Contents

Notas programación competitiva

Plantilla para C++

Plantilla para cualquier programa en C++.

```
#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:

$$log_b(x) + 1$$

Promedio

Se puede sumar o restar un número a un promedio ya calculado sin tener que recalcular la suma original

$$s = \frac{a_1 + \cdots + a_n}{n}$$

$$s' = \frac{a_1 + \cdots + a_n + a_{n+1}}{n+1} = \frac{ns + a_{n+1}}{n+1} = \frac{(n+1)s + a_{n+1}}{n+1} - \frac{s}{n+1} = s + \frac{a_{n+1} - s}{n+1}$$

$$s'' = \frac{a_1 + \cdots + a_{n-1}}{n-1} = \frac{ns - a_n}{n-1} = \frac{(n-1)s + a_n}{n-1} + \frac{s}{n-1} = s + \frac{s - a_n}{n-1}$$

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. Consideramos dos posibilidades para la máxima suma del arreglo que termina en la posición k:

1. El subarreglo solo contiene el elemento de la posición k.
2. El subarreglo consiste de un subarreglo que termina en la posición k - 1, seguido del elemento en la posición k.

En el segundo caso, dado que queremos encontrar un subarreglo con la máxima suma, el subarreglo que termina en la posición k - 1 debería tener también la máxima suma para ese punto. Entonces podemos resolver el problema al calcular la máxima suma de subarreglos para cada posición de izquierda a derecha. El siguiente código demuestra una implementación de este algoritmo:

```
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';
```

Este algoritmo solo tiene un ciclo, por lo que la complejidad es O(n). Esta es la mejor complejidad posible, ya que cualquier algoritmo para este problema tiene que analizar todos los datos al menos una vez.

Ordenando en C++

Casi nunca es una buena idea usar algoritmos de ordenamiento que hayas hecho a mano, esto porque ya hay buenas implementaciones en los lenguajes de programación. Por ejemplo, la STL de C++ ya tiene una función *sort* que puede ser usada para ordenar (es una combinación de quicksort, heapsort, insertion sort), también existe *stable_sort* en caso de que se necesite un ordenamiento estable. Aquí hay un ejemplo de esta función en acción.

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

Después de este ordenamiento, el contenido del vector será [2, 3, 3, 4, 5, 5, 8]. Por defecto se ordena de menor a mayor, pero una forma de implementar un orden inverso es:

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

Estructuras definidas por el usuario

Las estructuras que nosotros hagamos no tienen un operador de comparación automaticamente. Este operador se puede definir dentro de la estructura como una función de nombre *operator<* cuyo parametro debe ser un elemento del mismo tipo, debe devolver *true* si el elemento es más pequeño que el parametro y *falso* si no es así. Un ejemplo:

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

Funciones de comparación

También se puede dar como parametro una función externa para que se use dentro del *sort*. Por ejemplo:

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

Ahora usando esa función para un vector de strings:

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

Búsqueda binaria

Forma manual

Hay dos formas de implementar la búsqueda binaria por nuestra cuenta.

```
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;
}
```

Y.

```
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
}
```

Funciones de C++

La libreria estandar de C++ contiene las siguientes funciones que estan basadas en búsqueda binaria y por lo tanto funcionan en tiempo logaritmico:

- lower_bound devuelve un apuntador al primer elemento que vale al menos x.
- upper_bound devuelve un apuntador al primer elemento que vale más que x.
- equal_range devuelve los dos apuntadores de arriba.

Estas funciones asumen que el arreglo esta ordenado. Si no encuentran el elemento devuelven un apuntador pasado el último elemento.