns
int color[MAX][MAX]; // 0: Not discovered, 1: Discovered, 2: Processed
```

```
void dfs_visit(int px, int py) // (px, py): Current node
{
  color[px][py] = 1; // Mark node (px,py) as discovered
  for(int i = 0; i < 4; i++) // For each neighbor of (px, py)..
  {
    int nx = px + aux_x[i]; // Neighbor nx
    int ny = py + aux_y[i]; // Neighbor ny
    if(nx > -1 && ny > -1 && nx < r && ny < c && color[nx][ny] == 0)
      dfs_visit(nx, ny);    // Visit node (nx,ny)
  }
  color[px][py] = 2;  // Node (px,py) is now processed
}

void dfs()
{
  memset(color, 0, sizeof color); // No node has been discovered
  for(int i = 0; i < R; i++)   // For each row i..
    for(int j = 0; j < C; j++) // For each column j..
      if(color[i][j] == 0)     // If the node (i,j) has not been discovered..
        dfs_visit(i, j);       // Visit the connected component
}
```

TOPOLOGICAL SORT

```
vi color;  // 0: Not discovered, 1: Discovered, 2: Processed
vi sorted; // The array will contain a valid topological sort

void dfs_visit(int v) // v: Current node
{
  color[v] = 1; // Mark node v as discovered
  for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor of v..
  {
    int u = g[v][i];  // Neighbor u. Edge v -> u
    if(color[u] == 0) // If the node u has not been discovered..
      dfs_visit(u);   // Visit node u
  }
  color[v] = 2;        // Node v is now processed
  sorted.push_back(v); // Add the processed node v
}

void topsort()
{
  color = vi(n, 0);          // No node has been discovered
  sorted.clear();            // Clean the topological sort array
  for(int i = 0; i < n; i++) // For each node i..
    if(color[i] == 0)        // If the node i has not been discovered..
      dfs_visit(i);          // Visit the connected component
  reverse(sorted.begin(),sorted.end()); // Reverse the array
}
```

CYCLE DETECTION – DIRECTED GRAPH

```
vi color; // 0: Not discovered, 1: Discovered, 2: Processed
```

```
bool dfs_visit(int v) // v: Current node
{
  color[v] = 1;      // Mark node v as discovered
  bool ans = false; // No cycle found yet
  for(int i = 0; i < (int)g[v].size() && !ans; i++) // For each neighbor of v
  {
    int u = g[v][i];        // Neighbor u. Edge v → u
    if(color[u] == 0)       // If node u has not been discovered..
      ans = dfs_visit(u);   // Visit node u
    else if(color[u] == 1) // If node u has not been processed..
      ans = true;           // There is a cycle
  }
  color[v] = 2; // Node v is now processed
  return ans;   // Return result
}

bool hasCycle()
{
  color = vi(n, 0); // No node has been discovered
  for(int i = 0; i < n; i++)                  // For each node i..
    if(color[i] == 0 && dfs_visit(i, -1)) // Check the connected component..
      return true;                          // There is a cycle
  return false; // No cycles found
}
```

CYCLE DETECTION – UNDIRECTED GRAPH

```
vi color; // 0: Not discovered, 1: Discovered, 2: Processed

bool dfs_visit(int v, int p) // v: Current node, p: Previous node
{
  color[v] = 1;      // Mark node v as discovered
  bool ans = false; // No cycle found yet
  for(int i = 0; i < (int)g[v].size() && !ans; i++) // For each neighbor of v
  {
    int u = g[v][i];          // Neighbor u. Edge v → u
    if(color[u] == 0)         // If node u has not been discovered..
      ans = dfs_visit(u, v); // Visit node u
    else if(u != p)           // The node u is not the predecessor..
      ans = true;             // There is a cycle
  }
  color[v] = 2; // Node v is now processed
  return ans;   // Return result
}

bool hasCycle()
{
  color = vi(n, 0); // No node has been discovered
  for(int i = 0; i < n; i++)                  // For each node i..
    if(color[i] == 0 && dfs_visit(i, -1)) // Check the connected component..
      return true;                          // There is a cycle
  return false; // No cycles found
}
```

CYCLE FINDING – UNDIRECTED GRAPH

```
vi color;  // 0: Not discovered, 1: Discovered, 2: Processed
vi h;      // Position of the node in the current path
vi path;   // Current path of nodes visited by DFS
// Cycle information
int ncycles;  // Number of cycles in the graph
vi cycle;     // cycle[i]: Cycle ID to which the node i belongs
vi cycleSize; // cycleSize[i] = Size of cycle i

void init() // Cleans the arrays
{
  color = vi(n, 0);  //  0 : The node has not been discovered
  h = vi(n, 0);      //  Reserve space for n nodes
  path.clear();      //  There is no visited path yet
  cycle = vi(n, -1); // -1 : The node does not belong to a cycle
  cycleSize.clear(); // There are no cycles
  ncycles = 0;       // There are no cycles
}

void dfs_visit(int v, int p) // v: Current node, p: Previous node
{
  color[v] = 1;      // Mark node v as discovered
  path.push_back(v); // Add node v to current path
  h[v] = (p == -1)? 0 : (h[p] + 1); // Update height of node v
  for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor of v..
  {
    int u = g[v][i];       // Neighbor u. Edge v → u
    if(color[u] == 0)      // If node u has not been discovered..
      dfs_visit(u, v);     // Visit node u
    else if(u != p)        // If node u is not the predecessor..
    {
      cycleSize.push_back(h[v] - h[u] + 1); // Add cycle size
      for(int i = h[u]; i <= h[v]; i++)     // For each vertex in cycle..
        cycle[path[i]] = ncycles;           // Store cycle ID
      ncycles++;   // Increment number of cycles
    }
  }
  color[v] = 2;     // Node v is now processed
  path.pop_back(); // Remove node v from the current path
}
```

STRONG CONNECTED COMPONENTS – DIRECTED GRAPHS

```
vvi r;     // Graph with reversed edges
vi color; // 0: Not discovered, 1: Discovered, 2: Processed
vi st;     // Stack used on first pass
// SCC information
vi scc;    // scc[i] : SCC ID to which node i belongs
int nSCC; // Number of SCC the graph has

void dfs_visit(int v, int pass) // v: Current node
{
  color[v] = 1;               // Mark node v as discovered
```

```
  int sz = (pass == 1)? g[v].size() : r[v].size(); // Number of neighbors
  for(int i = 0; i < sz; i++) // For each neighbor of v..
  {
    int u = (pass == 1)? g[v][i] : r[v][i]; // Neighbor u. Edge v -> u
    if(color[u] == 0)     // If the node u has not been discovered..
      dfs_visit(u, pass); // Visit node u
  }
  if(pass == 1)        // If this is the 1st pass..
    st.push_back(v); // Add the processed node v to the stack
  else                 // If this is the 2nd pass..
    scc[v] = nSCC;   // Update the SCC index of node v
}

void dfs()
{
  color = vi(n, 0);           // No node has been discovered
  st.clear();                 // Clean the stack
  for(int i = 0; i < n; i++) // For each node i..
    if(color[i] == 0)         // If the node i has not been discovered..
      dfs_visit(i, 1);        // Visit the connected component (1st time)
  color = vi(n, 0);           // No node has been discovered
  scc = vi(n, -1);            // Clean scc array
  nSCC = 0;                   // No SCC
  for(int i = n - 1; i >= 0; i--) // For each node i (in stack order)..
    if(color[st[i]] == 0)         // If the node i has not been discovered..
    {
      dfs_visit(st[i], 2);        // Visit the connected component (2nd time)
      nSCC++;                     // Number of SCC increments
    }
}
```

ARTICULATION BRIDGES – UNDIRECTED GRAPHS

```
int dfs_cont; // Counter
vi dfs_num;   // Value of dfs_cont when a node is visited for the first time
vi dfs_low;   // Lowest dfs_num reachable from a node

void dfs_visit(int v, int p) // v: Current node, p: Previous node
{
  dfs_num[v] = dfs_low[v] = dfs_cont++; // First visit to the node v
  for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor of v..
  {
    int u = g[v][i];   // Neighbor u. Edge v -> u
    if(dfs_num[u] < 0) // If the node u has not been discovered..
    {
      dfs_visit(u, v);              // Visit node u
      if(dfs_low[u] > dfs_num[v]) // If the only way to reach u is through v
        cout << v << " " << u;     // Edge v -> u is a bridge
      dfs_low[v] = min(dfs_low[v], dfs_low[u]); // Update dfs_low
    }
    else if(u != p) // If the node u is not the predecessor..
      dfs_low[v] = min(dfs_low[v], dfs_num[u]); // Update dfs_low
  }
}
```

```
void dfs()
{
  dfs_cont = 0;          // Counter starts at 0
  dfs_num = vi(n, -1);  // No node has been discovered
  dfs_low = vi(n, INF); // The nodes are unreachable
  for(int i = 0; i < n; i++) // For each node i..
    if(dfs_num[i] < 0)       // If the node i has not been discovered..
      dfs_visit(i, -1);      // Visit the connected component
}
```

ARTICULATION POINTS – UNDIRECTED GRAPHS

```
int dfs_cont; // Counter
vi dfs_num;   // Value of dfs_cont when a node is visited for the first time
vi dfs_low;   // Lowest dfs_num reachable from a node
int dfs_root; // dfs_visit initial node
int children; // Number of neighbors of root node
// Information about Articulation Point (AP)
vb isAP;      // isAP[i] : check if the node is an AP

void dfs_visit(int v, int p) // v: Current node, p: Previous node
{
  dfs_num[v] = dfs_low[v] = dfs_cont++;      // First visit to the node v
  for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor of v..
  {
    int u = g[v][i];   // Neighbor u. Edge v -> u
    if(dfs_num[u] < 0) // If the node u has not been discovered..
    {
      if(v == dfs_root)            // If node v is the root..
        children++;                // Count the number of children
      dfs_visit(u, v);             // Visit node u
      if(dfs_low[u] >= dfs_num[v]) // If the only way to reach u is through v
        isAP[v] = true;            // Node v is an AP
      dfs_low[v] = min(dfs_low[v], dfs_low[u]); // Update dfs_low
    }
    else if(u != p) // If the node u is not the predecessor..
      dfs_low[v] = min(dfs_low[v], dfs_num[u]); // Update dfs_low
  }
}

void dfs()
{
  dfs_cont = 0;          // Counter starts at 0
  dfs_num = vi(n, -1);  // No node has been discovered
  dfs_low = vi(n, INF); // The nodes are unreachable
  isAP = vb(n, false);  // No node is an AP at the beginning
  for(int i = 0; i < n; i++) // For each node i..
    if(dfs_num[i] < 0)       // If the node i has not been discovered..
    {
      dfs_root = i;              // Store information of root node
      children = 0;              // Node starts with 0 children
      dfs_visit(i, -1);          // Visit the connected component
      isAP[i] = (children > 1); // The root is an AP if it has > 1 child
    }
}
```

SPANNING TREES

MINIMUM SPANNING TREE

```
typedef pair<int, int> ii; // Pair   : (Node, Node)
typedef pair<int, ii> iii; // Triple : (Weight, (Node, Node))
// Union-Find
vi pset;
void initSet(int n) { pset = vi(n); for(int i = 0; i < n; i++) pset[i] = i; }
int findSet(int i) { return (pset[i] == i)? i : (pset[i]= findSet(pset[i]));}
void unionSet(int i, int j) { pset[findSet(i)] = findSet(j); }
bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
// Kruskal
priority_queue<iii> edges; // List of edges

ii mst(int n) // n: Number of nodes
{
  int mstSize = 0;  // mst_size: Number of edges of the MST
  int mstCost = 0;  // mst_cost: Weight of the MST
  initSet(n);        // Initialize Union-Find
  while(!edges.empty() && mstSize < n - 1) // If there are edges..
  {
    iii edge = edges.top(); edges.pop(); // Get the lowest-cost edge
    int x = edge.second.first, y = edge.second.second; // Get nodes
    int w = abs(edge.first);                           // Get weight
    if(!isSameSet(x,y)) // If the nodes are not in the same CC..
    {
      unionSet(x,y);  // Connect the nodes
      mstSize++;      // The tree gains one edge
      mstCost += w;   // The weight of the tree increases
    }
  }
  return ii(mstSize, mstCost); // Returns: (Number of edges on MST, Weight)
}
```

To add edges:

```
// Clean the list of edges
while(!edges.empty())
  edges.pop();
// Store edges
cin >> x >> y >> w;
edges.push(iii(-w,ii(x,y))); // Use negative weight
// Run the algorithm
ii ans = mst(n); // If ans.first < n - 1, it is impossible to build a tree
cout << "MST Weight: " << ans.second;
```

MAXIMUM SPANNING TREE

Don't modify edge weight.

```
cin >> x >> y >> w;
edges.push(iii(w,ii(x,y))); // Store a positive weight
```

PARTIAL MINIMUM SPANNING TREE

When there are fixed edges in the MST.

```
ii mst(int n) // n: Number of nodes
{
  // initSet(n); <- Don't initialize Union-Find
}
```

To add edges:

```
// Read the edges
for(int i = 0; i < m; i++) // m: Number of edges
{
  cin >> x >> y >> w;
  edges.push(iii(-w, ii(x,y))); // Store a negative weight
}
// Process fixed edges first
initSet(N); // Initialize Union-Find
for(int i = 0; i < P; i++) // P: Number of fixed edges
{
  cin >> x >> y;
  unionSet(x,y); // Connect the nodes
}
// Then, run the algorithm
ii ans = mst(n);
cout << "MST Weight: " << ans.second;
```

WEIGHTED GRAPHS

IMPLEMENTATION

```
typedef pair<int, int> ii; // Pair: (Weight, Node)
typedef vector<ii> vii;    // Array of pairs
typedef vector<vii> vvii;  // Adjacency list
vvii g;                    // g: Graph
int n = 5;                 // n: Number of nodes
g = vvii(n);               // Creates a new graph with N nodes
```

INSERT EDGE

```
int x = 3, y = 4, w = 2;  // x, y: Node index [0, n-1], w: Weight
g[x].push_back(ii(w, y)); // Inserts directed edge (x → y, weight w)
```

DIJKSTRA - O((E + V) LG V)

```
vi dist; // dist[i]: Distance from source to node i
vi prev; // prev[i]: Parent of node i

void init() // Cleans the arrays
{
  prev = vi(n, -1);  // -1  : The node has no parent
  dist = vi(n, INF); // INF : The node is unreachable
}

void dijkstra(int s) // s: Source node
{
  priority_queue<ii,vector<ii>,greater<ii> > pq; // (Distance to node, node)
  pq.push(ii(0, s)); // Insert: (Distance to source, source)
  dist[s] = 0;       // Distance from source to itself is 0
  while(!pq.empty()) // While the queue has nodes..
  {
    int d = pq.top().first;   // Get distance
    int v = pq.top().second;  // Get node
    pq.pop();                 // Remove node from queue
    if(d <= dist[v])          // If node information from queue is updated..
      for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor of v..
      {
        int u = g[v][i].second;     // Neighbor u. Edge v -> u
        int w = g[v][i].first;      // Weight between v and u
        if(dist[v] + w < dist[u])   // If node u has a better distance..
        {
          dist[u] = dist[v] + w;    // Update distance
          prev[u] = v;              // Update parent
          pq.push(ii(dist[u], u)); // Insert (distance, node) to queue
        }
      }
  }
}
```

DIJKSTRA IN A MATRIX

```
typedef pair<int,int> ii; // Pair: (Node, Node)
typedef pair<int,ii> iii; // Triple: (Distance, (Node, Node))
// 4 Movements: Right, Left, Down, Up
int aux_x[4] = {0, 0, 1, -1};
int aux_y[4] = {1, -1, 0, 0};
int r,c;
int cost[MAX][MAX];
int dist[MAX][MAX]; // Matriz de distancias
ii prev[MAX][MAX];  // Matriz de padres

void init()
{
  for(int i = 0; i < r; i++)
    for(int j = 0; j < c; j++)
    {
      dist[i][j] = INF;
      prev[i][j] = ii(-1,-1);
    }
}

void dijkstra(int sx, int sy)
{
  priority_queue<iii,vector<iii>,greater<iii> > pq;
  pq.push(iii(dist[sx][sy], ii(sx,sy)));
  dist[sx][sy] = g[sx][sy];
  while(!pq.empty())
  {
    int d = pq.top().first;
    int px = pq.top().second.first;
    int py = pq.top().second.second;
    pq.pop();
    if(d <= dist[px][py])
      for(int i = 0; i < 4; i++)
      {
        int nx = px + aux_x[i];
        int ny = py + aux_y[i];
        if(nx > -1 && ny > -1 && nx < r && ny < c)
        {
          int w = cost[nx][ny];
          if(dist[px][py] + w < dist[nx][ny])
          {
            dist[nx][ny] = dist[px][py] + w;
            prev[nx][ny] = ii(px,py);
            pq.push(iii(dist[nx][ny],ii(nx,ny)));
          }
        }
      }
  }
}
```

BELLMAN FORD – O(VE)

Highlighted validations allow to ignore negative cycles that are not reachable from the source node. When removed, the algorithm detects negative cycles even if the graph is disconnected.

```
vi dist; // dist[i]: Distance from source to node i
vi prev; // prev[i]: Parent of node i

bool bellman_ford(int s) // s: Source node
{
  dist[s] = 0; // Distance from source to itself is 0
  for(int k = 0; k < n - 1; k++) // Relax "V - 1" times
    for(int v = 0; v < n; v++)    // For each node v..
      for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor of v..
      {
        int u = g[v][i].second; // Neighbor u. Edge v -> u
        int w = g[v][i].first;  // Weight between v and u
        if(dist[v] != INF && dist[v] + w < dist[u]) // Relax
        {
          dist[u] = dist[v] + w; // Update distance
          prev[u] = v;           // Update parent
        }
      }
  // Relax edges once more to check for negative cycles
  for(int v = 0; v < n; v++) // For each node v..
    for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor of v..
    {
      int u = g[v][i].second; // Neighbor u. Edge v -> u
      int w = g[v][i].first;  // Weight between v and u
      if(dist[u] != INF && dist[u] + w < dist[v]) // Relax
        return true; // If we can still relax edges, there is a neg cycle
    }
  return false; // There is no neg cycle
}
```

BELLMAN FORD – SHORTEST PATH USING AT MOST X EDGES

- Graph must be directed and acyclic.
- Vertex must be processed in reverse topological order.

```
void bellman_ford(int s, int x)
{
  dist[s] = 0;
  for(int k = 0; k <= min(n, x); k++) // Relax "min(V,X)" times
    for(int v = n - 1; v >= 0; v--)   // Process in reverse topological order
      for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor of v..
      {
        int u = g[v][i].second;   // Neighbor u. Edge v -> u
        int w = g[v][i].first;    // Weight of edge v -> u
        if(dist[v] + w < dist[u]) // Relax
          dist[u] = dist[v] + w;  // Update distance
      }
}
```

KARP'S MINIMUM MEAN-WEIGHT CYCLE – DIRECTED GRAPH

- Graph must have capacity for "n+1" nodes.

```
int dist[MAX][MAX]; // dist[v][k]: distance from s to v using exactly k edges

double solve()
{
  // 1. Add new node "s" to the graph.
  // For every node v in the graph, add edge s -> v with weight 0
  int s = n++;
  for(int v = 0; v < n - 1; v++) // For every node v..
    if(!g[v].empty())            // If it has neighbors..
      g[s].push_back(ii(0, v));  // Add a dummy edge from s to v
  // 2. Set distances
  for(int v = 0; v < MAX; v++)
    for(int k = 0; k < MAX; k++)
      dist[v][k] = INF;          // Set all distances to INF
  // 3. Compute dist[v][k] for every node v and all k (1 <= k <= n)
  dist[s][0] = 0;
  for(int k = 1; k <= n; k++)
    for(int v = 0; v < n; v++)
    {
      if(dist[v][k - 1] == INF)
        continue;
      for(int i = (int)g[v].size() - 1; i >= 0; i--)
      {
        int u = g[v][i].first;
        int w = g[v][i].second;
        dist[u][k] = min(dist[u][k], dist[v][k - 1] + w);
      }
    }
  // 4. If dist[v][n] == INF for every node v, the graph is acyclic
  bool acyclic = true;
  for(int i = 0; i < n && acyclic; i++)
    if(dist[i][n] != INF)
      acyclic = false;
  if(acyclic)
    return INF;
  // 5. Find a node v that minimizes the formula:
  //    (dist[v][n] - dist[v][k]) / (n - k)
  double ans = INF; // 1e15
  for(int v = 0; v < n - 1; v++)
  {
    if(dist[v][n] == INF)
      continue;
    double w = -INF; // -1e15
    for(int k = 0; k < n; k++)
      if(dist[v][k] != INF)
        w = max(w, 1.0 * (dist[v][n] - dist[v][k]) / (n - k));
      ans = min(ans, w);
  }
  return ans;
  // To get the cycle, keep track of the node v that yields the lowest ans
}
```

DIRECTED ACYCLIC GRAPHS (DAG)

SHORTEST/LONGEST PATH

```
vi dist; // dist[i]: Distance from source to node i
vi prev; // prev[i]: Parent of node i

int dag_sp(int s) // s: Source node
{
  dist[s] = 0;     // Distance from source to itself is 0
  topsort();       // Process the nodes in topological order
  for(int k = 0; k < (int)sorted.size(); k++)
  {
    int v = sorted[k];
    for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor..
    {
      int u = g[v][i].second;  // Neighbor u
      int w = g[v][i].first;   // Weight between v and u (LP: multiply by -1)
      if(dist[v] + w < dist[u]) // Relax
      {
        dist[u] = dist[v] + w; // Update distance
        prev[u] = v;           // Update parent
      }
    }
  }
}
```

For Longest Paths, dist values must be multiplied by -1:

```
ans = -INF;
for(int i = 0; i < n; i++)
  ans = max(ans, dist[i] * -1);
cout << "Longest path: " << ans << '\n';
```

COUNTING PATHS

```
vi ways; // ways[i]: Number of ways to reach node i

void count()
{
  ways = vi(n, 0);
  ways[sorted[0]] = 1;
  for(int i = 0; i < (int)sorted.size(); i++)
    for(int j = 0; j < (int)g[sorted[i]].size(); j++)
      ways[g[sorted[i]][j]] += ways[sorted[i]];
}
```

COUNTING PATHS IN A MATRIX

```
#define MAX 105
bool B[MAX][MAX]; // B[i][j]: True if cell(i,j) is blocked
int M[MAX][MAX];  // M[i][j]: Number of ways to reach cell(i,j)
```

```
int r,c;

void init()
{
  memset(B, false, sizeof B);
  memset(M, 0, sizeof M);
}

void count()
{
  int nx, ny;
  M[0][0] = 1; // There is only way get to get to the source 1
  for(int i = 0; i < r; i++)
    for(int j = 0; j < c; j++)
    {
      nx = i + 1; ny = j; // 1st Movement (+1, +0)
      if(nx > -1 && ny > -1 && nx < r && ny < c && !B[nx][ny])
        M[nx][ny] += M[i][j];
      nx = i; ny = j + 1; // 2nd Movement (+0, +1)
      if(nx > -1 && ny > -1 && nx < r && ny < c && !B[nx][ny])
        M[nx][ny] += M[i][j];
    }
}
```

En el main:

```
init();
// Don't forget to mark restricted cells as false
count();
cout << "There are" << M[5][5] << " paths from (0, 0) to (5, 5) << '\n';
```

ALL-PAIRS SHORTEST PATH

FLOYD WARSHALL - O(N^3)

```
#define MAX 105
int M[MAX][MAX]; // M[i][j]: Weight from node i to node j
int n;           // n: Number of nodes

void init()
{
  for(int i = 0; i < MAX; i++)
    for(int j = 0; j < MAX; j++)
      M[i][j] = (i == j)? 0 : INF;
}

void floyd()
{
  for(int k = 0; k < n; k++)
    for(int i = 0; i < n; i++)
      for(int j = 0; j < n; j++)
        M[i][j] = min(M[i][j], M[i][k] + M[k][j]);
}
```

TRANSITIVE CLOSURE

After runnning the algorithm, we can determine if two nodes *i* and *j* are connected directly or indirectly, by checking if M[i][j] is true or false.

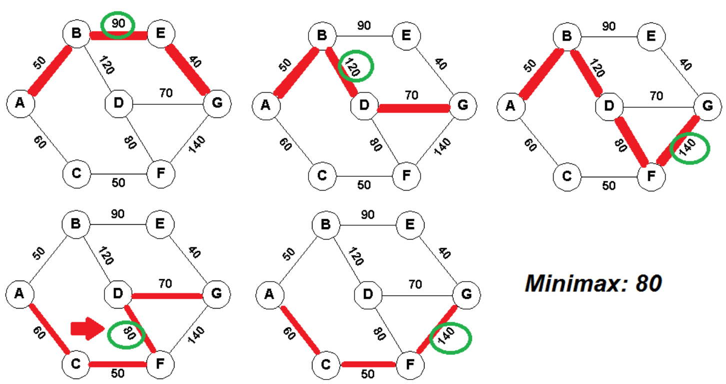
```
bool M[MAX][MAX]; // M[i][j]: true, if nodes i and j are connected
int n;            // n: Number of nodes

void init() { memset(M, false, sizeof M); }

void floyd()
{
  for(int k = 0; k < n ; k++)
    for(int i = 0; i < n; i++)
      for(int j = 0; j < n; j++)
        M[i][j] = M[i][j] | (M[i][k] & M[k][j]);
}
```

MINIMAX

Find the minimum among all the maximum edge weight on all possible paths between two nodes.



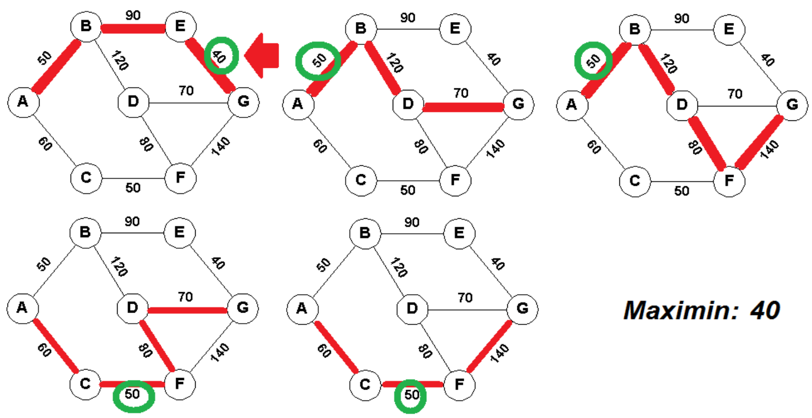

```
const int MAX = 105;
int M[MAX][MAX]; // M[i][j]: weight between node i and j
int n;           // n: Number of nodes

void init()
{
  for(int i = 0; i < MAX; i++)
    for(int j = 0; j < MAX; j++)
      M[i][j] = (i == j)? 0 : INF;
}

void floyd(int N)
{
  for(int k = 0; k < N; k++)
    for(int i = 0; i < N; i++)
      for(int j = 0; j < N; j++)
        M[i][j] = min(M[i][j], max(M[i][k], M[k][j]));
}
```

MAXIMIN

Find the maximum among all the minimum edge weight on all possible paths between two nodes.




```
const int MAX = 105;
int M[MAX][MAX]; // M[i][j]: weight between node i and j
int n;           // n: Number of nodes

void init() { memset(M, 0, sizeof M); }

void floyd()
{
  for(int k = 0; k < N; k++)
    for(int i = 0;i < N; i++)
      for(int j = 0; j < N; j++)
        M[i][j] = max(M[i][j], min(M[i][k], M[k][j]));
}
```

ARBITRAGE PROBLEM

```
#define MAX 105
double M[MAX][MAX]; // M[i][j]: Weight between currency i and j
int n;              // n: Number of nodes

void init()
{
  memset(M, 0, sizeof M);
  for(int i = 0; i < MAX; i++)
    M[i][i] = 1;
}

void floyd()
{
  for(int k = 0; k < n; k++)
    for(int i = 0; i < n; i++)
      for(int j = 0; j < n; j++)
        M[i][j] = max(M[i][j], M[i][k] * M[k][j]);
}
```

To check:

```
bool posible = false;
floyd();
for(int i = 0; i < n; i++)
  if(M[i][i] > 1.0)
    posible = true;
```

NEGATIVE CYCLE DETECTION

Run Floyd and verify if any number in the diagonal is negative.

```
bool hasNegCycle = false;
floyd();
for(int i = 0; i < n; i++)
  if(M[i][i] < 0)
    hasNegCycle = true;
```

PRINT SHORTEST PATH

```
#define MAX 105
int M[MAX][MAX]; // M: Matriz de distancias
int P[MAX][MAX]; // P: Matriz de Parents
int n;

void init()
{
   for(int i = 0; i < N; i++)
      for(int j = 0; j < N; j++)
         P[i][j] = i;        // Initialize Parent Matrix
   for(int i = 0; i < MAX; i++)
      for(int j = 0; j < MAX; j++)
         M[i][j] = (i == j)? 0 : INF; // Initialize Distance Matrix
}

void floyd()
{
   for(int k = 0; k < N; k++)
      for(int i = 0; i < N; i++)
         for(int j = 0; j < N; j++)
            if(M[i][k] + M[k][j] < M[i][j])
            {
               M[i][j] = M[i][k] + M[k][j];
               P[i][j] = P[k][j];
            }
}

void printPath(int i, int j)
{
   if(i != j) printPath(i, P[i][j]);
   printf(" %d", j));
}
```

FLOW NETWORK

MAXIMUM FLOW – DINIC O(V^2 E)

```
#define MAXN 1005     // Maximum number of nodes
#define MAXE 100005   // Maximum number of edges
#define INF 20000000

int E,n,s,t; // n: Number of nodes, s: Source, t: Sink
int dis[MAXN],head[MAXN],work[MAXN];
int cap[MAXE],flow[MAXE],to[MAXE],nxt[MAXE];

void init() { E = 0; memset(head, -1, sizeof head); }

void add(int v, int u, int f)
{
  to[E]=u,cap[E]=f,flow[E]=0,nxt[E]=head[v],head[v]=(E++);
  to[E]=v,cap[E]=0,flow[E]=0,nxt[E]=head[u],head[u]=(E++);
}

bool dinic_bfs()
{
  memset(dis, -1, sizeof dis); dis[s] = 0;
  queue<int> q; q.push(s);
  while(!q.empty())
  {
    int v = q.front(); q.pop();
    for(int e = head[v]; e >= 0; e = nxt[e])
      if(flow[e] < cap[e] && dis[to[e]] < 0)
      {
        dis[to[e]] = dis[v] + 1;
        q.push(to[e]);
      }
  }
  return (dis[t] >= 0);
}

int dinic_dfs(int v, int limit)
{
  if(v == t) return limit;
  for(int &e = work[v]; e >= 0; e = nxt[e])
  {
    int u = to[e], tmp;
    if(flow[e] < cap[e] && dis[u] == dis[v] + 1 &&
        (tmp = dinic_dfs(u, min(limit, cap[e] - flow[e]))) > 0)
    {
      flow[e] += tmp;
      flow[e^1] -= tmp;
      return tmp;
    }
  }
  return 0;
}

int dinic_flow()
```

```
{
  int ans = 0;
  while(dinic_bfs()) // While there is a level graph..
  {
    for(int i = 0; i < n; i++) work[i] = head[i];
    while(1)
    {
      int f = dinic_dfs(s, INF); // Find blocking flow
      if(f == 0) break;          // No more flow to send
      ans += f;                  // Add flow
    }
  }
  return ans;
}
```

Example:

```
// First set n, s and t
init();             // Clean edge information
cin >> x >> y >> f; // Directed edge x -> y with capacity f
add(x, y, f);       // Add the directed edge x -> y with capacity f
cout << "Max flow is = " << dinic_flow(); // Find max flow between s and t
```

FIND OUTGOING FLOW FROM A NODE

```
for(int e = head[x]; e >= 0; e = nxt[e]) // Check every edge e out of x..
  if(flow[e] > 0)                        // If there is flow through edge e..
    cout << "Flow: " << flow[e] << " from " << x << " to " << to[e];
```

MIN-CUT BETWEEN TWO NODES

The min-cut between two nodes is equal to the max-flow. To find the edges that belong to the S-T cut:

```
int vis[MAXN];

void mincut() // After calling dinic_flow
{
  queue<int> q; q.push(s);                // Enqueue source
  memset(vis, 0, sizeof vis); vis[s] = 1; // Mark as visited
  while(!q.empty()) // While queue is not empty..
  {
    int v = q.front(); q.pop();              // Get node v from queue
    for(int e = head[v]; e >= 0; e = nxt[e]) // Check every neighbor of v..
      if(dis[to[e]] != -1 && !vis[to[e]]) // Edge is inside of the S comp.
      {
        q.push(to[e]);  // Enqueue neighbor
        vis[to[e]] = 1; // Mark node u as visited
      }
      else if(dis[to[e]] == -1)     // This edge is part of the cut
        cout << v << " " << to[e]; // Print edge
  }
}
```

GENERAL MINIMUM CUT – STOER WAGNER O(N^3)

```
#define MAX 155
int g[MAX][MAX]; // g[i][j]: Cost of edge i -> j (0 if there is no edge)
int n;           // n: Number of nodes

int minCut()
{
  int V[MAX], W[MAX], best = INF;
  bool A[MAX];
  for(int i = 0; i < n; i++) V[i] = i; // init the remaining vertex set
  while(n > 1)
  {
    A[V[0]] = true; // initialize the set A and vertex weights
    for(int i = 1; i < n; i++) { A[V[i]] = false; W[i] = g[V[0]][V[i]]; }
    int prev = V[0]; // add the other vertices
    for(int i = 1; i < n; i++)
    {
      int pos = -1; // find the most tightly connected non-A vertex
      for(int j = 1; j < n; j++)
        if(!A[V[j]] && (pos < 0 || W[j] > W[pos])) pos = j;
        A[V[pos]] = true; // add it to A
        if(i == n - 1) // last vertex?
        {
          if(W[pos] < best) best = W[pos]; // remember the cut weight
          for(int j = 0; j < n; j++) // merge prev and v[j]
          {
            g[V[j]][prev] += g[V[pos]][V[j]];
            g[prev][V[j]] += g[V[pos]][V[j]];
          }
          V[pos] = V[--n]; break;
        }
        prev = V[pos];
        for(int j = 1; j < n; j++) // Update the weights of its neighbours
          if(!A[V[j]]) W[j] += g[V[pos]][V[j]];
    }
  }
  return best;
}
```

MIN-COST MAX-FLOW

```
#define MAXN 505
#define MAXE 500005
typedef int capt; // Capacity type
typedef int cstt; // Cost type
const cstt INF = 20000000;
typedef pair<cstt,int> ii;

int E,n; // n: Number of nodes (must be set)
int par[MAXN],head[MAXN],vis[MAXN],to[MAXE],nxt[MAXE];
capt flow,cap[MAXE];
cstt cost,dis[MAXN],cst[MAXE],pot[MAXN];
```

```
void init() { E = 0; memset(head, -1, sizeof head); }

void add_edge(int v, int u, capt f, cstt c)
{
  to[E]=u,cap[E]=f,cst[E]= c,nxt[E]=head[v],head[v]=(E++);
  to[E]=v,cap[E]=0,cst[E]=-c,nxt[E]=head[u],head[u]=(E++);
}

void mcmf(int s, int t)
{
  flow = cost = 0;
  memset(pot, 0, sizeof pot);
  while(true)
  {
    memset(par, -1, sizeof par);
    memset(vis, 0, sizeof vis);
    for(int i = 0; i < n; i++)
      dis[i] = INF;
    priority_queue<ii> q; q.push(ii(0, s));
    dis[s] = par[s] = 0;
    vis[s] = 1;
    while(!q.empty())
    {
      int v = q.top().second; q.pop();
      for(int e = head[v]; e != -1; e = nxt[e])
        if(cap[e] > 0)
        {
          int u = to[e];
          cstt d = dis[v] + cst[e] + pot[v] - pot[u];
          if(!vis[u] && d < dis[u])
          {
            vis[u] = 1;
            dis[u] = d;
            par[u] = e;
            q.push(ii(-d, u));
          }
        }
    }
    if(par[t] == -1)
      break;
    capt f = cap[par[t]];
    for(int i = t; i != s; i = to[par[i]^1])
      f = min(f, cap[par[i]]);
    for(int i = t; i != s; i = to[par[i]^1])
      cap[par[i]] -= f, cap[par[i]^1] += f;
    flow += f;
    cost += f * (dis[t] - pot[s] + pot[t]);
    for(int i = 0; i < n; i++)
      if(par[i] != -1)
        pot[i] += dis[i];
  }
}
```

After calling mcmf, the answer will be in the globar variables: **flow** and **cost**.

MAX CARDINALITY BIPARTITE MATCHING – O(V^2 + VE)

```
vi match;
vb visit;

int augment(int v)
{
  if(visit[v])
    return 0;
  visit[v] = true;
  for(int i = 0; i < (int)g[v].size(); i++)
  {
    int u = g[v][i];
    if(match[u] == -1 || augment(match[u]))
    {
      match[u] = v;
      return 1;
    }
  }
  return 0;
}

int mcbm(int left) // left: Num of vertex on left
{
  int ans = 0;
  match = vi(n, -1);
  for(int i = 0; i < left; i++)
  {
    visited = vb(left, false);
    ans += augment(i);
  }
  return ans;
}
```

MAX WEIGHTED MATCHING ON A GENERAL GRAPH (EDMONDS BLOSSOM) – O(V^3)

```
#define MAX 105
#define INF 2000000000

int match[MAX], visited[MAX];

bool dfs(int node, vector<vector<int> > &adj, vector<int> &blossom)
{
  int n = (int)adj.size();
  visited[node] = 0;
  for(int i = 0; i < n; i++)
    if(adj[node][i]) {
      if(visited[i] == -1) {
        visited[i] = 1;
        if(match[i] == -1 || dfs(match[i], adj, blossom)) {
          match[node] = i; match[i] = node;
          return true;
        }
      }
```

```
      if(visited[i] == 0 || !blossom.empty()) {
        blossom.push_back(i); blossom.push_back(node);
        if(node == blossom[0]) {
          match[node] = -1;
          return true;
        }
        return false;
      }
   }
   return false;
}

bool augmentingPath(vector<vector<int> > &adj)
{
  int n = (int)adj.size();
  for(int m = 0; m < n; m++)
    if(match[m] == -1) {
      vector<int> blossom;
      memset(visited, -1, sizeof visited);
      if(!dfs(m, adj, blossom)) continue;
      if(blossom.empty()) return true;
      int base = blossom[0];
      vector<vector<int> > newadj = adj;
      for(int i = 1; i < (int)blossom.size() - 1; i++)
        for(int j = 0; j < n; j++)
          newadj[base][j] = newadj[j][base] |= adj[blossom[i]][j];
      for(int i = 1; i < (int)blossom.size() - 1; i++)
        for(int j = 0; j < n; j++)
          newadj[blossom[i]][j] = newadj[j][blossom[i]] = 0;
      newadj[base][base] = 0;
      if(!augmentingPath(newadj)) return false;
      int k = match[base];
      if(k != -1)
        for(int i = 0; i < (int) blossom.size(); i++)
          if(adj[blossom[i]][k]) {
            match[blossom[i]] = k;
            match[k] = blossom[i];
            if(i & 1) {
              for(int j = i + 1; j < (int)blossom.size(); j += 2) {
                match[blossom[j]] = blossom[j+1];
                match[blossom[j+1]] = blossom[j];
              }
            }
            else {
              for(int j = 0; j < i; j += 2) {
                match[blossom[j]] = blossom[j+1];
                match[blossom[j+1]] = blossom[j];
              }
            }
            break;
          }
          return true;
  }
  return false;
}
```

```
// Esta funcion devuelve la cantidad de matchings que pude hacer
int edmondsBlossom(vector<vector<int> > &adj)
{
  int ans = 0;
  memset(match, -1, sizeof match);
  while(augmentingPath(adj)) ans++;
  return ans;
}
```

En el main:

```
vector<vector<int> > g; // Matriz de adyacencia
int N;
// 1) g[i][j] contiene el costo de unir i y j (0 si no es posible unirlos)
// 2) La matriz g debe tener INF en la diagonal
g.clear(); g.resize(N);
for(int i = 0; i < N; i++)
{
   g[i].clear(); g[i].resize(N); fill(g[i].begin(),g[i].end(), INF);
}
// Leer matriz de adyacencia y luego aplicar el algoritmo
int matchings = edmondsBlossom(g);
printf("Se pudieron hacer %d conexiones\n",matchings);
```

Para verificar los emparejamientos:

```
printf("Las conexiones son:\n");
for(int x = 0; x < N; x++)
{
   int y = match[x]; // Vemos con quien se agrupa el nodo x
   printf("(%d -> %d)\n",x,y);
}
```

EULER GRAPHS

EULER TOUR

```
typedef pair<int,int> ii; // first: nodo, second: 1 = used, 0 = not used

typedef struct {
   vector<vector<ii> > edges;
}graph;

void initialize_graph(graph *g, int n)
{
   g->edges.clear(); g->edges.resize(n);
   for(int i = 0; i < n; i++) g->edges[i].clear();
}

void insert_edge(graph *g, int x, int y, bool directed)
{
   g->edges[x].push_back(ii(y, 1));
   if(!directed) g->edges[y].push_back(ii(x, 1));
}
```

```
list<int> cyc;

void eulerTour(list<int>::iterator i, int u, graph *g)
{
   for(int j = 0; j < (int)g->edges[u].size(); j++)
   {
      ii v = g->edges[u][j];
      if(v.second)
      {
         g->edges[u][j].second = 0;
         for(int k = 0; k < (int)g->edges[v.first].size(); k++)
         {
            ii uu = g->edges[v.first][k];
            if(uu.first == u && uu.second)
            {
               g->edges[v.first][k].second = 0; break;
            }
         }
         eulerTour(cyc.insert(i, u), v.first, g);
      }
   }
}
```

En el main

```
initialize_graph(&g, N); // N: Cantidad de nodos
memset(degree, 0, sizeof degree);
for(int i = 0; i < M; i++) // M: Cantidad de aristas
{
   insert_edge(&g, x, y, false); // x, y: Nodos unidos por la arista i
   degree[x]++; degree[y]++;
}
// Verificar si el grafo tiene un euler tour
posible = true;
for(int i = 0; i < N && posible; i++)
   if(degree[i] % 2 != 0)
      posible = false;

if(!posible) printf("No hay tour de euler\n");
else
{
   // Hallar tour de Euler
   cyc.clear(); eulerTour(cyc.begin(), x, &g); sz = 0;
   for(list<int>::iterator it = cyc.begin(); it != cyc.end(); it++)
      ans[sz++] = (*it) + 1;
   // Imprimir respuesta
   for(int i = 0; i < N; i++) printf("%d\n",ans[i]);
}
```

TREES

LOWEST COMMON ANCESTOR

PRE-PROCESS O(N LG N), QUERY O(LG N)

```
const int MAX = 100005;
int n;          // n: Number of nodes
int P[MAX][17]; // P[i][j] : Parent of node i at distance 2^j.
int H[MAX];     // H[i]    : Height of node i.
int W[MAX];     // W[i]    : Distance from node i to root (weighted tree).

void build()
{
  for(int j = 1; 1 << j < n; j++)
    for(int i = 0; i < n; i++)
      if(P[i][j - 1] != -1)
        P[i][j] = P[P[i][j - 1]][j - 1];
}

int lca(int p, int q) // p: Node, q: Node
{
  if(H[p] < H[q]) swap(p, q);
  int lg = 1;
  while(1 << lg <= H[p]) lg++; lg--;
  for(int i = lg; i >= 0; i--)
    if(H[p] - H[q] >= (1 << i))
      p = P[p][i];
  if(p == q)
    return p;
  for(int i = lg; i >= 0; i--)
    if(P[p][i] != -1 && P[p][i] != P[q][i])
      p = P[p][i], q = P[q][i];
  return P[p][0];
}
```

From a graph g, run dfs(*v*) from a **random** node *v* to initialize the parameters:

```
// Call memset(P, -1, sizeof P) before calling dfs
// v: Node, p: Parent of v, h: Accrued Height, s: Accrued Weight
void dfs(int v, int p = -1, int h = 0, int s = 0)
{
  H[v] = h;     // Assign height
  W[v] = s;     // Assign weight (In weighted trees)
  P[v][0] = p; // Assign parent
  for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor..
  {
    int u = g[v][i].second; // Neighbor u
    int w = g[v][i].first;  // Weight between v and u (In weighted trees)
    if(u != p) // If node u is different from predecessor..
      dfs(u, v, h + 1, s + w); // Visit node u
   }
}
```

DISTANCE BETWEEN TWO NODES

```
int dist(int p, int q)
{
  return W[p] + W[q] - 2 * W[lca(p,q)]; // For unweighted trees, use H
}
```

GET KTH NODE ON THE PATH FROM P TO Q

```
int getKth(int p, int q, int k) // p: Node, q: Node, k: Position (0-indexed)
{
  int r = lca(p, q);  // The path ascends (p to r), then descends (r to q)
  if(k > H[p] - H[r]) // If the node is in the descending part
  {
    k = H[p] + H[q] - 2 * H[r] - k; // Reverse the index: Path Lenght - K
    swap(p, q);                     // Swap the nodes
  }
  int lg = 1;                        // lg: Largest x that makes 2^x <= H[p]
  while(1 << lg <= H[p]) lg++; lg--; // Find lg
  for(int i = lg; i >= 0; i--) // For each exponent i..
    if(1 << i <= k)  // If 2^i still covers the position K..
    {
      p = P[p][i];  // Go to parent of p at distance 2^i
      k -= 1 << i;  // Update index
    }
  return p; // Return node
}
```

OBTENER EL MAX-EDGE-WEIGHT ENTRE DOS NODOS

La mayoría de funciones son variaciones de la sección anterior

```
#define INF 20000000

int N, T[MAXN], P[MAXN][MAXLOGN], L[MAXN];
long long W[MAXN];
int maxi[MAXN][MAXLOGN]; // maxi es para max-edge weight

void initialize()
{
   for(int i = 0; i < N; i++)
      for(int j = 0; 1 << j < N; j++)
         maxi[i][j] = -INF;
}

void initialize_LCA()
{
   for(int i = 0; i < N; i++)
      for(int j = 0; 1 << j < N; j++)
         P[i][j] = -1;
   for(int i = 0; i < N; i++) P[i][0] = T[i];
   for(int j = 1; 1 << j < N; j++)
      for(int i = 0; i < N; i++)
```

```
         if(P[i][j - 1] != -1)
         {
            P[i][j] = P[P[i][j - 1]][j - 1];
            maxi[i][j] = max(maxi[P[i][j - 1]][j - 1], maxi[i][j - 1]);
         }
}

int getMaxEdge(int p, int q) // Gets the max edge in the path from p to q
{
   int rmaxi = -INF, lg;
   if(L[p] < L[q]) p ^= q ^= p ^= q;
   for(lg = 1; 1 << lg <= L[p]; lg++); lg--;
   for(int i = lg; i >= 0; i--)
      if(L[p] - (1 << i) >= L[q])
      {
         rmaxi = max(rmaxi, maxi[p][i]); p = P[p][i];
      }
   if(p == q) return rmaxi;
   for(int i = lg; i >= 0; i--)
      if(P[p][i] != -1 && P[p][i] != P[q][i])
      {
         rmaxi = max(rmaxi, maxi[p][i]);
         rmaxi = max(rmaxi, maxi[q][i]);
         p = P[p][i]; q = P[q][i];
      }
   rmaxi = max(rmaxi, maxi[p][0]);
   rmaxi = max(rmaxi, maxi[q][0]);
   return rmaxi;
}

vector<int> discovered;

void build_LCA_tree(graph *g, int root)
{
   discovered.clear(); discovered.resize(N);
   fill(discovered.begin(), discovered.end(), false);
   queue<int> q; q.push(root); discovered[root] = true;
   T[root] = -1; L[root] = 0; W[root] = 0;
   maxi[root][0] = 0; // Esta es la linea que varia
   while(!q.empty())
   {
      int u = q.front(); q.pop();
      for(int i = 0; i < (int)g->edges[u].size(); i++)
      {
         int v = g->edges[u][i].second, w = g->edges[u][i].first;
         if(!discovered[v])
         {
            q.push(v); discovered[v] = true;
            T[v] = u;
            L[v] = L[u] + 1;
            W[v] = W[u] + w;
            maxi[v][0] = w; // w minuscula
         }
      }
   }
}
```

En el main:

```
initialize();
build_LCA_tree(&g, edge.second.first);
initialize_LCA();
scanf("%d %d\n",&x,&y); x--; y--;
printf("%d\n",getMaxEdge(x,y)); // Responder Query
```

OPERATIONS ON TREES

FIND TREE ROOT – DIRECTED GRAPH

```
int root()
{
  // Count each node indegree
  vi indeg(n, 0);
  for(int v = 0; v < n; v++) // For every node v..
    for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor..
    {
      int u = g[v][i]; // Neighbor u
      indeg[u]++;      // Increase indegree of node u by one
    }
  // Count number of nodes with indegree 0 and 1
  int cont0 = 0, cont1 = 0, pos;
  for(int i = 0; i < n; i++)
    if(indeg[i] == 0)
      cont0++, pos = i;
    else if(indeg[i] == 1)
      cont1++;
  // Check tree topology
  return (cont0 == 1 && cont1 == n - 1)? pos : -1;
}
```

VERIFY TREE TOPOLOGY – DIRECTED GRAPH

```
bool isTree()
{
  if(n == 0) return true;   // Empty graph is a tree
  int r = root();           // Find root
  if(r == -1) return false; // If graph has not a root, it not a tree
  color = vi(n, 0); // Initialize each node color
  dfs_visit(r);     // DFS from the root
  for(int i = 0; i < n; i++) // For each node i..
    if(!color[i])   // If node i is not reachable..
      return false; // Graph is not a tree
  return true; // Graph is a tree
}
```

VERIFY TREE TOPOLOGY – UNDIRECTED GRAPH

```
vb disc;
```

```
void dfs(int v, int p) // v: Current node, p: Parent of node v
{
  disc[v] = true; // Mark node v as discovered
  for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor..
  {
    int u = g[v][i]; // Neighbor u
    if(!disc[u]) // If node u is not discovered..
      dfs(u, v); // Visit node u
    else if(u != p) // Otherwise..
      return false; // The graph is not a tree (there is a cycle)
  }
}

bool isTree() // Verify if graph is a tree
{
  disc = vb(n, false); // Mark nodes as undiscovered
  for(int i = 0; i < n; i++)   // For each node i..
    if(!disc[i] && !dfs(i,-1)) // If subgraph rooted at i is not a tree
      return false;            // Graph is not a tree
  return true; // Graph is a tree
}
```

MIN VERTEX COVER – DIRECTED TREE

```
int dp[MAX][2];

int solve(int v, int used) // v: Current node, used: is node v in the cover?
{
  if(g[v].empty()) return used;             // Leaf node
  if(dp[v][used] != -1) return dp[v][used]; // Memoization

  int ans = used; // Count node v if it is in the set
  for(int i = 0; i < (int)g[v].size(); i++) // For each neighbor..
  {
    int u = g[v][i]; // Neighbor u
    if(used) // If node v is in the set..
      ans = ans + min(solve(u,0), solve(u,1)); // Node u is optional
    else     // Otherwise..
      ans = ans + solve(u,1); // Node u must be in the set
  }
  return dp[v][flag] = ans;
}

int mvc(int r = 0) // r: Root node
{
  memset(dp, -1, sizeof dp);          // Cleand DP states
  return min(solve(r,0), solve(r, 1); // Root can be taken or ignored
}
```

BINARY SEARCH TREE

NODE STRUCTURE

```
struct node
{
  node *l, *r;
  int val;
  node() { l = NULL; r = NULL; }
  node(int pval) { val = pval; l = NULL; r = NULL; }
};
```

POSTORDER

```
void postorder_visit(node *x)
{
  if(x == NULL) return;
  postorder_visit(x->l);
  postorder_visit(x->r);
  cout << " " << x->val;
}

void postorder(node *root)
{
   postorder_visit(root);
}
```

RECONSTRUCT TREE FROM INORDER AND PREORDER

```
int hash[256];

void mapToIndices(int inorder[], int n)
{
   for(int i = 0; i < n; i++) hash[inorder[i]] = i;
}

Node* buildInorderPreorderVisit(int in[], int pre[], int n, int offset)
{
   if(n == 0) return NULL;
   int rootVal = pre[0];           // El 1r elemento del preorden es la raiz
   int i = hash[rootVal] - offset; // Posicion donde aparece en el inorder
   node *root = new Node(rootVal); // Crear nodo
   root->left  = buildInorderPreorderVisit(in, pre + 1, i, offset);
   root->right = buildInorderPreorderVisit(in + i + 1, pre + i + 1,
                                          n - i - 1, offset + i + 1);
   return root;
}

Node* buildInorderPreorder(int in[], int pre[], int n) // n: Cant elem
{
   mapToIndices(in, n);
   return buildInorderPreorderVisit(in, pre, n, 0);
}
```

RECONSTRUCT TREE FROM INORDER AND POSTORDER

Usar la función ***mapToIndices*** de la sección anterior.

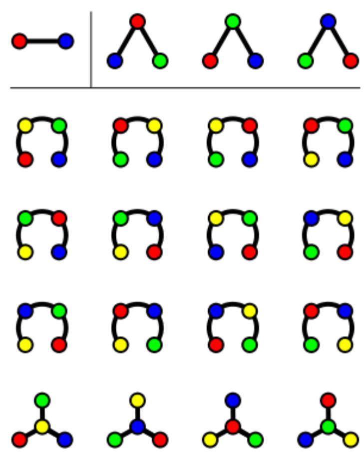
```
node* buildInorderPostorderVisit(int in[], int post[], int n, int offset)
{
   if(n == 0) return NULL;
   int rootVal = post[n - 1];       // El 1r elemento del preorden es la raiz
   int i = hash[rootVal] - offset; // Posicion donde aparece en el inorder
   Node *root = new Node(rootVal); // Crear nodo
   root->left = buildInorderPostorderVisit(in, post, i, offset);
   root->right = buildInorderPostorderVisit(in + i + 1, post + i,
                                            n - i - 1, offset + i + 1);
   return root;
}

node* buildInorderPostorder(int in[], int pre[], int n) // n: Cant elem
{
   mapToIndices(in, n);
   return buildInorderPostorderVisit(in, pre, n, 0);
}
```

PROPERTIES

NUMBER OF TREES USING N NODES

Cayley's Formula: N^{N-2}



STRINGS

STL

FIND AND REPLACE

```
string s = "The frog jumps over the frog", t = "frog";
size_t pos = s.find(t); // pos = 4 (when not found, string::npos is returned)
s.replace(s.find(t), t.length(), "bee"); // s = "The bee jumps over the frog"
```

FIND_FIRST_OF

```
string s = "Replace the vowels in this sentence by asterisks.";
size_t pos = s.find_first_of("aeiou");
while(pos != string::npos)
{
  s[pos] = '*';
  pos = s.find_first_of("aeiou", pos + 1);
}
```

INSERT

```
string s = "to be question", a = "the ", b = "or not to be";
string::iterator it;
s.insert(6, a);                // to be (the )question
s.insert(6, b, 3, 4);          // to be (not )the question
s.insert(10,"that is old",8); // to be not (that is )the question
s.insert(10,"to be ");         // to be not (to be )that is the question
s.insert(15, 1, ':');          // to be not to be(:) that is the question
it=s.insert(s.begin()+5,','); // to be(,) not to be: that is the question
s.insert(s.end(),3,'.');       // to be, not to be: that is the question(...)
s.insert(it+2,b.begin(),b.begin()+3);
                               // to be, or not to be: that is the question...
```

SUBSTR

```
string s = "We think in generalities, but we live in details.";
string a = s.substr(12,12); // "generalities"
string b = s.substr(12);    // "generalities, but we live in details."
```

ERASE

```
string s = "This is an example sentence."; // "This is an example sentence."
s.erase(10,8);                          //            ^^^^^^^^
                                        // "This is an sentence."
s.erase(s.begin()+9);                   //           ^
                                        // "This is a sentence."
s.erase(s.begin()+5, s.end()-9);        //       ^^^^^
                                        // "This sentence."
```

HASHING

```
typedef unsigned long long hasht; // hash is a reserved word in C++11
const int MAX = 50005;  // Size of the largest string to hash
hasht C = 5381, K = 33; // Hashing parameters
hasht pw[MAX], H[MAX];  // pw[i]: K^i, H[i]: Hash of substr [0, i-1]

void init() // Initializes powers of K
{
  pw[0] = 1; // K^0
  for(int i = 1; i < MAX; i++) // For each power i..
    pw[i] = K * pw[i-1];       // K^i = K x K^(i-1)
}

void build(const string &s, int n) // s: String to hash, n: Length of s
{
  H[0] = C;
  for(int i = 1; i <= n; i++)
    H[i] = H[i-1] * K + (s[i-1] - 'a' + 1); // (lower-case letters)
}

hasht calc(int l, int r) // Hash of the substring [l, r-1]
{
  return H[r] - H[l] * pw[r - l];
}
```

HASH TABLE

```
const int MAXN = 100005 // MAXN: Max number of elements in the hash table
vector<pair<hasht,string> > ht[MAXN]; // Hash table <string, string>

void insert(hasht k, string v) // Inserts a hash and its associated value
{
  int idx = k % MAXN;
  ht[idx].push_back(make_pair(k,v));
}

string get(hasht k) // Returns the associated value of a hash
{
  int idx = k % MAXN;
  for(int i = 0; i < ht[idx].size(); i++)
    if(ht[idx][i].first == k)
      return ht[idx][i].second;
  return "eh"; // Return a value that indicate inexistency
}

void update(hasht k, string v) // Updates the associated value of a hash
{
  int idx = k % MAXN;
  for(int i = 0; i < ht[idx].size(); i++)
    if(ht[idx][i].first == k)
      ht[idx][i].second = v;
}
```

```
void clear() // Cleans the hash table
{
  for(int i = 0; i < MAXN; i++)
    ht[i].clear();
}
```

Example:

```
string s = "home", t = "casa"; // s: Key String, t: Associated value of s
int n = s.size();        // n: Length of key string
build(s, n);             // Hash the key string
insert(calc(0, n), t);   // Inserts hashed key and its associated value
cout << get(calc(0, n)); // Retrieves value associated to a hash
```

KMP

Finds all the ocurrences of a string T within a string S in **O(|S| + |T|)**.

```
vi kmp(const string &s, const string &t) // Find t within s
{
  // Find borders of the string t
  vi b(t.size() + 1, -1);
  for(int i = 1; i <= t.size(); i++)
  {
    int p = b[i - 1];
    while(p != -1 && t[p] != t[i - 1])
      p = b[p];
    b[i] = p + 1;
  }
  // Find matches
  vi ans;
  for(int i = 0, p = 0; i < s.size(); i++)
  {
    while(p != -1 && (p == t.size() || t[p] != s[i]))
      p = b[p];
    p++;
    if(p == t.size())
      ans.push_back(i + 1 - t.size());
  }
  return ans;
}
```

STRING COMPRESSION

Find the shortest substring *T* of a string *S*, such that *S* is the concatenation of one or more copies of *T*.

"*abcabcabcabc*" = "*abc*" + "*abc*" + "*abc*" + "*abc*". The substring "*abc*" has length **3** and it is repeated **4** times.

```
string s = "abcabcabcabc";          // s: String to compress
int n = (int)s.size();              // n: Length of s
int r = kmp(s + s, s).size();       // r: Number of ocurrences of S in S + S
cout << "Length: " << n / (r - 1); // Substring has length |S| / (r - 1)
cout << "Frequency: " << r - 1;     // Substring is repeated r - 1 times in S
```

Z ALGORITHM

Given a string S (length n), the algorithm produces an array Z in **O(n)**, where Z[i] = length of the longest substring starting from S[i] which is also a prefix of S. If i + Z[i] == n, then the suffix starting from S[i] is also a prefix.

idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[i]	f	i	x	p	r	e	f	i	x	s	u	f	f	i	x
Z[i]	15	0	0	0	0	0	3	0	0	0	0	1	3	0	0

```
vi z;

void ZFun(const string &s, int n)
{
  z = vi(n, 0);
  z[0] = n;      // z[0] is undefined
  for(int i = 1, L = 0, R = 0; i < n; i++)
  {
    if(i <= R)
      z[i] = min(R - i + 1, z[i - L]);
    while(i + z[i] < n && s[z[i]] == s[i + z[i]])
      z[i]++;
    if(i + z[i] - 1 > R)
      L = i,  R = i + z[i] - 1;
  }
}
```

STRING MATCHING

To find all the ocurrences of a string T within a string S in **O(|S| + |T|)**, apply Z algorithm on *T* + "$" + *S*. Every index i after the "$" where Z[*i*] = |T| indicates an occurrence of *T* in *S*. Example: S = "*casaca*", T = "*ca*",

idx	0	1	2	3	4	5	6	7	8
S[i]	c	a	$	c	a	s	a	c	a
Z[i]	9	0	0	2	0	0	0	2	0

STRING COMPRESSION

To find the shortest substring *T* of a string *S*, such that *S* is the concatenation of one or more copies of *T*, find the smallest index i, such that i + Z[i] == |S|.

"*abcabcabcabc*" = "*abc*" + "*abc*" + "*abc*" + "*abc*". The substring "*abc*" has length **3** and it is repeated **4** times.

idx	0	1	2	3	4	5	6	7	8	9	10	11
S[i]	a	b	c	a	b	c	a	b	c	a	b	c
Z[i]	12	0	0	9	0	0	6	0	0	3	0	0

- T has length *i*.
- T is repetead |S| / *i* times within S.

LONGEST COMMON SUBSEQUENCE

```
const int MAX = 1005;
int dp[MAX][MAX];

int solve(const string &s, int n, const string &t, int m) // n = |s|, m = |t|
{
  // Base case
  for(int i = 0; i <= n; i++) dp[i][0] = 0;
  for(int i = 0; i <= m; i++) dp[0][i] = 0;
  // General case
  for(int i = 1; i <= n; i++)
    for(int j = 1; j <= m; j++)
      if(s[i-1] == t[j-1])
        dp[i][j] = dp[i-1][j-1] + 1;
      else
        dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
   return dp[n][m]; // Size of the LCS
}
```

RECONSTRUCT LCS

```
string ans; // Limpiar antes de llamar a reconstruct

// Llamar a rec(s, t, |s|, |t|)
void rec(const string &s, const string &t, int i, int j)
{
  if(i == 0 || j == 0)
    return;
  if(s[i-1] == t[j-1])
    rec(s, t, i-1, j-1), ans.append(1, s[i-1]);
  else if(dp[i][j-1] > dp[i-1][j])
    rec(s, t, i, j-1);
  else
    rec(s, t, i-1, j);
}
```

EDIT DISTANCE

```
const int MAX = 2005;
int dp[MAX][MAX];

int solve(const string &a, const string &b)
{
  int n = a.size(), m = b.size();
  // Base cases: When s or t are empty
  for(int i = 0; i <= n; i++) dp[i][0] = i;
  for(int j = 0; j <= m; j++) dp[0][j] = j;
  // General case
  for(int i = 1; i <= n; i++)
    for(int j = 1; j <= m; j++)
      if(a[i-1] == b[j-1])
        dp[i][j] = dp[i-1][j-1];
      else
```

```
          dp[i][j] = 1 + min(dp[i-1][j-1], min(dp[i-1][j], dp[i][j-1]));
    return dp[n][m];
}
```

RECONSTRUIR RESPUESTA

```
#define INS 1
#define DEL 2
#define REP 3
#define MAT 4
int costIns = 1, costDel = 1, costRep = 1; // Cost to insert/delete/replace
int const MAX = 1005;
int dp[MAX][MAX];
int rec[MAX][MAX];

int solve(const string &s, int n, const string &t, int m) // n = |S|, m = |T|
{
  // Base cases: When s or t are empty
  for(int i = 0; i <= n; i++) { dp[i][0] = i; rec[i][0] = DEL; }
  for(int i = 0; i <= m; i++) { dp[0][i] = i; rec[0][i] = INS; }
  // General case
  for(int i = 1; i <= n; i++)
    for(int j = 1; j <= m; j++)
      if(s[i-1] == t[j-1]) // Match
      {
        dp[i][j] = dp[i-1][j-1];
        rec[i][j] = MAT;
      }
      else
      {
        int cDel = dp[i-1][j] + costDel;
        int cIns = dp[i][j-1] + costIns;
        int cRep = dp[i-1][j-1] + costRep;
        dp[i][j] = min(cDelete, min(cInsert, cReplace));
        if(cDel <= cIns && cDel <= cRep) rec[i][j] = DEL;
        if(cIns <= cDel && cIns <= cRep) rec[i][j] = INS;
        if(cRep <= cIns && cRep <= cDel) rec[i][j] = REP;
      }
   return dp[n][m];
}

// Llamar a print(|s|, |t|, s, t)
void print(int i, int j, const string &s, const string &m)
{
  if(i == 0 && j == 0)
    return;
  if(rec[i][j] == MAT)
    print(i-1, j-1, s, t);
  if(rec[i][j] == REP)
    print(i-1, j-1, s, t), cout << "Rep at " << j << "to letter " << B[j-1];
  if(rec[i][j] == INS)
    print(  i, j-1, s, t), cout << "Ins at " << j << "letter " << B[j-1];
  if(rec[i][j] == DEL)
    print(i-1,   j, s, t), cout << "Del at " << j+1;
}
```

PALINDROMES

LONGEST PALINDROMIC SUBSTRING – ALGORITMO DE MANACHER O(N)

```
// Transform S into T. For example, S = "abba", T = "^#a#b#b#a#$".
// ^ and $ signs are sentinels appended to each end to avoid bounds checking
string preProcess(string s)
{
   if(s.length() == 0) return "^$";
   string ret = "^";
   for(int i = 0; i < s.length(); i++) ret += "#" + s.substr(i, 1);
   ret += "#$";
   return ret;
}

string longestPalindrome(string s)
{
   string T = preProcess(s);
   int n = T.length(), C = 0, R = 0; int *P = new int[n];
   for(int i = 1; i < n - 1; i++)
   {
      int i_mirror = 2 * C - i; // equals to i' = C - (i - C)
      P[i] = (R > i)? min(R - i, P[i_mirror]) : 0;
      // Attempt to expand palindrome centered at i
      while(T[i + 1 + P[i]] == T[i - 1 - P[i]]) P[i]++;
      // If palindrome centered at i expand past R,
      // adjust center based on expanded palindrome.
      if(i + P[i] > R) { C = i; R = i + P[i]; }
   }
   int maxLen = 0, centerIndex = 0;
   for(int i = 1; i < n - 1; i++)
      if(P[i] > maxLen) { maxLen = P[i]; centerIndex = i; }
   delete[] P;
   return s.substr((centerIndex - 1 - maxLen) / 2, maxLen);
}
```

LONGEST PALINDROMIC SUBSTRING – ALGORITMO DE MANACHER O(N)

```
// Returns half of length of largest palindrome centered at
every position in the string
vector<int> manacher(string s)
{
   vector<int> ans((int)s.size(), 0); int maxi = 0;
   for(int i = 1; i < (int)s.size(); i++)
   {
      int k = 0;
      if(maxi + ans[maxi] >= i) k = min(ans[maxi] + maxi - i, ans[2 * maxi - i]);
      for(; s[i + k] == s[i - k] && i - k >= 0 && i + k < (int)s.size(); k++);
      ans[i] = k - 1;
      if(i + ans[i] > maxi + ans[maxi]) maxi = i;
   }
   return ans;
}
```

TRIES

```
const int MAX = 100005; // Max number of nodes

struct Trie
{
  int g[MAX][26];
  int n; // n: Last node

  Trie() { clear(); }

  void clear() // Clear the trie
  {
    n = 0; // Initial node is root with index 0
    memset(g[0], -1, sizeof g[0]); // Initialize neighbors of the root
  }

  void insert(const string &s) // s: String to insert in the trie
  {
    int cur = 0; // Start at the root
    for(int i = 0; i < (int)s.size(); i++) // For each character in s..
    {
      int c = s[i] - 'a'; // Get letter s[i]
      if(g[cur][c] == -1) // If the neighbor for letter c does not exist..
      {
        g[cur][c] = ++n;  // Assign a new node
        memset(g[n], -1, sizeof g[n]); // Initialize neighbors of new node
      }
      cur = g[cur][c]; // Go to next node
    }
  }
};
```

TRIES (OLD)

```
const int ALPH_SIZE = 26; // Tamaño del alfabeto

struct Node
{
   int words;    // Num de palabras que terminan en el nodo
   int prefixes; // Num de palabras que tienen como prefijo el camino al nodo
   int hijos;    // Numero de bifurcaciones que salen del nodo
   int reachableWords; // Num de palabras a las que puedo llegar desde este nodo
   vector<Node*> links; // Enlaces a los nodos hijos
   Node();
};

Node::Node()
{
   words = prefixes = hijos = reachableWords = 0;
   links.resize(ALPH_SIZE, NULL);
}

class Trie
```

```
{
   public:
      Trie();
      bool contains(const string& s) const;
      int nodeCount() const;
      int countWords(const string& s) const;
      int countPrefixes(const string& s) const;
      int countRepeated() const;
      void printAllWords() const;
      void insert(const string s);
      void dfs();
   private:
      Node* myRoot; // Raíz del trie
      int myCount;  // # nodos del trie
      int countRepeated(Node* t) const;
      void printAllWords(const Node* t, const string& s) const;
      void dfs(Node* t, bool flag);
};
//-----------------------------------------------------------------------
// Constructor del Trie
Trie::Trie()
{
   myRoot = new Node();
   myCount = 1;
}
//-----------------------------------------------------------------------
// Retorna la cantidad de nodos del trie
int Trie::nodeCount() const
{
   return myCount;
}
//-----------------------------------------------------------------------
// Retorna true si el string s aparece en el trie
bool Trie::contains(const string& s) const
{
   Node* t = myRoot;
   int len = (int)s.size();
   for(int k = 0; k < len; ++k)
   {
      if(t == NULL) return false;
      t = t->links[s[k] - 'a'];
   }
   if(t == NULL) return false;
   return (t->words > 0);
}

//-----------------------------------------------------------------------
// Retorna la cantidad de veces que se repite el string s en el trie
int Trie::countWords(const string& s) const
{
   int len = (int)s.size();
   Node* t = myRoot;
   for(int k = 0; k < len; ++k)
   {
      if(t->links[s[k] - 'a'] == NULL) return 0;
      t = t->links[s[k] - 'a'];
```

```
   }
   return t->words;
}

//--------------------------------------------------------------------------
// Retorna la cantidad de palabras que tienen como prefijo a s
int Trie::countPrefixes(const string& s) const
{
   int len = (int)s.size();
   Node* t = myRoot;
   for(int k = 0; k < len; ++k)
   {
      if(t->links[s[k] - 'a'] == NULL) return 0;
      t = t->links[s[k] - 'a'];
   }
   return t->prefixes;
}

//--------------------------------------------------------------------------
// Imprime todas las palabras del trie en orden alfabetico
void Trie::printAllWords(const Node* t, const string& s) const
{
   if(t->words > 0) printf("%s\n",s.c_str());
   for(int k = 0; k < ALPH_SIZE; ++k)
      if(t->links[k]) printAllWords(t->links[k], s + char(k + 'a'));
}

void Trie::printAllWords() const
{
   printAllWords(myRoot, "");
}
//--------------------------------------------------------------------------
// Retorna la cantidad de palabras que aparecen mas de una vez en el trie
int Trie::countRepeated(Node* t) const
{
   int aux = 0;
   if(t->words > 1) ++aux;
   for(int k = 0; k < ALPH_SIZE; ++k)
      if(t->links[k]) aux += countRepeated(t->links[k]);
   return aux;
}

int Trie::countRepeated() const
{
   return countRepeated(myRoot);
}
//--------------------------------------------------------------------------
// Inserta un string s en el trie
void Trie::insert(const string s)
{
   int len = (int)s.size();
   Node* t = myRoot;
   for(int k = 0; k < len; ++k)
   {
      if(t->links[s[k] - 'a'] == NULL)
      {
```

```
            t->links[s[k] - 'a'] = new Node();
            ++(t->hijos); ++myCount;
         }
         ++(t->reachableWords);
         t = t->links[s[k] - 'a'];
         ++(t->prefixes);
      }
      ++(t->words);
}
//--------------------------------------------------------------------------
// Ejemplo de como hacer DFS en el Trie. Flag es true solo si t es la raiz
void Trie::dfs(Node* t, bool flag)
{
   // 1) Aca podemos llevar la cuenta de la respuesta. Ej:
   // if(t->words > 0) ans += t->cnt;
   //
   // 2) Aca podemos aumentar la cuenta de algun nodo. Ej:
   // for(int i = 0; i < ALPH_SIZE; i++)
   //   if(t->links[i])
   //      t->links[i]->cnt = t->cnt + 1;
   //
   // 3) Llamada recursiva
   for(int i = 0; i < ALPH_SIZE; i++)
      if(t->links[i])
         dfs(t->links[i], false);
}

void Trie::dfs()
{
   dfs(myRoot, true);
}
```

En el main

```
string test[] = {"tree","trie","algo","assoc","all","also"};
Trie* myTrie;
myTrie = new Trie();
for(int i = 0; i < 6; ++i) myTrie->insert(test[i]);
myTrie->printAllWords(); cout << endl;
delete myTrie;
```

SUFFIX ARRAY

i	sa[i]	lcp[i]	Suffix
0	8	-	a
1	1	1	argarita
2	4	2	arita
3	3	0	garita
4	6	0	ita
5	0	0	margarita
6	2	0	rgarita
7	5	1	rita
8	7	0	ta

BUILD - O(N LOG^2 N)

```
const int MAX = 100005, MAXLG = 17;
string s;   // s: String to process
int n;      // n: Size of string s
int P[MAX][MAXLG], sa[MAX]; // sa: Suffix Array
iii L[MAX];

void buildSA()
{
  if(n == 1) { sa[0] = 0; return; }
  for(int i = 0; i < n; i++)
    P[i][0] = s[i];
  for(int cont = 1, step = 1; cont < n; step++, cont <<= 1)
  {
    for(int i = 0; i < n; i++)
      L[i] = iii(ii(P[i][step-1], i + cont < n? P[i+cont][step-1] : -1), i);
    sort(L, L + n);
    for(int i = 0; i < n; i++)
      P[L[i].second][step] = i && L[i].first == L[i-1].first?
                             P[L[i-1].second][step] : i;
  }
  for(int i = 0; i < n; i++)
    sa[i] = L[i].second;
}
```

Example:

```
s = "mississippi";
n = (int)s.size();
buildSA();
for(int i = 0; i < (int)s.size(); i++)
  cout << " " << sa[i];  // 10 7 4 1 0 9 8 6 3 5 2
```

LONGEST COMMON PREFIX - O(LG N)

```
int getLCP(int x, int y) // x,y: Indexes of the string s
{
  if(x == y)
    return n - x;
  int ans = 0;
  for(int k = MAXLG - 1; k >= 0 && x < n && y < n; k--)
    if((1 << k) <= n && P[x][k] == P[y][k])
      x += 1 << k, y += 1 << k, ans += 1 << k;
  return ans;
}
```

LCP ARRAY - O(N LG N)

```
int lcp[MAX]; // lcp[i]: LCP bewteen sorted suffix i and i-1

void buildLCP()
{
```

```
  lcp[0] = 0; // lcp[0] is irrelevant
  for(int i = 1; i < n; i++)
    lcp[i] = getLCP(sa[i], sa[i-1]);
}
```

Example:

```
s = "mississippi";
n = (int)s.size();
buildSA();
for(int i = 0; i < (int)s.size(); i++)
  cout << " " << sa[i];  // 10 7 4 1 0 9 8 6 3 5 2
buildLCP();
for(int i = 0; i < (int)s.size(); i++)
  cout << " " << lcp[i]; // 0 1 1 4 0 0 1 0 2 1 3
```

SUFFIX ARRAY – APLICACIONES (OLD)

APLICACIÓN 1: STRING MATCHING - O(M LOG N)

El algoritmo buscar una cadena **P** de longitud **M** (sin '.' al final) en la cadena **T** de longitud **N** (con '.' al final).

Previamente se debe ejecutar la función constructSA para construir el Suffix Array de T.

```
//return lower/upper bound as the first/second item of the pair, respectively

typedef pair<int, int> ii;

char P[MAX_N];      // Cadena a buscar en T
int m;              // Longitud de P

ii stringMatching()
{
   int lo = 0, hi = n - 1, mid = lo;
   while(lo < hi)
   {
      mid = (lo + hi) / 2;
      int res = strncmp(T + SA[mid], P, m);
      if(res >= 0) hi = mid;
      else lo = mid + 1;
   }
   if(strncmp(T + SA[lo], P, m) != 0) return ii(-1, -1);

   ii ans; ans.first = lo;
   lo = 0; hi = n - 1; mid = lo;
   while(lo < hi)
   {
      mid = (lo + hi) / 2;
      int res = strncmp(T + SA[mid], P, m);
      if(res > 0) hi = mid;
      else lo = mid + 1;
   }
   if(strncmp(T + SA[hi], P, m) != 0) hi--;
   ans.second = hi;
```

```
   return ans;
}
```

En el main:

```
printf("Enter a string T:\n"); n = (int)strlen(gets(T));
T[n++] = '.'; T[n] = '\0';  // IMPORTANTE!!!!!
constructSA();

printf("Enter a string P:\n"); m = (int)strlen(gets(P));
ii pos = stringMatching();
if(pos.first != -1 && pos.second != -1)
{
   printf("%s is found SA [%d .. %d] of %s\n", P, pos.first, pos.second, T);
   printf("They are:\n");
   for(int i = pos.first; i <= pos.second; i++) printf("  %s\n", T + SA[i]);
}
else printf("%s is not found in %s\n", P, T);
```

APLICACIÓN 2: LONGEST REPEATED SUBSTRING - O(N)

Sea: T = 'margarita', el LRS de T es 'ar', ya que se repite 2 veces.

```
// Se asume que ya se tiene la cadena T (con '.' al final), su longitud n
// y que ya se llamó a los método constructSA() y computeLCP()
void LRS()
{
   char ans[MAX_N]; strcpy(ans, "");
   int maxLCP = 0;
   for(int i = 1; i < n; i++)
      if(LCP[i] > maxLCP)
      {
         maxLCP = LCP[i];
         strncpy(ans, T + SA[i], maxLCP);
         ans[maxLCP] = 0;
      }
      printf("The LRS is %s with length = %d\n", ans, maxLCP);
}
```

APLICACIÓN 3: LONGEST COMMON SUBSTRING

Sea: T = “UPC” y P = “ICPC”. El LCS de ambos es “PC”.

```
int owner(int idx) { return (idx < n - m - 1)? 1 : 2; }

void LCS()
{
   char ans[MAX_N]; strcpy(ans, "");
   int maxLCP = -1 ;
   for(int i = 1, maxLCP = -1; i < n; i++)
      if(LCP[i] > maxLCP && owner(SA[i]) != owner(SA[i-1]))
      {
         maxLCP = LCP[i];
```

```
        strncpy(ans, T + SA[i], maxLCP);
        ans[maxLCP] = 0;
      }
   printf("The LCS is %s with length = %d\n", ans, maxLCP);
}
```

En el main:

```
printf("Enter the string T:\n"); n = (int)strlen(gets(T));
printf("Now, enter another string P:\n"); m = (int)strlen(gets(P));
strcat(T, ".");       // add '.' at the back
strcat(T, P);         // append P
n = (int)strlen(T);   // update n
constructSA(); computeLCP();
LCS();
```

APLICACIÓN 4: MENOR ROTACIÓN LEXICOGRÁFICA

Sea: T = "casita", duplicando la cadena se tiene que T = "casitacasita". N = strlen(T) = 12.

SA[i]	Suffix
11	a
5	acasita
7	asita
1	asitacasita
6	casita
0	casitacasita
9	ita
3	itacasita
8	sita
2	sitacasita
10	ta
4	tacasita

La solución es el primer sufijo cuyo índice este por debajo de N / 2. En este caso sería “acasita”.

```
void MenorRotacionLex()
{
   int tam = n, ans; n = 2 * tam;
   char R[MAX_N]; strcpy(R, T);
   for(int i = tam; i < 2*tam; i++) T[i] = T[i - tam];
   constructSA();
   for(int i = 0; i < n; i++)
      if(SA[i] < tam)
      {
         ans = SA[i];  break;
      }
   printf("Menor rotacion lexicografica de %s: %s\n", R, R + ans);
   printf("La rotación debe darse en el indice: %d\n", ans + 1);
}
```

En el main:

```
printf("Enter the string T:\n");
n = (int)strlen(gets(T));
MenorRotacionLex();
```

Nota: El algoritmo da el mayor índice. Si existe más de una rotación que forme la misma palabra y necesitemos el primer índice, podemos usar KMP para comprimir la cadena de entrada antes de aplicar el algoritmo (Problema Uva 719 - Glass Beads).

```
/// COMPRESION DE CADENA antes de llamar a la función
string S = string(T);
vector<int> kmp;
kmp = KMP(S+S,S);
len = n / (kmp.size() - 1);
strncpy(T, S.c_str(), len); T[len] = '\0';
n = (int)strlen(T);
MenorRotacionLex();
```

GEOMETRY

PI AND EPSILON

```
#define PI (2*acos(0.0))
#define EPS 1e-07
```

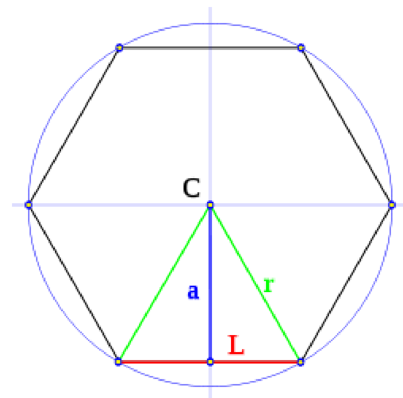
REGULAR POLYGONS

AREA

Siendo: **L** la longitud de un lado, **n** el número de lados, **P** el perímetro y **r** el radio de la circunferencia circunscrita, tenemos:

$$A_p = \frac{P \cdot a}{2} \qquad A_p = \frac{nr^2 \sin(\frac{2\pi}{n})}{2}$$

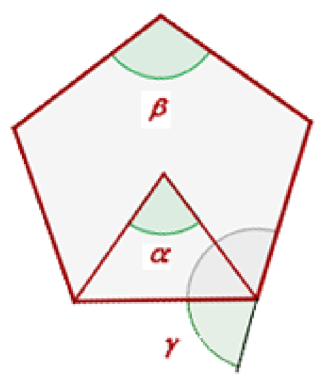
$$A_p = n \cdot \frac{L^2}{4} \cdot \tan\left(\frac{\pi}{2} \frac{(n-2)}{n}\right)$$



ANGLES

Siendo: **n** el número de lados del polígono, tenemos:

$$\alpha = \frac{360}{n} \qquad \beta = 180 - \frac{360}{n} = 180 - \alpha \qquad \gamma = \alpha = \frac{360}{n}$$



DIAGONALS

El número de diagonales en un polígono regular es: $D_n = \frac{n(n-3)}{2}$

STRUCTURES

POINT

```
struct point { double x, y; };
```

LINE

```
struct line { double a, b, c; }; // ax + by + c = 0
```

CIRCLE

```
struct circle { point c; double r; };
```

VECTOR

```
typedef vec point; // vector y punto usan la misma estructura
```

POINTS

DISTANCE BETWEEN TWO POINTS

```
double dist(point p, point q) { return hypot(p.x - q.x, p.y - q.y); }
```

ROTAR UN PUNTO X GRADOS EN SENTIDO ANTI HORARIO

```
void rotate(point p, double theta, point *ans) // Rota alrededor del origen
{
   double rad = theta * PI / 180.0; // A Radianes   // Matriz de rotacion:
   ans->x = p.x * cos(rad) - p.y * sin(rad);        // [cos(rad) -sin(rad)]
   ans->y = p.x * sin(rad) + p.y * cos(rad);        // [sin(rad)  cos(rad)]
}
```

REFLEJAR UN PUNTO

```
void reflejar(point p, int m, point *ans) // m es la pendiente: y = mx + b
{
   ans->x = (p.x - m * m * p.x) / (m * m + 1) + (2 * m * p.y) / (m * m + 1);
   ans->y = (2 * m * p.x) / (m * m + 1) + (m * m * p.y - p.y) / (m * m + 1);
}
```

VERIFICAR SI DOS PUNTOS SON IGUALES

```
bool areSame(point p, point q)
{
   return fabs(p.x - q.x) < EPS && fabs(p.y - q.y) < EPS;
}
```

CHESSBOARD DISTANCE ("CHEBYSHEV DISTANCE")

Distancia que le tomaría al rey (ajedrez) llegar de una casillero p a un casillero q

```
int distancia(point p, point q) {return max(abs(p.x - q.x),abs(p.y - q.y)); }
```

LINES

CONVERTIR DOS PUNTOS EN LÍNEA

```
void points_to_line(point p1, point p2, line *L)
{
   if(p1.x == p2.x) { L->a = 1; L->b = 0; L->c = -p1.x; } // Formula: x = k
   else
   {                                          // Formula Base: y = mx + k
      L->a = -(p1.y - p2.y) / (p1.x - p2.x); // -mx + y - k = 0; => a = -m
      L->b = 1;                              // -mx + y - k = 0; => b = 1
      L->c = -(L->a * p1.x) - (L->b * p1.y); // c = -ax - by
   }
}
```

CONVERTIR PUNTO Y PENDIENTE EN LÍNEA

```
void point_and_slope_to_line(point p, double m, line *L)
{
   L->a = -m; L->b = 1; L->c = -(L->a * p.x) - (L->b * p.y); //-mx + y -k=0
}
```

ÁNGULO DE INTERSECCIÓN

```
double angulo_interseccion(line L1, line L2)
{
   return atan2(L1.a * L2.b - L2.a * L1.b, L1.a * L2.a + L1.b * L2.b);
}
```

DETERMINAR PUNTO DE INTERSECCIÓN

```
bool parallelQ(line L1, line L2)
{
   return (fabs(L1.a - L2.a) <= EPS) && (fabs(L1.b - L2.b) <= EPS);
}

bool same_lineQ(line L1,line L2)
{
   return parallelQ(L1,L2) && (fabs(L1.c-L2.c) <= EPS);
}

bool intersection_point(line L1, line L2, point *p) // Resolver Sist. Ecu. Lin.
{
   if(same_lineQ(L1,L2)) return false; // Misma linea
   if(parallelQ(L1,L2))  return false; // Lineas paralelas
   p->x = (L2.b * L1.c - L1.b * L2.c) / (L2.a * L1.b - L1.a * L2.b);
   if(fabs(L1.b) > EPS) p->y = - (L1.a * p->x + L1.c) / L1.b;
```

```
   else                  p->y = - (L2.a * p->x + L2.c) / L2.b;
   return true;
}
```

COMPARACION DE DOUBLES CON PRECISION

```
int comparar(double d1, double d2)
{
   if(d2 - d1 > EPS) return -1; // d2 > d1
   if(d1 - d2 > EPS) return 1;  // d1 > d2
   return 0;                    // d1 == d2
}
```

DISTANCIA DE UN PUNTO A UNA LÍNEA

```
double distToLine (point p, point A, point B, point *c)
{
   // Disancia de p a la línea AB. El punto mas cercano se guarda en c
   double scale = (double)
       ((p.x - A.x) * (B.x - A.x) + (p.y - A.y) * (B.y - A.y)) /
       ((B.x - A.x) * (B.x - A.x) + (B.y - A.y) * (B.y - A.y));
   c->x = A.x + scale * (B.x - A.x);
   c->y = A.y + scale * (B.y - A.y);
   return distancia(p,*c);
}
```

DISTANCIA DE UN PUNTO A UN SEGMENTO

```
double distToLineSegment(point p, point A, point B, point *c)
{
   // Distancia de p al segmento AB. El punto mas cercano se guarda en c
   if((B.x - A.x) * (p.x - A.x) + (B.y - A.y) * (p.y - A.y) < EPS)
   {
      c->x = A.x; c->y = A.y;
      return distancia(p, A);
   }
   if((A.x - B.x) * (p.x - B.x) + (A.y - B.y) * (p.y - B.y) < EPS)
   {
      c->x = B.x; c->y = B.y;
      return distancia(p, B);
   }
   return distToLine(p, A, B, c);
}
```

INTERSECCIÓN DE SEGMENTOS

Modificación de algunas funciones anteriores:

```
bool isSameLine; // Se agrega esta variable global

bool intersection_point(line L1, line L2, point *p)
```

```
{
   // La siguiente parte del código es la que cambia:
   if(same_lineQ(L1,L2)) { isSameLine = true; return true; }
   // Resto del codigo se mantiene igual
}
```

Implementación:

```
bool SegmentIntersection(point pAx, point pAy, point pBx, point pBy)
{
   line L1,L2;
   point pIntersection,pAux; // pInterseccion: Punto de cruce de lineas
   points_to_line(pAx, pAy, &L1); points_to_line(pBx, pBy, &L2);
   isSameLine = false;
   bool res = intersection_point(L1, L2, &pIntersection);
   if(isSameLine)
   {
      if(comparar(dist(pAx, pAy), dist(pBx, pBy)) > 0)
      {
         if(comparar(distToLineSegment(pBx, pAx, pAy, &pAux), 0.0) == 0 ||
            comparar(distToLineSegment(pBy, pAx, pAy, &pAux), 0.0) == 0)
            return true;
      }
      else
      {
         if(comparar(distToLineSegment(pAx, pBx, pBy, &pAux), 0.0) == 0 ||
            comparar(distToLineSegment(pAy, pBx, pBy, &pAux), 0.0) == 0)
            return true;
      }
      return false;
   }
   else if(res && comparar(distToLineSegment(pIntersection,pAx,pAy,&pAux),0.0) == 0 &&
                  comparar(distToLineSegment(pIntersection,pBx,pBy,&pAux),0.0) == 0)
      return true;
   return false;
}
```

VECTORS

CONVERTIR DOS PUNTOS A VECTOR (P1 -> P2, P1: BASE, P2: CABEZA)

```
vec toVector(point p1, point p2) { return vec(p2.x - p1.x, p2.y - p1.y); }
```

ESCALAR VECTOR

```
vec scaleVector(vec v, double s) { return vec(v.x * s, v.y * s); }
```

TRASLADAR UN PUNTO DE ACUERDO A UN VECTOR

El punto se mueve una distancia igual a la magnitud de v, siguiendo su direccion

```
point translate(point p, vec v) { return point(p.x + v.x, p.y + v.y); }
```

PRODUCTO PUNTO

```
double dot(point p, point q) { return p.x * q.x + p.y * q.y; }
```

PRODUCTO CRUZ

```
double cross(point p, point q) { return p.x * q.y - q.x * p.y; }
```

MODULO^2 DE UN VECTOR

```
double norm(point p) { return p.x * p.x + p.y * p.y; }
```

ÁNGULO ENTRE VECTORES

```
double angleVectors(vec v1, vec v2)
{
   return acos(dot(v1, 2) / sqrt(norm(v1) * norm(v2))); // 0 <= angulo <= pi
}
```

CIRCLES

CÍRCULO EN BASE A 3 PUNTOS

Devuelve el centro del circulo en base a 3 puntos.

```
bool center_from_3points(point p1, point p2, point p3, point *c)
{
   point a1, a2; // a1: midpoint of line p2p3, a2: midpoint of line p1p3
   a1.x = (p2.x + p3.x) * 0.5;  a1.y = (p2.y + p3.y) * 0.5;
   a2.x = (p1.x + p3.x) * 0.5;  a2.y = (p1.y + p3.y) * 0.5;
   point b1, b2; // b1: point on the line a1c (use congruent triangles)
   b1.x = a1.x - (p3.y - p2.y);  b1.y = a1.y + (p3.x - p2.x);
   b2.x = a2.x - (p3.y - p1.y);  b2.y = a2.y + (p3.x - p1.x);
   line L1, L2; // perpendicular lines to p2p3 and p1p3 respectively
   point_to_line(a1, b1, &L1);
   point_to_line(a2, b2, &L2);
   return intersection_point(L1, L2, c);
}
```

CENTRO DEL CÍRCULO EN BASE A 2 PUNTOS Y AL RADIO

El centro devuelto esta a la izquierda del vector p1 -> p2. Para devolver el otro punto, invertir p1 y p2 al momento de llamar a la función.

```
bool circle2PtsRad(point p1, point p2, double r, point *c)
{
   double d2 = (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
   double det = r * r / d2 - 0.25;
   if(det < 0.0) return false;
```

```
   double h = sqrt(det);
   c->x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
   c->y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
   return true;
}
```

PUNTOS DE TANGENCIA

```
int tangent_points(point p, circle c, point *t1, point *t2)
{
   // Solo 1 punto de tangencia
   if(comparar(distancia(p, c.c), c.r) == 0)
   {
      t1->x = p.x; t1->y = p.y;
      return 1;
   }
   // 2 puntos de tangencia
   point pr1, pr2;
   double h = distancia(p, c.c), ang = asin(c.r / h);
   // Rotar centro alrededor del punto p
   point protated(c.c.x - p.x, c.c.y - p.y);
   rotate(protated, -ang, &pr1); rotate(protated,  ang, &pr2);
   pr1.x += p.x; pr1.y += p.y; pr2.x += p.x; pr2.y += p.y;
   // Ajustar puntos rotados
   vec v1 = scaleVector(toVector(p, pr1), sqrt(h*h - c.r*c.r) / distancia(p,pr1));
   vec v2 = scaleVector(toVector(p, pr2), sqrt(h*h - c.r*c.r) / distancia(p,pr1));
   t1 = translate(p, v1), t2 = translate(p, v2);
   return 2;
}
```

MENOR ÁNGULO FORMADO POR DOS PUNTOS DE LA CIRCUNFERENCIA

Si se requiere el arco, multiplicar el resultado por el radio del círculo .

```
double angulo(point p1, point p2, circle c)
{
   if(comparar(distancia(p1, p2), 2 * c.r) == 0) return PI;
   double x = distancia(p1, p2);
   return acos(1 - (x * x) / (2 * c.r * c.r));
}
```

INTERSECCIÓN DE CÍRCULOS

```
bool IntersectaCircunferencias(circle c1, circle c2)
{
   double d = distancia(c1.c, c2.c); // Distancia entre centros
   if(comparar(d, 0) == 0) return comparar(c1.r, c2.r) == 0; // Concentricos
   if(comparar(d, c1.r + c2.r) == 1 || comparar(d, fabs(c1.r - c2.r)) == -1)
      return false;
   return true;
}
```

ÁREA DE CÍRCULO INSCRITO Y CIRCUNSCRITO A UN POLÍGONO

A: Área del Polígono. **N**: Número de lados del polígono.

$$Area_{CirculoInscrito} = \pi \frac{A}{N \tan(\frac{\pi}{N})} \qquad Area_{CirculoCircunscrito} = \pi \frac{2A}{N \sin(\frac{2\pi}{N})}$$

```
double incircleArea = A * PI / (N * tan(PI / N));
double excircleArea = 2 * A * PI / (N * sin(2 * PI / N));
```

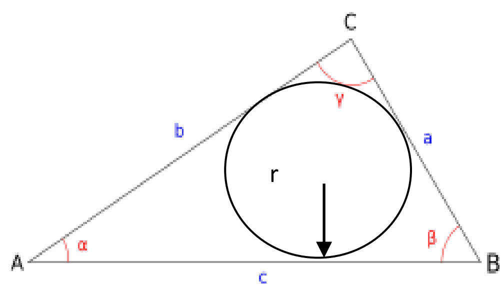
TRIANGLES

FORMULARIO DE ÁREAS DE UN TRIÁNGULO

En base a sus 3 lados

Sea un triángulo ABC y sus lados ***a***, ***b*** y ***c***, el área **A** es:

$$A = \sqrt{s\,(s-a)\,(s-b)\,(s-c)},$$



Donde **s** es el semiperímetro:

$$s = \frac{a+b+c}{2}$$

El radio **r** de la circunferencia inscrita en un triángulo es:

$$r = \sqrt{\frac{(s-a)(s-b)(s-c)}{s}}$$

En base a sus 3 vértices

$$area = \left| \frac{Ax(By-Cy)+Bx(Cy-Ay)+Cx(Ay-By)}{2} \right|$$

En función al circunradio:

$$A = \frac{1}{4R}abc$$

En función al inradio:

$$A = sr$$

En función al exradio:

$$A = r_a(s-a) = r_b(s-b) = r_c(s-c)$$

r_a, r_b, r_c: exradios relativos a los lados a, b y c

Triángulo Equilátero:

$$A = \frac{1}{4}l^2\sqrt{3} = \frac{1}{3}h^2\sqrt{3}$$

En función a dos lados y al ángulo entre ellos:

$$A = \frac{1}{2}bc \cdot sen(A) = \frac{1}{2}ac \cdot sen(B) = \frac{1}{2}ab \cdot sen(C)$$

En función del circunradio y de los senos de los ángulos

$$A = 2 \cdot R^2 \cdot sin(A) \cdot sin(B) \cdot sin(C)$$

En función del inradio y del exradio relativo a la hipotenusa

$A = r_a \cdot r$ **r_a**: exradio relativo a la hipotenusa.

En función de los exradios relativos a los catetos

$A = r_b \cdot r_c$ **r_b**, **r_c**: exradios relativos a los catetos.

En función de m y n (triángulo rectángulo únicamente)

$A = m \cdot n$ **m**, **n**: segmentos de la base partidos por la circunferencia inscrita.

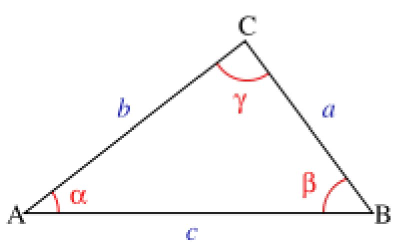
ÁREA EN BASE A LAS 3 MEDIANAS

```
double areaFromMedians(double ma, double mb, double mc)
{
   double x = 0.5 * (ma + mb + mc);
   double a = x * (x - ma) * (x - mb) * (x - mc);
   if(a < 0.0) return -1.0;          // No existe triangulo
   else return sqrt(a) * 4.0 / 3.0;
}
```

ÁREA EN BASE A 3 PUNTOS

```
double area(int x1, int y1, int x2, int y2, int x3, int y3)
{
   return fabs( (x1*(y2-y3) + x2*(y3-y1) + x3*(y1-y2)) / 2.0 );
}
```

LEYES TRIGONOMÉTRICAS



Condición de existencia:

$$a \prec b + c$$
$$b \prec a + c$$
$$c \prec a + b$$

Ley de senos:

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$$

Ley de cosenos:

$$\begin{aligned} a^2 &= b^2 + c^2 - 2bc\cos\alpha \\ b^2 &= a^2 + c^2 - 2ac\cos\beta \\ c^2 &= a^2 + b^2 - 2ab\cos\gamma \end{aligned},$$

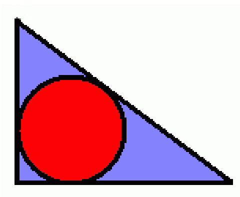
Ley de tangentes:

$$\frac{a-b}{a+b} = \frac{\tan[\frac{1}{2}(\alpha-\beta)]}{\tan[\frac{1}{2}(\alpha+\beta)]}.$$

Formula de Mollweide:

$$\frac{a-b}{c} = \frac{\sin\left(\frac{\alpha-\beta}{2}\right)}{\cos\left(\frac{\gamma}{2}\right)}.$$

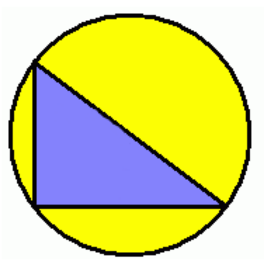
RADIO DEL CÍRCULO INSCRITO



$$r = \frac{A}{s}$$

r: Radio del círculo inscrito
A: Área del triángulo
s: Semiperímetro del triángulo

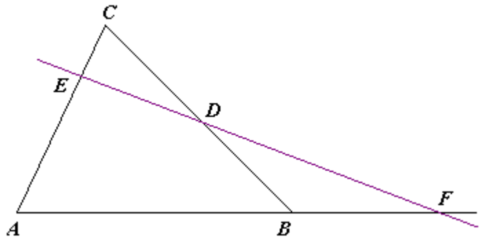
RADIO DEL CÍRCULO CIRCUNSCRITO



$$R = \frac{a \times b \times c}{4 \times A}$$

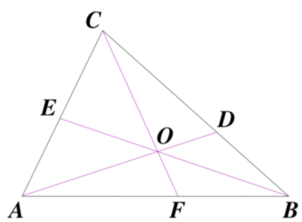
R: Radio del círculo circunscrito
a, **b**, **c**: Lados del triángulo
A: Área del triángulo

TEOREMA DE MENELAO



$$\frac{AE}{EC} \cdot \frac{CD}{DB} \cdot \frac{BF}{FA} = 1$$

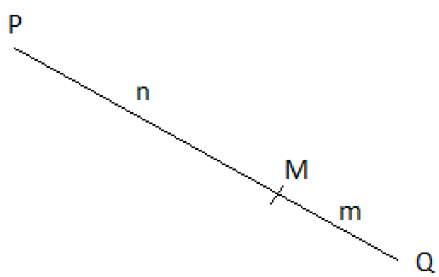
TEOREMA DE CEVA



$$\frac{AF}{FB} \cdot \frac{BD}{DC} \cdot \frac{CE}{EA} = 1,$$

$$\frac{\sin\angle BAD}{\sin\angle CAD} \cdot \frac{\sin\angle CBE}{\sin\angle ABE} \cdot \frac{\sin\angle ACF}{\sin\angle BCF} = 1.$$

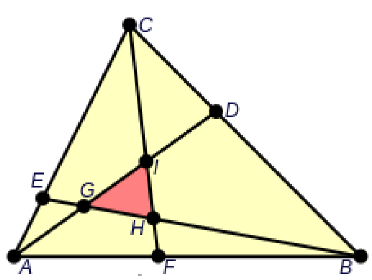
TEOREMA DE THALES



$$M_x = \frac{m * P_x + n * Q_x}{n + m}$$

$$M_y = \frac{m * P_y + n * Q_y}{n + m}$$

TEOREMA DE ROUTH



$$\overline{AF}/\overline{BF} = r$$
$$\overline{BD}/\overline{CD} = s$$
$$\overline{CE}/\overline{AE} = t$$

$$A_{IGH} = \frac{(r \cdot s \cdot t - 1)^2}{(s \cdot t + s + 1)(r \cdot t + t + 1)(r \cdot s + r + 1)} A_{ABC}.$$

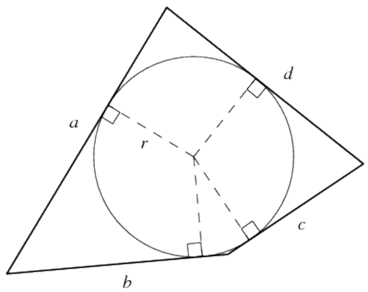
QUADRILATERALS

CONDICIÓN DE EXISTENCIA EN BASE A 4 LADOS

```
vector<int> lados; // Contiene los 4 lados del posible cuadrilátero
sort(lados.begin(), lados.end());
if(lados[0] + lados[1] + lados[2] > lados[3]) printf("Existe");
```

CUADRILATERO TANGENCIAL

Es un cuadrilátero convexo cuyos lados son tangentes a una circunferencia inscrita.

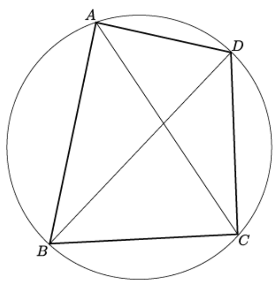


$$a + c = b + d = \frac{a+b+c+d}{2} = s$$

$$A = r \cdot (a+c) = r \cdot (b+d) = r \cdot s$$

CUADRILATERO CICLICO

Cuadrilátero que tiene sus cuatro vértices en una circunferencia. Cuando se quiere construir un cuadrilátero de **área máxima con 4 lados de longitud fija**, este **debe ser cíclico** y podemos usar estas fórmulas para hallar el área que debe tendrá el cuadrilátero.



$$A = \sqrt{(s-a)(s-b)(s-c)(s-d)}$$

$$s = \frac{a+b+c+d}{2}$$

RECTANGLES

ESTRUCTURA DE RECTÁNGULOS

```
struct rect { int xmin, xmax, ymin, ymax; }; // puede ser double
```

ÁREA DE RECTÁNGULO

```
double area(rect r) { return (r.xmax - r.xmin) * (r.ymax - r.ymin); }
```

CONDICIÓN DE INTERSECCIÓN DE RECTÁNGULOS

```
bool intersectan(rect r1, rect r2)
{
   if(r1.xmin < r2.xmax && r1.xmax > r2.xmin &&
      r1.ymin < r2.ymax && r1.ymax > r2.ymin)
      return true;
   return false;
}
```

INTERSECCIÓN DE DOS RECTÁNGULOS

```
if(intersectan(r1, r2))
```

```
{
   r1.xmin = max(r1.xmin, r2.xmin); r1.xmax = min(r1.xmax, r2.xmax);
   r1.ymin = max(r1.ymin, r2.ymin); r1.ymax = min(r1.ymax, r2.ymax);
   printf("%d %d %d %d\n", r1.xmin, r1.ymin, r1.xmax, r1.ymax);
}
else printf("No Overlap\n");
```

ÁREA DE UNIÓN DE RECTÁNGULOS - O(N^2)

```
struct edge // Cada lado vertical del rectangulo representa un evento
{
   double x, ymin, ymax;  // Lado vertical: 1 punto en x, 2 puntos en y
   int m;                 // Tipo de evento: inicio(1)/fin(-1)
   bool operator<(const edge &e) const
   {
      return x < e.x;
   }
};

double areaUnionRect(vector<rect> R)
{
   int n = (int)R.size();
   vector<double> ys(2 * n);  // Todas las coordenadas y
   vector<edge> e(2 * n);     // Vector de eventos
   for(int i = 0; i < n; ++i) // Cada rectangulo define 2 eventos
   {
      e[2 * i].ymin = e[2 * i + 1].ymin = ys[2 * i] = R[i].ymin;
      e[2 * i].ymax = e[2 * i + 1].ymax = ys[2 * i + 1] = R[i].ymax;
      e[2 * i].x = R[i].xmin; e[2 * i + 1].x = R[i].xmax;
      e[2 * i].m = 1;         e[2 * i + 1].m = -1; // inicio y fin
   }
   sort(ys.begin(), ys.end()); sort(e.begin(), e.end());
   double ans = 0, cur = 0;        // cur: ancho de los rectangulos activos
   for(int i = 0; i < 2 * n; ++i)  // Desplazamos una linea imaginaria en Y
   {
      if(i) ans += (ys[i] - ys[i - 1]) * cur; // area = base * altura
      int fag = 0;           // Rectangulos activos
      double sx = cur = 0;
      for(int j = 0; j < 2 * n; ++j) // Recorrer todos los eventos
         if(e[j].ymin <= ys[i] && ys[i] < e[j].ymax) // La linea cruza el evento
         {
            if(!fag) sx = e[j].x; // Primer rectangulo activo
            fag += e[j].m;        // Actualizamos los rectangulos activos
            if(!fag) cur += e[j].x - sx; // Ya no hay rectangulos activos
         }
   }
   return ans;
}
```

ÁREA DE UNIÓN DE RECTÁNGULOS - O(N LG N)

```
#define MAX 30005 // Máxima coordenada de los rectángulos
using namespace std;
```

```
class lazyProp
{
   private:
      vector<int> T; // T: Almacena cuantos números positivos en el rango
      vector<int> F; // F: Almacena+ cuantos +1 están cubriendo el rango
      int N;
      void pull(int node, int L, int R);
      void increment(int node, int L, int R, int x, int y, int val);
    public:
      lazyProp();
      int query();
      void increment(int x, int y, int val);
};

lazyProp::lazyProp() { N = MAX; T.assign(N * 4, 0); F.assign(N * 4, 0); }

void lazyProp::pull(int node, int L, int R)
{
   if(F[node]) T[node] = R - L + 1;
   else
   {
      T[node] = 0;
      if(L != R) T[node] = T[2 * node + 1] + T[2 * node + 2];
   }
}

int lazyProp::query()
{
   return T[0];
}

void lazyProp::increment(int node, int L, int R, int x, int y, int val)
{
   if(R < x || L > y) return;
   if(x <= L && R <= y) { F[node] += val; pull(node, L, R); }
   else
   {
      int mid = (L + R) * 0.5;
      increment(2 * node + 1, L, mid, x, y, val);
      increment(2 * node + 2, mid + 1, R, x, y, val);
      pull(node, L, R);
   }
}

void lazyProp::increment(int x, int y, int val)
{
   return increment(0, 0, N - 1, x, y, val);
}

struct rect { int xmin, xmax, ymin, ymax; };

struct edge // Los eventos son los lados verticales del rectangulo
{
   int x, ymin, ymax; // Barrido horizontal: 1 punto en x, 2 puntos en y
   int m; // Tipo de evento: abierto (1), cerrado(-1)
   bool operator < (const edge &e) const
```

```
    {
       if(x != e.x) return x < e.x; // Ordenar ascendente en x
       if(m != e.m) return m < e.m; // Primero los eventos que cierran
       if(ymin != e.ymin) return ymin < e.ymin;
       return ymax < e.ymax;
    }
};

int areaUnionRect(vector<rect> R)
{
   int n = (int)R.size();
   vector<edge> e(2 * n); // Vector de eventos
   for(int i = 0; i < n; i++)
   {
      e[2 * i].ymin = e[2 * i + 1].ymin = R[i].ymin;
      e[2 * i].ymax = e[2 * i + 1].ymax = R[i].ymax;
      e[2 * i].x = R[i].xmin; e[2 * i + 1].x = R[i].xmax;
      e[2 * i].m = 1;         e[2 * i + 1].m = -1; // Inicio y fin
   }
   sort(e.begin(), e.end()); // Ordenar eventos
   int ans = 0, h = 0;
   lazyProp *st = new lazyProp();
   for(int i = 0; i < 2 * n; i++)
   {
      if(i) ans += (e[i].x - e[i - 1].x) * h; // Area = base x altura
      st->increment(e[i].ymin, e[i].ymax - 1, (e[i].m == 1)? +1 : -1);
      h = st->query();
   }
   delete st;
   return ans;
}
```

CUBES

ESTRUCTURA DE CUBOS (Ó PARALELEPIPEDOS)

```
struct cube { int xmin, xmax, ymin, ymax, zmin, zmax; };
```

VOLUMEN DE CUBOS

```
double volumen(cube r)
{
   return (r.xmax - r.xmin) * (r.ymax - r.ymin) * (r.zmax - r.zmin);
}
```

CONDICIÓN DE INTERSECCIÓN DE CUBOS

```
bool intersectan(cube r1, cube r2)
{
   if(r1.xmin < r2.xmax && r1.xmax > r2.xmin &&
      r1.ymin < r2.ymax && r1.ymax > r2.ymin &&
      r1.zmin < r2.zmax && r1.zmax > r2.zmin)
```

```
      return true;
   return false;
}
```

INTERSECCIÓN DE DOS CUBOS

```
if(intersectan(r1, r2))
{
   r1.xmin = max(r1.xmin, r2.xmin); r1.xmax = min(r1.xmax, r2.xmax);
   r1.ymin = max(r1.ymin, r2.ymin); r1.ymax = min(r1.ymax, r2.ymax);
   r1.zmax = min(r1.zmax, r2.zmax); r1.zmin = max(r1.zmin, r2.zmin);
}
else printf("No Overlap\n");
```

COMPUTATIONAL GEOMETRY

GREAT CIRCLE DISTANCE

```
double calc(double pLat, double pLon, double qLat, double qLon, double r)
{
  pLat *= PI / 180; pLon *= PI / 180;
  qLat *= PI / 180; qLon *= PI / 180;
  return r * acos(cos(pLat) * cos(pLon) * cos(qLat) * cos(qLon) +
                  cos(pLat) * sin(pLon) * cos(qLat) * sin(qLon) +
                  sin(pLat) * sin(qLat));
}
```

POLYGONS

AREA

```
// Cross Product
double cross(point p, point q) { return p.x * q.y - q.x * p.y; }

double area(const vector<point> &p)
{
  int n = (int)p.size();
  double ans = 0.0;
  for(int i = 0; i < n; i++)
    ans += cross(p[i], p[(i + 1) % n]); // Producto cruz entre vecinos
  return fabs(ans) / 2.0;
}
```

PERIMETER

```
double dist(point p, point q) { return hypot(p.x - q.x, p.y - q.y); }

double perimeter(const vector<point> &p)
{
  int n = (int)p.size();
  double ans = 0.0;
  for(int i = 0; i < n; i++)
    ans += dist(p[i], p[(i + 1) % n]); // Distancia entre vecinos
  return ans;
}
```

TURNS

```
int comparar(double d1, double d2)
{
  if(d2 - d1 > EPS) return -1; // d1 < d2
  if(d1 - d2 > EPS) return 1;  // d2 > d1
  return 0;                    // d1 == d2
}
```

```
double cross(point p, point q, point r) // cross product of vectors qr and qp
{
  return (r.x - q.x) * (p.y - q.y) - (r.y - q.y) * (p.x - q.x);
}

// -1: right turn, 1: left turn, 0: colineares
int turn(point p, point q, point r) { return comparar(cross(p,q,r), 0); }

// Giro Antihorario
bool ccw(point p, point q, point r) { return turn(p, q, r) > 0; }

// Giro Horario
bool cw(point p, point q, point r) { return turn(p, q, r) < 0; }
```

DETERMINE IF POLYGON IS CONVEX

```
bool isConvex(const vector<point> &p)
{
  int n = (int)p.size();
  if(n < 3) return false;
  bool orientation = ccw(p[0], p[1], p[2]);
  for(int i = 0; i < n; i++) // Los puntos deben seguir la misma orientacion
    if(ccw(p[i], p[(i+1) % n], p[(i+2) % n]) != orientation)
      return false;
  return true;
}
```

DETERMINAR SI UN PUNTO ESTA ESTRICTAMENTE DENTRO DE UN POLIGONO

```
// Producto punto
double dot(point p, point q) { return p.x * q.x + p.y * q.y; }
// Modulo^2 de un punto (distancia al origen)
double norm(point p) { return p.x * p.x + p.y * p.y; }

double angle(point a, point o, point b) // Angulo AOB en radianes
{
   point u(a.x - o.x, a.y - o.y); // u: vector oa
   point v(b.x - o.x, b.y - o.y); // v: vector ob
   return acos(dot(u, v) / sqrt(norm(u) * norm(v))); // Definicion prod punto
}

bool inPolygon(point p, vector<point> P)
{
   double sum = 0.0;
   int n = (int)P.size();
   for(int i = 0; i < n; i++)
      if(cross(p, P[i], P[(i + 1) % n]) < 0)
         sum -= angle(P[i], p, P[(i + 1) % n]);
      else
         sum += angle(P[i], p, P[(i + 1) % n]);
   return fabs(fabs(sum) - 2 * PI) < EPS; // Suma debe ser 360°
}
```

PUNTO DE CORTE ENTRE SEGMENTO Y LÍNEA

Punto de intersección entre el segmento pq y la línea AB

```
point lineIntersectSeg(point p, point q, point A, point B)
{
   double a = B.y - A.y;
   double b = A.x - B.x;
   double c = B.x * A.y - A.x * B.y;
   double u = fabs(a * p.x + b * p.y + c);
   double v = fabs(a * q.x + b * q.y + c);
   return point((p.x * v + q.x * u) / (u + v), (p.y * v + q.y * u) / (u + v));
}
```

CORTAR UN POLÍGONO

Devuelve el lado izquierdo del polígono cortado por la línea ab. Para obtener el otro lado invertir A y B.

```
vector<point> cutPolygon(point a, point b, vector<point> Q)
{
  vector<point> P;
  Q.push_back(Q.front());
  for(int i = 0; i < (int)Q.size(); i++)
  {
    double left1 = cross(a, b, Q[i]), left2 = 0.0;
    if(i != (int)Q.size() - 1) left2 = cross(a, b, Q[i + 1]);
    if(left1 > -EPS) P.push_back(Q[i]); // Q[i] esta a la izq de ab
    if(left1 * left2 < -EPS)            // lado(Q[i],Q[i+1]) cruza ab
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
  }
  Q.pop_back();
  if(P.empty()) return P;
  if(fabs(P.back().x - P.front().x) > EPS ||
     fabs(P.back().y - P.front().y) > EPS)
    return P;
  else
    P.pop_back(); // El ultimo punto se repite, lo eliminamos
  return P;
}
```

ORDENAR LOS PUNTOS DE UN POLIGONO EN SENTIDO ANTIHORARIO

```
point pivot;

bool angle_cmp(point a, point b)
{
   if(turn(pivot, a, b) == 0) // Los puntos son colineares con el pivote
      return distancia(pivot, a) < distancia(pivot, b);
   double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
   double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
   return atan2(d1x, d1y) - atan2(d2x, d2y) < 0;
}
```

```
void sort_clockwise(vector<point> &P)
{
   int lowY = 0, n = (int)P.size();
   for(int i = 1; i < n; i++)
      if(P[i].y < P[lowY].y || (P[i].y == P[lowY].y && P[i].x > P[lowY].x) )
         lowY = i;
   point temp = P[0]; P[0] = P[lowY]; P[lowY] = temp; pivot = P[0];
   sort(++P.begin(), P.end(), angle_cmp);
}
```

CENTRO DE MASAS DE UN POLIGONO

Los puntos del poligono deben estar en sentido horario.

```
double signed_area(vector<point> P)
{
   double ans = 0.0;
   int n = (int)P.size();
   for(int i = 0; i < n; i++) ans += cross(P[i], P[(i + 1) % n]);
   return ans / 2.0;
}

void center_mass(vector<point> P, point *c) // c: Centro de masas
{
   double x1,y1,x2,y2, A = signed_area(P);
   c->x = c->y = 0;
   int n = (int)P.size();
   for(int i = 0; i < n; i++)
   {
      x1 = P[i].x; x2 = P[(i + 1) % n].x;
      y1 = P[i].y; y2 = P[(i + 1) % n].y;
      c->x += (x1 + x2) * (x1 * y2 - x2 * y1);
      c->y += (y1 + y2) * (x1 * y2 - x2 * y1);
   }
   c->x /= 6 * A; c->y /= 6 * A;
}
```

CENTRO DE MASAS DE UN CONJUNTO DE PUNTOS

```
void center_mass_points(vector<point> P, point *c) // c: Centro de masas
{
   c->x = c->y = 0;
   int n = (int)P.size();
   for(int i = 0; i < n; i++)
   {
      c->x += P[i].x;
      c->y += P[i].y;
   }
   c->x /= n; c->y /= n; // El centro de masas es el promedio
}
```

CANTIDAD DE LATTICE POINTS EN UN SEGMENTO

```
int latticePoints(point a, point b)
{
   return gcd(abs(a.x - b.x), abs(a.y - b.y)) + 1;
}
```

TEOREMA DE PICK

- Para polígonos con coordenadas enteras

  I = puntos en el interior

  B = puntos en el borde

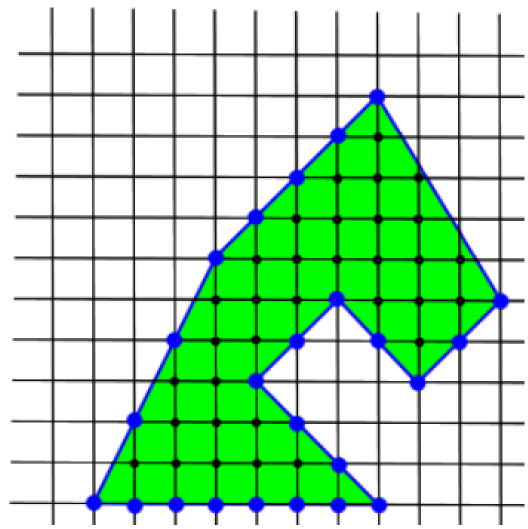
  S = superficie

- **Teorema de Pick**

  $S = I + B \div 2 - 1$

  $I = S - B \div 2 + 1$

  (Se puede probar por inducción en la superficie)



Podemos hallar los puntos en el borde **B** en O(n).

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

int border(vector<point> P)
{
   int n = (int)P.size(), ans = 0;
   for(int i = 0; i < n; i++)
      ans += gcd(abs(P[i].x - P[(i+1) % n].x), abs(P[i].y - P[(i+1) % n].y));
   return ans;
}
```

Para hallar **I**. Requiere la función que calcula el área del polígono

```
double puntosInterior(vector<point> P)
{
   return area(P) - (border(P) / 2.0) + 1.0;
}
```

CONVEX HULL

GRAHAM SCAN - O(N LOG N)

```
#define EPS 1e-7

struct point { double x, y; };

double distancia(point p, point q) { return hypot(p.x - q.x, p.y - q.y); }

int comparar(double d1, double d2)
{
   if(d2 - d1 > EPS) return -1;
   if(d1 - d2 > EPS) return 1;
   return 0;
}

int turn(point p, point q, point r)
{
   double result = (r.x - q.x) * (p.y - q.y) - (r.y - q.y) * (p.x - q.x);
   return comparar(result,0);
}

bool ccw(point p, point q, point r) { return turn(p,q,r) > 0; }

point pivot;

bool angle_cmp(point a, point b)
{
   if(turn(pivot, a, b) == 0)
         return distancia(pivot, a) < distancia(pivot, b);
   double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
   double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
   return atan2(d1x, d1y) - atan2(d2x, d2y) < 0;
}

vector<point> grahamScan(vector<point> P)
{
   int n = (int)P.size();
   if(n <= 3) return P;
   // 1. Buscamos el Y mas bajo y si hay empate, el más derecho. O(n)
   int lowY = 0, n = (int)P.size();
   for(int i = 1; i < n; i++)
       if(P[i].y < P[lowY].y || (P[i].y == P[lowY].y && P[i].x > P[lowY].x) )
           lowY = i;
   point temp = P[0]; P[0] = P[lowY]; P[lowY] = temp; pivot = P[0];
   sort(++P.begin(), P.end(), angle_cmp);
   // 2. Verificar ccw. O(n)
   stack<point> S;
   point prev, act;
   S.push(P[n - 1]); S.push(P[0]);
   int i = 1;
   while(i < n)
   {
      act = S.top(); S.pop();
      prev = S.top(); S.push(act); // Obtener el segundo de la pila
      if(ccw(prev, act, P[i])) S.push(P[i++]);
      else S.pop();
   }
   vector<point> convexHull;
```

```
    while(!S.empty()) { convexHull.push_back(S.top()); S.pop(); }
    convexHull.pop_back(); // Eliminamos el ultimo punto duplicado
    return convexHull;
}
```

MONOTONE CHAIN - O(N LOG N)

```
struct point { double x, y; };

int turn(point p, point q, point r)
{
  double result = (r.x - q.x) * (p.y - q.y) - (r.y - q.y) * (p.x - q.x);
  return comparar(result, 0);
}
bool cw(point p, point q, point r)
{
  return turn(p, q, r) < 0;
  // Para que NO acepte puntos colineares:
  // return turn(p, q, r) <= 0;
}

bool cmp(point p1,point p2)
{
  return p1.x < p2.x || (p1.x == p2.x && p1.y < p2.y);
}

vector<point> convexHull(vector<point> P)
{
  int n = (int)P.size(), k = 0;
  vector<point> H(2 * n);
  sort(P.begin(), P.end(),cmp);
  for(int i = 0; i < n; i++)                // Hull superior
  {
    while(k >= 2 && cw(H[k - 2], H[k - 1], P[i])) k--;
    H[k++] = P[i];
  }
  for(int i = n - 2, t = k + 1; i >= 0; i--) // Hull inferior
  {
    while(k >= t && cw(H[k - 2], H[k - 1], P[i])) k--;
    H[k++] = P[i];
  }
  H.resize(k); H.pop_back(); // Eliminamos el ultimo punto
  return H;
}
```

DIVIDE AND CONQUER

CLOSEST PAIR – O(N LG N)

```
#define INF 200000000

struct point
{
```

```
    double x, y;
    point(){}
    point(double px, double py){ x = px; y = py; }
};

double dist(point p1, point p2) { return hypot(p1.x - p2.x, p1.y - p2.y); }

double closestRecursive(vector<point> vx, vector<point> vy)
{
    if(vx.empty() || vy.empty()) return INF;           // No points to analyze
    if((int)vx.size() == 1) return INF;                // Only 1 point
    if((int)vx.size() == 2) return dist(vx[0], vx[1]); // Trivial case

    vector<point> vxL,vyL,vxR,vyR,b;
    double dL,dR,d;
    point cut = vx[vx.size() / 2]; // Midpoint

    /// Left side
    for(int i = 0; i < (int)vx.size(); i++)
       if(vx[i].x < cut.x || vx[i].x == cut.x && vx[i].y <= cut.y)
          vxL.push_back(vx[i]);
    for(int i = 0; i < (int)vy.size(); i++)
       if(vy[i].x < cut.x || vy[i].x == cut.x && vy[i].y <= cut.y)
          vyL.push_back(vy[i]);
    dL = closestRecursive(vxL, vyL);

    /// Right side
    for(int i = 0; i < (int)vx.size(); i++)
       if(!(vx[i].x < cut.x || vx[i].x == cut.x && vx[i].y <= cut.y))
          vxR.push_back(vx[i]);
    for(int i = 0; i < (int)vy.size(); i++)
       if(!(vy[i].x < cut.x || vy[i].x == cut.x && vy[i].y <= cut.y))
          vyR.push_back(vy[i]);
    dR = closestRecursive(vxR, vyR);

    /// Merge
    d = min(dL, dR);
    for(int i = 0; i < (int)vy.size(); i++) // For each point i..
       if(fabs(vy[i].x - cut.x) <= d) // Point i is close enough to cut point
          b.push_back(vy[i]);         // So, it's a possible candidate

    for(int i = 0; i < (int)b.size(); i++) // For each candidate point i..
       for(int j = 0; j < 7; j++) // At most 7 following points are candidates
          if(i + j + 1 < (int)b.size())
          {
             point p = b[i], q = b[i + j + 1];
             if(dist(p, q) < d) d = dist(p, q);
          }
    return d;
}

bool cmpXY(point p1, point p2)
{
    if(p1.x != p2.x) return p1.x < p2.x;
    return p1.y < p2.y;
}
```

```
bool cmpYX(point p1, point p2)
{
   if(p1.y != p2.y) return p1.y < p2.y;
   return p1.x < p2.x;
}

double closest(vector<point> p) // Distance of the closest pair
{
   vector<point> vx(p), vy(p);
   sort(vx.begin(), vx.end(), cmpXY); // Sort by x axis
   sort(vy.begin(), vy.end(), cmpYX); // Sort by y axis
   for(int i = 1; i < (int)vx.size(); i++) // Check for duplicate points
      if(vx[i - 1].x == vx[i].x && vx[i - 1].y == vx[i].y)
          return 0.0;
   return closestRecursive(vx, vy);
}
```

ROTATING CALIPERS

FARTHEST PAIR – O(N LG N)

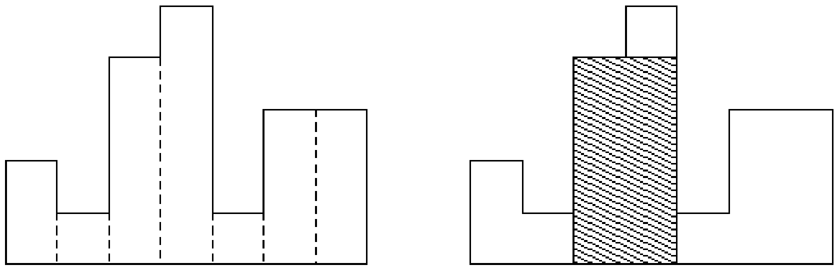
```
double cross(point p, point q, point r) // cross product of vectors qr and qp
{
  return (r.x - q.x) * (p.y - q.y) - (r.y - q.y) * (p.x - q.x);
}

double diameterOfPoints(vector<point> P) // Distance of the farthest pair
{
  vector<point> H = convexHull(P); // Modificar método cw: turn(p,q,r) <= 0;
  int n = (int)H.size();
  if(n == 1) return 0; // No hay diametro
  if(n == 2) return distancia(H[0], H[1]); // Devolver la distancia entre si
  double ans = 0;
  for(int k = 0, j = 2; k < n; k++)
  {
    // Encontrar el punto j mas lejano a la linea H[k] -> H[k + 1]
    while(cross(H[(j+1)%n],H[k],H[(k+1)%n]) > cross(H[j],H[k],H[(k+1)%n]))
      j = (j + 1) % n;
    // El punto j es antipodal a los puntos H[k] y H[k + 1]
    ans = max(ans, distancia(H[j], H[k]));
    ans = max(ans, distancia(H[j], H[(k + 1) % n]));
    // Si el punto j + 1 esta a la misma distancia, tambien es un antipodal
    if(cross(H[(j+1)%n], H[k], H[(k+1)%n]) == cross(H[j], H[k], H[(k+1)%n]))
    {
      ans = max(ans, distancia(H[(j + 1) % n], H[k]));
      ans = max(ans, distancia(H[(j + 1) % n], H[(k + 1) % n]));
    }
  }
  return ans;
}
```

MISCELANEA

LARGEST RECTANGLE IN A HISTOGRAM – O(N)

- **Entrada**: Vector con las alturas de cada barra.
- **Salida**: El área del mayor rectángulo que se puede formar.



```
int largestArea(const vector<int> &h)
{
   int n = (int)h.size(), t;
   vector<int> area(n, 0);
   stack<int> st;
   for(int i = 0; i < n; i++) // Calcular Li para cada barra
   {
      while(!st.empty())       // Las alturas >= a h[i] no serviran
         if(h[i] <= h[st.top()]) st.pop();
         else break;
      t = st.empty()? -1 : st.top();
      area[i] = i - t - 1;     // Mayor distancia hacia la izquierda
      st.push(i);              // Colocar en la pila
   }
   while(!st.empty()) st.pop();     // Limpiar pila
   for(int i = n - 1; i >= 0; i--) // Calcular Ri y sumarlo con Li
   {
      while(!st.empty())       // Las alturas >= a h[i] no serviran
         if(h[i] <= h[st.top()]) st.pop();
         else break;
      t = st.empty()? n : st.top();
      area[i] += t - i - 1;    // Mayor distancia hacia la derecha
      st.push(i);              // Colocar en la pila
   }
   int ans = 0;
   for(int i = 0; i < n; i++) // Calcular area[i]
   {
      area[i] = h[i] * (area[i] + 1); // area[i] = base[i] x altura[i]
      ans = max(ans, area[i]);        // Nos quedamos con la mayor area
   }
   return ans;
}
```

MAX UNAFFECTED RECTANGLE – O(N^2)

- **Entrada** : Matriz binaria, donde 1 representa un casillero libre y 0 representa uno ocupado.
- **Salida** : Área del mayor rectángulo que se puede abarcar solo considerando casilleros con valor 1.

Se reduce al problema del histograma. Requiere el código de la sección anterior.

```
int maxUnaffectedRect(vector<vector<int> > M)
{
   int area, R = (int)M.size(), C = (int)M[0].size();
   // Matriz acumulada (1 histograma por fila)
   for(int i = 1; i < R; i++)
      for(int j = 0; j < C; j++)
         if(M[i][j])
            M[i][j] = M[i - 1][j] + 1;
   // Calcular maxima area para cada fila
   int ans = 0;
   for(int i = 0; i < R; i++) ans = max(ans, largestArea(M[i]));
   // Recuperar matriz original
   for(int i = R - 1; i > 0; i--)
      for(int j = 0; j < C; j++)
         if(M[i][j])
            M[i][j] = M[i][j] - M[i - 1][j];
   return ans;
}
```

MAX UNAFFECTED SQUARE – O(N^2)

- **Entrada** : Matriz binaria, donde 1 representa un casillero libre y 0 representa uno ocupado.
- **Salida** : Área del mayor cuadrado que se puede abarcar solo considerando casilleros con valor 1.

```
#define MAX 105

int maxUnaffectedSquare(int M[][MAX], int R, int C)
{
   int S[MAX][MAX];
   for(int i = 0; i < R; i++) S[i][0] = M[i][0];
   for(int j = 0; j = C; j++) S[0][j] = M[0][j];
   // DP
   for(int i = 1; i < R; i++)
      for(int j = 1; j < C; j++)
         if(M[i][j] == 1)  // Take min of all neighbors
            S[i][j] = 1 + min(min(S[i][j - 1],
                                  S[i - 1][j]),
                                  S[i - 1][j - 1]);
         else S[i][j] = 0; // A mismatch means we won't find a square
   // Find max
   int ans = S[0][0];
   for(int i = 0; i < R; i++)
      for(int j = 0; j < C; j++)
         if(S[i][j] > ans)
            ans = S[i][j];
   return ans;
}
```

POSTFIX CONVERTION & CALCULATOR

La expresión infija a convertir debe cumplir:

- Tener números de 1 solo dígito.
- Tener variables de 1 sola letra.
- Solo usar los operadores suma (+), resta (-) y multiplicación (*)

Ejemplo: "3*a+c+(2*e)"

INFIX TO POSTFIX

```
bool hasGreaterPrecedence(char a, char b) // precedencia(a) >= precedencia(b)
{
   if((a == '+' || a == '-') && b == '*') return false;
   return true;
}

string toPostfix(string s) // s: expresion infija
{
   stack<char> q;
   string ans = ""; // ans: expresion postfija
   for(int i = 0; i < (int)s.size(); i++)
      if(s[i] >= '0' && s[i] <= '9') ans.push_back(s[i]);      // Operando
      else if(s[i] >= 'a' && s[i] <= 'z') ans.push_back(s[i]); // Variable
      else if(s[i] >= 'A' && s[i] <= 'Z') ans.push_back(s[i]); // Variable
      else if(s[i] == '(') q.push(s[i]); // Los ( van de inmediato a la pila
      else if(s[i] == ')')
      {
         // Buscar un ( en la pila, ir agregando los caracteres a la rpta
         while(q.top() != '(') { ans.push_back(q.top()); q.pop(); }
         q.pop(); // Eliminamos el ( de la pila
      }
      else if(s[i] == '+' || s[i] == '-' || s[i] == '*') // Operador
      {
         // Sacar operadores de la pila mientras no toque un (
         while(!q.empty() && q.top() != '(')
         {
            // Si el operador de la pila tiene mayor precedencia..
            if(hasGreaterPrecedence(q.top(), s[i]))
            {
               // Sacarlo de la pila y anexarlo a la respuesta
               ans.push_back(q.top()); q.pop();
            }
            else break;
         }
         q.push(s[i]);
      }
   // Lo que quede en la pila se anexa a la respuesta
   while(!q.empty()) { ans.push_back(q.top()); q.pop(); }
   return ans;
}
```

POSTFIX EVALUATION

```
int v[30]; // Valor de cada variable

int evaluatePostfix(string s) // s: expresion postfija
{
   stack<int> q;
   for(int i = 0; i < (int)s.size(); i++)
      if(s[i] == '+' || s[i] == '-' || s[i] == '*') // Operador
      {
         int n2 = q.top(); q.pop(); // Extraer 2do operando
         int n1 = q.top(); q.pop(); // Extraer 1er operando
         int ans;
         switch(s[i]) // Realizar operacion
         {
            case '+': ans = n1 + n2; break;
            case '-': ans = n1 - n2; break;
            case '*': ans = n1 * n2; break;
            default:  ans = -1;
         }
         q.push(ans); // Agregar a la pila
      }
      else if(s[i] >= '0' && s[i] <= '9') q.push(s[i] - '0');     // Operando
      else if(s[i] >= 'a' && s[i] <= 'z') q.push(v[s[i] - 'a']); // Variable
      // Case insensitive
      else if(s[i] >= 'A' && s[i] <= 'Z') q.push(v[s[i] - 'A']);

   return q.top();
}
```

ROMAN NUMERALS

El mayor número romano que se puede formar es 3999 que corresponde a MMMCMXCIX

CONVERTIR UN ENTERO A ROMANO

```
string intToRoman(int value)
{
   struct data_t { int value; char const* numeral; };
   static data_t const data[] = { 1000, "M", 900, "CM", 500, "D", 400, "CD",
                                   100, "C",  90, "XC",  50, "L",  40, "XL",
                                    10, "X",   9, "IX",   5, "V",   4, "IV",
                                     1, "I",   0, NULL };

   string result = "";
   for(data_t const* curr = data; curr->value > 0; ++curr)
      while(value >= curr->value)
      {
         result += curr->numeral;
         value  -= curr->value;
      }
   return result;
}
```

CONVERTIR UN ROMANO A ENTERO

```
int romanToInt(char *value)
{
   int data[26];
   data['I' - 'A'] = 1;   data['V' - 'A'] = 5;
   data['X' - 'A'] = 10;  data['L' - 'A'] = 50;
   data['C' - 'A'] = 100; data['D' - 'A'] = 500; data['M' - 'A'] = 1000;
   int result = 0;
   for(int i = 0; value[i]; i++)
      if(value[i + 1] && data[value[i] - 'A'] < data[value[i + 1] - 'A'])
      {
         result += data[value[i + 1] - 'A'] - data[value[i] - 'A'];
         i++;
      }
      else result += data[value[i] - 'A'];
   return result;
}
```

PARTITIONS OF A SET

```
int dp[1 << MAX];
int N; // N: Cantidad de elementos
int v[1 << MAX]; // v[i]: Valor asociado a la particion i

int partition(int mask)
{
   if(mask & (mask - 1) == 0) return v[mask]; // No hay mas particiones
   if(dp[mask] != -1) return dp[mask];        // Mascara ya procesada
   int m = (mask - 1) & mask;
   int best = v[mask];     // Procesar esta mascara
   while(m)
   {
      int best = min(best, partition(m) + partition(mask ^ m));
      m = (m - 1) & mask; // Siguiente particion
   }
   return dp[mask] = best;
}
```

En el main:

```
// Calcular los valores de v[i] para cada i
memset(dp, -1, sizeof dp);
printf("Respuesta: %d\n", partition((1 << N) - 1));
```

PERMUTATIONS

NTH LEXICOGRAPHIC PERMUTATION

```
typedef long long ll;

ll factorial[21];
```

```
// Calcula factoriales hasta 20
void init()
{
   factorial[0] = factorial[1] = 1;
   for(int i = 2; i < 21; i++) factorial[i] = factorial[i - 1] * i;
}

// Eliminar el caracter en la posicion 'pos' de la cadena 'c'
void deleteCharAtPos(int pos, char *c, int len)
{
   pos++;
   while(pos <= len) { c[pos - 1] = c[pos]; pos++; }
}

// Devuelve la n-esima permutacion (1-based index)
void nth_permutation(char *s, int len, ll n, char *p, int idx)
{
   if(len == 1) { p[idx++] = s[0]; p[idx] = '\0'; return; }
   int pos = 0; ll residue = n, numSuffixPermutations = factorial[len - 1];
   if(numSuffixPermutations < residue)
   {
      pos = residue / numSuffixPermutations;
      if(residue % numSuffixPermutations == 0) pos--;
      residue -= (numSuffixPermutations * pos);
   }
   p[idx] = s[pos]; deleteCharAtPos(pos, s, len);
   nth_permutation(s, len - 1, residue, p, idx + 1);
}

// Devuelve la n-esima permutacion de la cadena s y la guarda en p
void nth_permutation(char *s, ll n, char *p)
{
   nth_permutation(s, strlen(s), n, p, 0);
}
```

En el main:

```
init();
// n: Numero de permutacion, s: cadena original, p: cadena permutada
ll n; char s[21], p[21];
scanf("%s %lld\n",s,&n); sort(s, s + strlen(s)); // Leer y ordenar s
nth_permutation(s, n, p); puts(p); // Hallar p e imprimirlo
```

KD-TREE

```
template<class T,class K>
struct KDTree
{
   struct node
   {
      T a; K b;
      node *left,*right;
      bool del;
      node():del(0){}
   };
```

```
node *root, *null, *data;
int alloc;

KDTree()
{
   data = new node[1000000];
   null = new node; alloc = 0;
   null->left = null->right = null; root = null;
}

void clear() { root = null; alloc = 0; }

node* new_node(const T& a, const T& b)
{
   node* res = &data[alloc++];
   res->left = res->right = null;
   res->a = a; res->b = b;
   return res;
}

void insert(const T& a, const K& b) { insert(root, a, b, 1); }

void remove(const T& a, const K& b) { remove(root, a, b, 1); }

void query(const T& a1, const T& a2, const K& b1, const K& b2)
{
   query(root, a1, a2, b1, b2, 1);
}

void insert(node* &cur, const T& a, const K& b, int level)
{
   if(cur == null) cur = new_node(a,b);
   else if(level & 1)
   {
      if(a < cur->a) insert(cur->left, a, b, 1 - level);
      else insert(cur->right, a, b, 1 - level);
   }
   else
   {
      if(b < cur->b) insert(cur->left, a, b, 1 - level);
      else insert(cur->right, a, b, 1 - level);
   }
}

void remove(node* &cur, const T& a, const K& b, int level)
{
   if(cur == null)return;
   else if(level & 1)
   {
      if(a < cur->a) remove(cur->left, a, b, 1 - level);
      else if(cur->a < a) remove(cur->right, a, b, 1 - level);
      else cur->del = 1;
   }
   else
   {
```

```
         if(b < cur->b) remove(cur->left, a, b, 1 - level);
         else if(cur->b < b) remove(cur->right, a, b, 1 - level);
         else cur->del = 1;
      }
   }

 void query(node* &cur, const T& a1, const T& a2, const K& b1, const K& b2, int level)
   {
      if(cur==null) return;
      if(!cur->del && a1 <= cur->a && cur->a <= a2 && b1 <= cur->b && cur->b <= b2)
         cout << cur->a << " " << cur->b << endl;
      if(level & 1)
      {
         if(a1 <= cur->a) query(cur->left, a1, a2, b1, b2, 1 - level);
         if(cur->a <= a2) query(cur->right, a1, a2, b1, b2, 1 - level);
      }
      else
      {
         if(b1 <= cur->b) query(cur->left, a1, a2, b1, b2, 1 - level);
         if(cur->b <= b2) query(cur->right, a1, a2, b1, b2, 1 - level);
      }
   }
};
```

En el main :

```
KDTree<int,int> tree;
for(int i = 0; i < 100; i++)
   for(int j = 0; j < 100; j++)
      tree.insert(i, j);
tree.query(10, 20, 30, 40);
```