Contents

Notas programación competitiva

Plantilla para C++

Plantilla para cualquier programa en C++.

```cpp
#include <bits/stdc++.h>

using namespace std;

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);

    return 0;
}
```

La principal ventaja de esta plantilla es que el ***include*** que usa trae toda la libreria estándar, Yo siempre uso ***cin*** y ***cout*** así que esta plantilla incluye optimizaciones para el manejo de entrada. También hay que recordar que ***'\n'*** es más rápido que ***endl***.

Comando para compilar los programas de C++

Recomiendo este comando para compilar, a menos que el concurso sugiera uno.

```
g++ -O2 -Wall test.cpp
```

Con esto se optimiza el código además de indicar que se muestren la mayoría de advertencias.

Manejo de archivos

Si el concurso requiere que se lea o escriba en archivos, se puede agregar esto al principio.

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

Con esto ya no se ocupan modificar más cosas.

Tipos de dato y sus rangos

Tipo de dato	Tamaño (en bytes)	Rango
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long long int	8	-(2^63) to (2^63) - 1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
float	4	-3.4 × 10^38 to 3.4 × 10^38
double	8	-1.7 × 10^308 to 1.7 × 10^308
long double	12	-1.1 × 10^4932 to 1.1 × 10^4932

Algo a considerar sobre *long long* es que tiene un prefijo.

```
long long x = 123456789123456789LL;
```

También pasa algo parecido a la división entera cuando se multiplican dos enteros hacia un *long long*, pasa que, a pesar de que esa variable es *long long*, el resultado de multiplicar dos enteros es un entero, y no se guarda bien.

Números grandes y el módulo

A veces el problema pide que se de la respuesta módulo de algún número (cuando la respuesta puede ser muy grande), esto se puede manejar con una de las propiedades del módulo cuando se trata de adición, substracción o multiplicación.

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

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

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

Gracias a esta propiedad podemos estar sacando el módulo después de cada operación, por lo que el número nunca se hace demasiado grande. También, en caso de que haya restas, podemos evitar terminar con un residuo negativo si le sumamos m en caso de que el residuo sea menor a 0.

Números flotantes

En la mayoría de los casos con usar ***double*** es suficiente, si eso no basta se puede usar ***long double***. Pero algo a tener en cuenta cuando se usan números flotantes es que compararlos con == no siempre funciona, esto por pequeños *errores* de precisión, por lo que una forma de compararlos es ver si la diferencia entre ellos es lo suficientemente pequeña.

```
if (abs(a - b) < 1e-9) {
    // a and b are equal
}
```

Matemáticas

Sucesión simple

Fórmulas útiles para calcular la suma de números consecutivos, hay una fórmula mucho más general pero no la entendí.

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \left[\frac{n(n+1)}{2}\right]^2$$

Sucesión aritmetica

Otra cosa es una sucesión aritmetica, donde la diferencia entre dos números cualesquiera es la misma.

$$3, 7, 11, 15$$

En esta sucesión aumentan de 4 en 4. La formula se ve así.

$$a + \cdots + b = \frac{n(a+b)}{2}$$

Es esta fórmula a es el primer número, b el segundo y n es la cantidad de números, se puede ver que en la fórmula no se considera el tamaño de los saltos.

Sucesión geometrica

Esta sucesión es una secuencia de números donde la proporción entre dos números consecutivos es la misma.

$$a + ak + ak^2 + \cdots + b = \frac{bk - a}{k - 1}$$

Tabla de verdad

A	B	not A	not B	A and B	A or B	A implies B	A equals B
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

Factorial

Se puede definir iterativamente.

$$\prod_{x=1}^{n} x = 1 * 2 * 3 * \cdots * n$$

O recursivamente.

$$0! = 1$$

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

Números de Fibonacci.

Se define recursivamente así.

$$f(0) = 0$$

$$f(1) = 1$$

$$f(n) = f(n-1) + f(n-2)$$

También hay una fórmula directa para calcularlos.

$$f(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$$

Logaritmos

Propiedades de los logaritmos.

$$log_k(x) = a \implies k^a = x$$

$$log_k(ab) = log_k(a) + log_k(b)$$

$$log_k(x^n) = n * log_k(x)$$

$$log_k(\frac{a}{b}) = log_k(a) - log_k(b)$$

Otra propiedad interesante es que la cantidad de digitos de un entero x en base b es:

$$log_b(x) + 1$$

Estimando la eficiencia

Aquí hay una tabla para tener una idea de la complejidad necesaria según el tamaño de la entrada.

tamaño de entrada	complejidad requerida
n <= 10	O(n!)
n <= 20	O(2^n)
n <= 500	O(n^3)
n <= 5000	O(n^2)
n <= 10^6	O(n*logn) o O(n)
n es grande	O(1) o O(logn)

Esto no es para tomarlo como la verdad absoluta, pero puede servir para descartar ideas sin desperdiciar tiempo.

Suma máxima de subarreglo

Aquí se habla de un problema clásico cuya solución más *directa* es O(n^3), pero que pensandola mejor se puede lograr reducir a O(n). Dado un arreglo de n númeors, hay que calcular cual es la suma máxima de una subsección, osea la suma más grande de una secuencia consecutiva de valores en el arreglo, esto se vuelve más interesante si el arreglo puede tener números negativos. Aquí hay un ejemplo.

-1 | 2 | 4 | -3 | 5 | 2 | -5 | 2 |

El siguiente subarreglo produce una suma de 10:

-1	2	4	-3	5	2	-5	2
	^	^	^	^	^		

Asumimos que un subarreglo vacio esta permitido, por lo que la suma máxima siempre será al menos 0. Ahora, para resolverlo en O(n) partimos de la idea de que para cada posición queremos calcular la suma máxima posible hasta ahí, entonces la del arreglo completo será la mayor de todas esas sumas. We assume that an empty subarray is allowed, so the maximum subarray sum is always at least 0. Surprisingly, it is possible to solve the problem in O(n) time, which means that just one loop is enough. The idea is to calculate, for each array position, the maximum sum of a subarray that ends at that position. After this, the answer for the problem is the maximum of those sums. Consider the subproblem of finding the maximum-sum subarray that ends at position k. There are two possibilities:

1. The subarray only contains the element at position k.
2. The subarray consists of a subarray that ends at position k - 1, followed by the element at position k.

In the latter case, since we want to find a subarray with maximum sum, the subarray that ends at position k - 1 should also have the maximum sum. Thus, we can solve the problem efficiently by calculating the maximum subarray sum for each ending position from left to right. The following code implements the algorithm:

```
int best = 0, sum = 0;
for (int k = 0; k < n; k++) {
    sum = max(array[k],sum+array[k]);
    best = max(best,sum);
}
cout << best << '\n';
```

The algorithm only contains one loop that goes through the input, so the time complexity is O(n). This is also the best possible time complexity, because any algorithm for the problem has to examine all array elements at least once.

Sorting in C++

It is almost never a good idea to use a home-made sorting algorithm in a contest, because there are good implementations available in programming languages. For example, the C++ standard library contains the function sort that can be easily used for sorting arrays and other data structures. The following code sorts a vector in increasing order:

```
vector<int> v = {4,2,5,3,5,8,3};
sort(v.begin(),v.end());
```

After the sorting, the contents of the vector will be [2, 3, 3, 4, 5, 5, 8]. The default sorting order is increasing, but a reverse order is possible as follows:

```
sort(v.rbegin(),v.rend());
```

User-defined structs

User-defined structs do not have a comparison operator automatically. The operator should be defined inside the struct as a function operator<, whose parameter is another element of the same type. The operator should return *true* if the element is smaller than the parameter, and *false* otherwise. For example, the following struct P contains the x and y coordinates of a point. The comparison operator is defined so that the points are sorted primarily by the x coordinate and secondarily by the y coordinate.

```
struct P {
    int x, y;
    bool operator<(const P &p) {
        if (x != p.x) return x < p.x;
        else return y < p.y;
    }
};
```

Comparison functions

It is also possible to give an external comparison function to the sort function as a callback function. For example, the following comparison function *comp* sorts strings primarily by length and secondarily by alphabetical order:

```
bool comp(string a, string b) {
    if (a.size() != b.size()) return a.size() < b.size();
    return a < b;
}
```

Now a vector of strings can be sorted as follows:

```
sort(v.begin(), v.end(), comp);
```

Binary search

Manual approach

There are two ways to implement binary search by ourselves

```
int a = 0, b = n - 1;
while (a <= b) {
    int k = (a + b) / 2;
    if (array[k] == x) {
        // x found at index k
    }
    if (array[k] > x) b = k-1;
```

```
    else a = k+1;
}
```

And.

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (k + b < n && array[k + b] <= x) k += b;
}
if (array[k] == x) {
    // x found at index k
}
```

C++ functions

The C++ standard library contains the following functions that are based on binary search and work in logarithmic time:

- lower_bound returns a pointer to the first array element whose value is at least x.
- upper_bound returns a pointer to the first array element whose value is larger than x.
- equal_range returns both above pointers.

The functions assume that the array is sorted. If there is no such element, the pointer points to the element after the last array element.