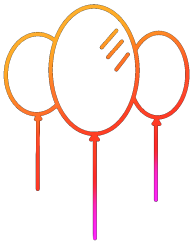
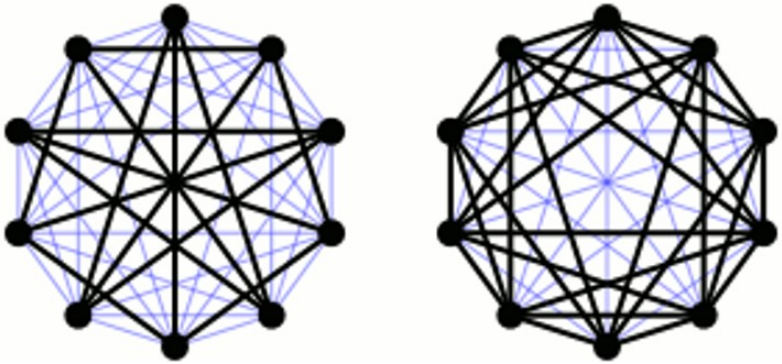


Programación Competitiva

Introducción



Edgar Omar Cruz Gutiérrez
Oscar Cabrera Rojas
CPCFI, Facultad de Ingeniería, UNAM

CONTENIDOS

Este libro fue escrito por el **Club de Programación Competitiva de la Facultad de Ingeniería** (**CPCFI**), UNAM, como complemento y acompañamiento al curso propedéutico ofrecido por el mismo, en él se presenta una compilación de conocimientos útiles para la práctica de la programación competitiva, así para alentar a las nuevas generaciones en el club a comenzar a emprender su viaje por este maravilloso mundo.

1

¿QUÉ ES LA PROGRAMACIÓN COMPETITIVA?

La programación competitiva es un deporte mental en donde se busca resolver problemas en un entorno controlado, en la menor cantidad de tiempo posible y bajo ciertos requisitos de:

- Complejidad espacial. (memoria)
- Complejidad temporal. (operaciones que toma a un algoritmo entregar un resultado correcto)

La verdadera meta de la programación competitiva es producir programadores, científicos e informáticos polifacéticos, *todo-terreno* que estén mucho más preparados para producir mejor software, y de enfrentar problemas de investigación de ciencias de la computación más difíciles en el futuro. Para ésto combina dos conceptos: diseño de algoritmos, e implementación de algoritmos.

El núcleo de la programación competitiva es acerca de inventar algoritmos eficientes que sean capaces de resolver problemas computacionales bien definidos. El diseño de algoritmos requiere resolución de problemas, y habilidad matemática. Muy seguido la solución a un problema es la combinación

de métodos bien conocidos acompañados con la pericia del programador.

Las matemáticas juegan un papel importante en la programación competitiva, de hecho, no hay límites claros entre el diseño de algoritmos y las matemáticas, aún así, es posible realizar una introducción a la programación competitiva sin entrar en detalles matemáticos, cosa que será inevitable conforme el atleta quiera adentrarse. Los conceptos matemáticos fundamentales para comenzar incluyen: teoría de conjuntos, lógica y funciones.

Por otro lado, la implementación de algoritmos es relevante debido a que las soluciones a los problemas son evaluadas probando el algoritmo implementado en diferentes pruebas que usan un conjunto de casos de prueba. Ésto implica que, después de dar con un algoritmo que resuelva el problema, el siguiente paso es implementarlo correctamente. La programación competitiva difiere enormemente de la ingeniería de software tradicional: Los programas son cortos, usualmente tienen como máximo unos cientos de líneas de código, debe ser escrito rápidamente y no hay necesidad de darle mantenimiento pasada la competencia.

Hasta el momento el lenguaje de programación más popular más utilizado en competencias es C++ debido a que es un lenguaje muy eficiente, y a su librería que contiene un gran colección de estructuras de datos y algoritmos. Por ello es el lenguaje en el que cualquier ejemplo pertinente se muestra en este libro. Si aún no eres capaz de programar en C++, es un gran momento para empezar a aprender.

2

ALGUNAS DEFINICIONES

2.1 ALGORITMO

De manera informal se define como un procedimiento finito(tiene un término definido) en donde se toman datos como entrada (*input*) y después de realizar una serie de operaciones sobre ellos, se entrega una salida (*output*).

2.2 COMPLEJIDAD TEMPORAL

En términos de computación se debe entender como complejidad y su análisis, a la estimación de recursos que le tomará a un determinado algoritmo el completar una tarea; aunque estos recursos pueden ser de diferente índole, nos interesan principalmente los recursos de almacenamiento (memoria) y de ejecución (tiempo), siendo este último el más importante en la mayoría de los casos.

El análisis de complejidad se realiza con la intención de verificar si nuestra solución además de ofrecer un resultado correcto, cumple con los requisitos de tiempo y memoria que el jurado nos ha proporcionado; esto implica que las soluciones no son únicas.

Dado que dependiendo de los recursos de cada máquina (procesador, tareas simultaneas, etc.) un algoritmo puede terminar una tarea en una cantidad de tiempo variable, suele ser más preciso utilizar la cantidad de operaciones como un indicador del tiempo que tomará finalizar la ejecución de un algoritmo.

Dentro de las restricciones clásicas de un problema de programación competitiva, podemos encontrar que nuestra solución debe proporcionar una respuesta correcta en **1000 ms** (**1 segundo**), dado que el comportamiento de cada máquina es distinto, podemos asumir que en promedio un procesador puede llevar a cabo

$$10^8$$

operaciones por segundo, por lo tanto nuestra solución debe realizar internamente a lo más esa cantidad de operaciones.

Usualmente muchos de los problemas involucrados en la programación competitiva (CP, por sus siglas en inglés *competitive programming*) requieren que se revise cada uno de los objetos de una lista o conjunto, es decir; si contamos con la lista que contiene los números enteros

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

e intentamos verificar mediante un algoritmo si dentro de esa lista se encuentra el número diez, podemos descomponer la tarea de la siguiente manera:

Primero almacenamos nuestro conjunto de enteros en una estructura. Podemos pensar esta como un tren en donde cada vagón corresponde a un contenedor que almacenará un único número.

En la Fig. 1 podemos observar la representación visual, donde cada rectángulo representa un *vagón*:

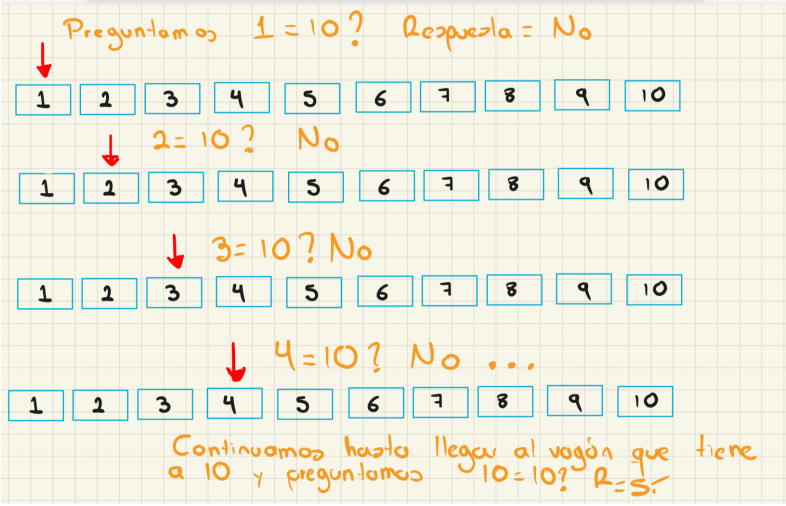


Figura 1: Procedimiento de la búsqueda lineal en una colección de los primeros diez números naturales.

Si prestamos atención notaremos que cada vez que preguntamos si un vagón contiene al número diez estamos, en efecto, realizando una operación (una operación de comparación en este caso), por lo tanto si el elemento que buscamos se encuentra en el último vagón habremos realizado diez operaciones en total, esto también aplica si el elemento simplemente no se encuentra dentro de la lista, ya que se debe buscar en cada uno de los vagones para poder determinar su inexistencia.

Tal como podemos llegar a pensar existen escenarios en donde encontrar un elemento no requiera buscar a través de todos los contenedores. Por ejemplo si el elemento que buscamos es el número uno, únicamente se llevaría a cabo una operación. Sin embargo, no podemos basar nuestros algoritmos siendo optimistas y prediciendo que el elemento siempre se encontrará en su *primer intento* (**mejor caso**); pensar que el elemento se encontrará en una posición intermedia en toda la colección (**caso promedio**) nos haría cometer predicciones incorrectas cuando sea el turno de buscar al diez, como en el ejemplo. Cuando realizamos un análisis de complejidad debemos centrarnos en el **peor caso**, que se modelará con la notación *Big O*.

Dado el caso anterior, nuestro peor escenario se ejemplifica cuando el elemento buscado se encuentra en la última posición, en cuyo caso tendríamos que realizar una comparación por cada vagón del tren. Entonces, si contamos con n vagones (siendo n un número natural), encontrar un elemento nos puede llegar a tomar , en efecto, n operaciones y su expresión en notación Big O es: $O(n)$.

Para cerrar el ejemplo, ¿qué sucedería si contamos con 10^9 elementos? Esto implicaría que tendríamos que realizar 10^9 operaciones, sin embargo, ¿qué implicaciones acarrea que la tarea tenga que ser realizada en 1 segundo? Tal como mencionamos anteriormente un procesador en promedio puede realizar 10^8 operaciones, por lo tanto no sería posible ejecutar la tarea dados los requisitos brindados por el jurado.

Una conclusión importante de este resultado es que nuestra solución no sería incorrecta ya que proporciona la respuesta esperada para el problema dado, sin embargo, sí se considera

ineficiente al no ser capaz de hacerlo con las restricciones dadas. Es por tal motivo que se requiere realizar este tipo análisis de complejidad, dado que en ocasiones no es suficiente con que la solución entregue el resultado esperado, sino que además es imprescindible que lo haga de una manera óptima.

La estrategia que llevamos a cabo a través de este ejemplo recibe el nombre de **búsqueda lineal**, si continuamos nuestro viaje por la CP nos daremos cuenta que existen maneras más eficientes de realizar la misma tarea, tal como la **búsqueda binaria**, en donde la complejidad temporal en notación Big O es igual a $O(log(n))$, donde el logaritmo se asume en base dos. Ésto reduce significativamente el tiempo de ejecución (podemos comprobar fácilmente en nuestra calculadora que para un arreglo con 10^9 elementos tomaría aproximadamente 30 operaciones a dicho algoritmo encontrar una respuesta dada contra las 1,000,000,000 de operaciones que le tomaría a la búsqueda lineal).

En la Tab. 1 se presenta una tabla de referencia de la máxima cantidad de elementos que pueden procesarse según diferentes complejidades expresadas en notación Big O, y su aplicación en algoritmos populares.

2.3 COMPLEJIDAD ESPACIAL

Este concepto hace referencia a la cantidad de almacenamiento que requiere nuestro algoritmo para funcionar, dicha cantidad se refiere a las estructuras de datos que utilicemos en la lógica de nuestra solución, en específico, el tamaño que van a tener y el tipo de dato que van a contener.

n	Peor algoritmo AC	Comentario
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Contando permutaciones
$\leq [15..18]$	$O(2^n \times n^2)$	e.g. DP TSP[1]
$\leq [18..28]$	$O(2^n \times n)$	e.g. DP
100	$O(n^4)$	e.g. DP con 3 dimensiones + ciclo $O(n)$, ${}_nC_{k=4}$
400	$O(n^3)$	e.g. Algoritmo Floyd Warshall
$2K$	$O(n^2 log_2 n)$	e.g. 2 ciclos anidados + una ED de árbol relacionada.
$10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort
$1M$	$O(n log_2 n)$	e.g. Merge Sort, construcción de Segment Tree
$100M$	$O(n), O(log_2 n), O(1)$	La mayoría de problemas en contest tienen $n \leq 1M$ (cuello de botella por I/O)[2]

[1] Técnica de programación dinámica aplicada en el popular problema del viajante (*TSP*).

[2] *Input / Output*.

Tabla 1: Regla general en complejidades temporales para el *peor algoritmo aceptado (AC)* para varios test case individuales con tamaño de entrada n, asumiendo un CPU capaz de computar 100M de operaciones en tres segundos.[1]

A pesar de que actualmente son pocos los escenarios en donde nos encontraremos con limitantes en el almacenamiento que requieran que optimicemos nuestro código, es importante tener al menos un entendimiento básico de cómo funciona el mismo en memoria.

Cualquier tipo de dato que se encuentre almacenado en una computadora puede ser representado a través del sistema binario, los ejemplos más sencillos de entender son aquellos relacionados con números enteros, por ejemplo:

El número 1 puede ser representado como 00000001, donde cada una de las posiciones del número anterior representa

una potencia de 2. En la Fig. 2 agreguemos una representación visual de lo anterior para entenderlo de mejor manera.

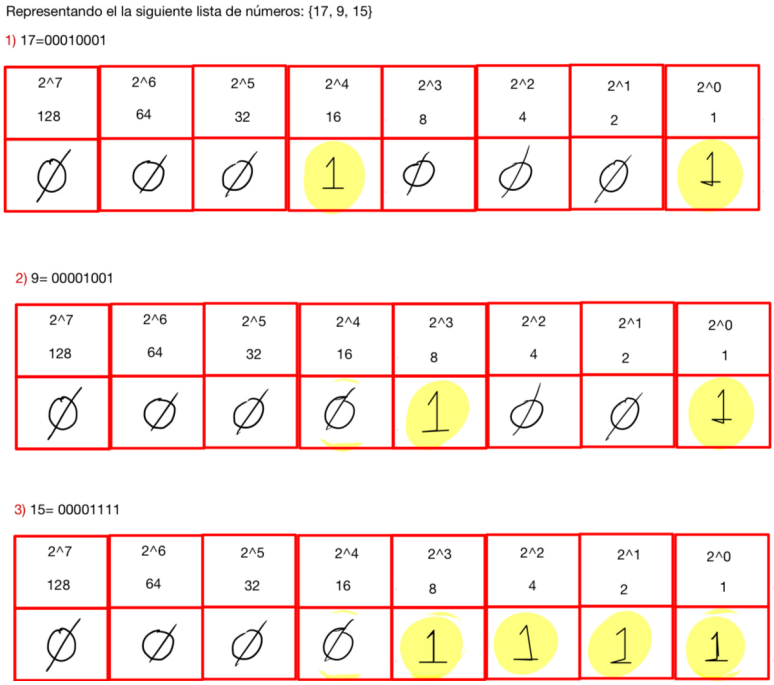


Figura 2: Representación de los números 17, 9 y 15 en código binario.

Dado el ejemplo anterior nosotros podríamos comenzar a preguntarnos cuál es el intervalo al que pertenece al conjunto de los números naturales (*unsigned int*), cuyo valor podemos representar con esos 8 bits utilizados en nuestra tabla, y la respuesta es de 0 hasta 255. Si quisiéramos representar números enteros (aquellos con signo o *signed*) podemos representar un intervalo de -128 hasta 127.

Usualmente un número entero ocupa 4 bytes de memoria (4 bytes = 32 bits) con lo cual podemos representar el intervalo de números `[-2,147,483,648, 2,147,483,647]` en caso de que se tome en cuenta el signo (*signed int*) o el intervalo de `[0, 4,294,967,295]` en caso de que no.

Es extremadamente importante que mantengamos en nuestra memoria la capacidad de números que pueden representar ciertos tipos de datos como *signed int*, *unsigned int*, *long long*, *float*, *double*, *short*, y el resto de tipos primitivos. Si buscamos almacenar el número 5,000,000,000 en un tipo de dato int, sea *signed* o *unsigned*, causará un desbordamiento y por lo tanto un error en nuestro programa. En la Tab. 2 se muestra los valores mínimos y máximos que toman los tipos de datos primitivos en C++.

Tipo de dato	[B]	Valor mínimo que toma	Valor máximo que toma
bool	1	0	1
signed char	1	0	255
unsigned char	1	-128	127
signed int	4	-2,147,483,648 $\approx -2 \times 10^9$	2,147,483,647 $\approx 2 \times 10^9$
unsigned int	4	0	4,294,967,295 $\approx 4 \times 10^9$
signed short	2	-32,768	32,767
unsigned short	2	0	65,535
signed long long int	8	-9,223,372,036,854,775,808 $\approx -9 \times 10^{18}$	9,223,372,036,854,775,807 $\approx 9 \times 10^{18}$
unsigned long long int	8	0	18,446,744,073,709,551,615 $\approx 18 \times 10^{18}$
float	4	$1{,}1 \times 10^{-38}$	$3{,}4 \times 10^{38}$
double	8	$2{,}2 \times 10^{-308}$	$1{,}7 \times 10^{308}$
long double	12	$3{,}3 \times 10^{-4932}$	$1{,}1 \times 10^{4932}$

Tabla 2: Capacidad de tipos de datos primitivos en C++.

Si conocemos el tamaño en bytes que requiere un tipo de dato para ser almacenado, podemos calcular la cantidad de memoria aproximada que será requerida para la ejecución de nuestro algoritmo, por ejemplo, si regresamos al caso donde teníamos una lista de 10 números enteros almacenados en

nuestra memoria y sabemos que cada número entero requiere de 4 bytes para ser representado, podemos calcular fácilmente que nuestro programa requerirá de aproximadamente 40 bytes de memoria para ejecutarse. Este límite está muy por debajo de los requisitos que usualmente nos son exigidos dentro de una competencia, donde usualmente se nos proporcionan cantidades que rondan los 64 Mb (64 000 000 de bytes aproximadamente, o visto de otra forma, la capacidad para albergar alrededor de 16 000 000 de enteros).

Tal como podemos observar, es complicado que los algoritmos que implementemos no cumplan con los requisitos que nos solicita el jurado, sin embargo, esto no nos exime de aprender al menos lo básico, ya que como se mencionó, es en exceso frecuente que un mal manejo del tipo de dato y sus límites no permita que obtengamos el título de Aceptado en nuestra respuesta.

3

CREACIÓN DE UNA PLANTILLA PROPIA

Cuando hablamos de desarrollar una plantilla nos referimos a crear un *archivo* en el cual podamos definir atajos con el objetivo de destinar la menor cantidad de tiempo a implementar la solución de un problema determinado, cualquier atleta eventualmente llega al punto en el que optimizar uno o dos minutos en escribir la solución a problemas conocidos se vuelve crucial. Después de resolver cierta cantidad de problemas (un número bajo en realidad) será posible que identifiquemos fragmentos de código que pueden ser reutilizados en un gran abanico de situaciones.

Por ejemplo, gran parte de los problemas hallados en los jueces en línea cuentan con un número de casos de prueba (o *test cases* como se les llamará por convención general) para los cuales nuestro algoritmo debe procesar cierta entrada y entregar una salida válida. Es por tal motivo que surge la necesidad de implementar en nuestra plantilla las instrucciones para completar esta tarea de manera automática, de tal forma que esos segundos sean aprovechados en generar una solución y no en escribir código repetitivo.

La creación de estos atajos o *shortcuts* se realiza indicando a nuestro programa que ciertos elementos (aquellos repetitivos y recurrentes) queremos sean reconocidos a través de una forma distinta a como se hace originalmente (atajo), y para ello existen distintos mecanismos dependiendo del tipo de información que queremos abreviar, como se verá a continuación.

El primer elemento que es conveniente definir en una plantilla es la indicación de utilización del *namespace* estándar para los elementos de la biblioteca estándar.

Los namespace son agrupaciones lógicas de identificadores que permite evitar conflictos de nombres en grandes proyectos o bibliotecas, sirve para evitar conflictos entre nombres de elementos definidos por el programador, y los proporcionados por la biblioteca estándar, al utilizar la sentencia

```
using namespace std;
```

se hace la indicación al compilador para que se puedan utilizar directamente todos los elementos dentro del espacio de nombres `std`.

Un ejemplo de implementación de un namespace propio se muestra en el Lst. 3.1 con el fin de asentar la definición de éstos.

```
1 namespace MiEspacio {
2     void saludar() {
3         std::cout << "Hola desde MiEspacio!" << std::endl;
4     }
5 }
6
```

```
7  int main() {
8      MiEspacio::saludar(); // Uso del namespace explí
       citamente.
9      return 0;
10 }
```

Listado 3.1: Uso de un namespace propio.

Incluir la utilización de `std` hace que podamos utilizar las funciones estándar directamente, sin tener que indicar explícitamente que estamos utilizando los elementos estándares. Como se puede ver a continuación, hace que el código sea mucho más fácil de entender y de escribir.

```
1 std::cout << "Hola, mundo!" << std::endl;
2 // Ahora habiendo hecho using namespace std;
3 cout << "Hola, mundo!" << endl;
```

El segundo elemento conveniente de incluir en una plantilla es la importación de la biblioteca estándar de C++ (*Standard Library*), que contiene las librerías más utilizadas: `iostream`, `vector`, `algorithm`, `map`, `set`, `queue`, `stack`, `deque`, `string`, `cstring`, `cmath`, `cstdio`, `cstdlib`, `ctime`, `cassert`. Esto se hace mediante la línea

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

Utilizarla facilita la programación de soluciones al no tener que escribir uno por uno los imports utilizados y disponer de las funciones necesarias sin preocuparse por dónde están. Sin embargo, cabe destacar que esta práctica no es estándar y se recomienda únicamente en contextos como la programación competitiva. Además, la disponibilidad de este encabezado

depende de la configuración específica del compilador y no está garantizada en todas las versiones de GCC, aunque es común en entornos basados en g++.

3.1 NOMENCLATURA DE DATOS CON 'TYPEDEF'

Aparte de la optimización de tiempo al momento de escribir soluciones es importante mantener un *código limpio*, dentro de la medida de lo posible, sobre todo para aquellos problemas cuya solución no es trivial y necesite de un código ciertamente más *robusto*.

La palabra reservada (en C++) `typedef` permite renombrar un tipo de dato existente en una manera propia, usualmente contracta.

Un uso donde se ejemplifica de manera muy clara el uso de `typedef` es al utilizar tipos de dato `struct`. Muy probablemente te habrás topado o has utilizado tú mismo la siguiente notación al momento de trabajar con estructuras en C o C++:

```
1 typedef struct {
2     // miembros
3 } structConTypedef;
```

En lugar de trabajarlas como:

```
1 struct {
2     // miembros
3 } structSinTypedef;
```

¿Por qué? Habrás notado que ambas formas de definir la estructura difiere en el uso de `typedef`. La primer forma mostrada suele ser más popular debido a la comodidad de

declarar variables de tipo `structConTypedef` (nombre de la estructura) directamente, sin tener que referenciar que el identificador es también del tipo de dato `struct`, entonces la forma de declarar (y de manipular en general) ambas estructuras se haría de la forma:

```
1 structConTypedef myStruct;
2 struct structSinTypedef myOtherStruct;
```

Lo que estamos haciendo en el primer caso es renombrar el tipo de dato `struct structConTypedef` como simplemente `structConTypedef`, lo que nos permite declarar variables directamente como del tipo `structConTypedef`, y el compilador entenderá directamente que nos referimos también a un tipo de dato `struct`.

¿Cómo entonces le podemos exprimir el potencial para nuestras soluciones? A simple vista puede no parecer muy útil. *¿Por qué querría contraer la palabra 'int', o 'float'?* preguntará el programador novato y con cierta razón, el potencial más grande reside en su uso para renombrar estructuras de datos.

Una de las ventajas del lenguaje C++ y por el cuál lo utilizamos para la programación competitiva, además de su velocidad, es debido a las implementaciones construidas funcionales listas para usar de las estructuras de datos incluidas en la STL y la comodidad de su utilización mediante una interfaz de alto nivel.

La estructura más recurrida en la resolución de problemas de tipo programación competitiva son los vectores (arreglos dinámicos redimensionables), y la forma convencional para declararlos de, por ejemplo enteros, es la siguiente:

```
// Sintaxis: typedef tipo_original alias;
typedef vector<int> vi;
```

Aún no se ve nada demencial, pero ¿y si necesitáramos trabajar con un par, o con un vector de pares? ¿Y con un vector de vector de pares? Dichos tipos de dato son labor del día a día para el atleta de programación competitiva, y la gran recurrencia de la escritura de estas líneas hacen valioso el uso del renombramiento de tipos de datos. En el Lst. 3.2 se muestra un uso bastante real de cómo se pueden aprovechar las bondades de `typedef` para codificar de manera más clara y fluida.

```
typedef long long ll; // Enteros largos
typedef long double ld; // Flotantes de doble precisión
    largos
typedef pair<int,int> ii; // Par de enteros
typedef pair<ll,ll> pll; // Sí, se puede anidar su uso
typedef vector<int> vi; // Vector de enteros
typedef vector<ll> vll; // Vector de long long's
typedef vector<ii> vii; // Vector de pares de enteros
typedef vector<vi> vvi; // Vector de vector de enteros
typedef vector<vii> vvii; // Vector de vector de pares de
    enteros
```

Listado 3.2: Usos más comunes de `typedef` en la construcción de plantillas.

3.2 LA DIRECTIVA 'DEFINE'

La palabra reservada `define` es una directiva de preprocesamiento utilizada para crear constantes y macros. Para comprender estos usos es relevante decir que `define` funciona a través de sustitución textual, lo que quiere decir que, cuando halle una coincidencia en la sintaxis definida, tratará dicha coincidencia sustituyéndola directamente por lo indicado.

Es utilizada para acortar ciertos elementos repetitivos, por ejemplo, el nombre de miembros en ED, o de ciertos métodos. Ya hablamos de los vectores y los pares, en el Lst. 3.3 se muestra el uso para acortar la llamada a los miembros `first` y `second` del tipo de dato `pair`, y del método `push_back` tan recurrente de los vectores.

```
1 // Al ser una directiva, no se debe de escribir el punto y
      coma (;) después de su uso, otra directiva de uso
     recurrente y que puede servir de ejemplo es #include.
2 /* Sintaxis: #define NOMBRE valor
3 'NOMBRE' es la coincidencia buscada en el programa, 'valor
      ' la sustitución realizada tras la coincidencia. */
4 #define fi first
5 #define se second
6 #define pb push_back
```

Listado 3.3: Usando la sustituación textual de `define`.

En el Lst. 3.4 se muestra un uso de ejemplo para ambos elementos, `typedef` y `define` en un ejemplo donde se utilicen vectores y pares en sus contracciones.

```
vi v;
ii p;
for(int i = 0; i < 10; i++) {
    p.fi = i;
    p.se = i+1;
    v.pb( p.fi * p.se );
}
// Su equivalente sin usar contracciones
vector<int> v;
pair<int,int> p;
for(int i = 0; i < 10; i++) {
    p.first = i;
    p.second = i+1;
    v.push_back( p.first * p.second );
}
```

Listado 3.4: Uso de vectores y pares optimizado en un ejemplo sencillo.

Aún así, la directiva esconde un uso adicional en su naturaleza, el uso de **macros**.

Las macros pueden parecerse a las funciones, debido a que hacen una distinción entra la coincidencia exacta de lo que se va a buscar (lo que se podría ver como el nombre de una función), y parámetros que *recibe* dicha *función*. Para entender cómo funcionan las macros desde cero hacer esta comparación está bien, sin embargo no hay que olvidar que la directiva hace una sustitución textual antes de que comience la compilación. Como se explicó antes, el pre-procesador reemplaza las macros por el texto que éstas definen.

El uso de macros debe de hacerse con cuidado. Por ejemplo, no tiene un control de tipo de datos, a pesar de que *recibe argumentos*, los utiliza para la expansión en tiempo de preprocesador y por lo tanto no verifican tipos, lo que significa que puedes pasar cualquier cosa como argumento, lo que conducirá a errores; no admite sobrecargas y si se definen macros muy complejas es propensa la aparición de errores sutiles e inesperados al no seguir reglas de ámbitos ni de tipos, dificultando la depuración del programa, ya que el código sustituido puede ser difícil de rastrear.

Ejemplos de dichos errores que pueden aparecer se pueden apreciar en los Lst. 3.5, donde se define una macro que sustituye su utilización por el valor numérico del cuadrado de un número, pero que al hacer la sustitución directa, comete errores aritméticos en el cálculo; y en el Lst. 3.6, donde se define una macro para hallar el máximo entre dos números que en principio está bien definida, pero que generará un error al momento de su utilización.

```
1 #define SQR(x) x * x
2 cout << SQR(1 + 2); // Esperado: 3 * 3 = 9
3 // Resultado incorrecto: 1 + 2 * 1 + 2 = 5
```

Listado 3.5: Definición de una macro para calcular el cuadrado de un número con una definición mal planteada.

```
1 #define MAX(a, b) ((a) > (b) ? (a) : (b))
2 cout << MAX(i++, j);
3 /*
4 Incrementa ‘i‘ dos veces, comportamiento inesperado:
5 ((i++) > (j) ? (i++) : (j))
```

```
*/
```

Listado 3.6: Definición de una macro para calcular el máximo entre dos números, que fallará en ciertos casos en su ejecución.

Una forma correcta de aprovechar entonces las macros, es limitándolas a un entorno controlado cuyo comportamiento no tiene margen de variación. Ejemplos de usos tanto recomendados como reales de implementación dentro de la programación competitiva se muestran en el Lst. 3.7.

```
// Sintaxis: #define MACRO(params) definicion_de_la_macro
#define all(v) v.begin(),v.end()
#define rall(v) v.rbegin(),v.rend()
#define sz(a) (int)(a.size())
#define fori(i,a,n) for(int i = a; i < n; i++)
#define endl '\n'
```

Listado 3.7: Macros de uso recomendado para una plantilla.

Para evitar este tipo de errores, `define` se debe utilizar únicamente cuando no haya otra opción de lo que se quiera sustituir. Por ejemplo, se puede utilizar `define` para sustituir lo explicado en la Sec. 3.1, para definir constantes (Sec. 3.3) o para definir el comportamiento de ciertas funciones (las funciones generan un código más limpio y fácil de entender) (Lst. 3.5 y 3.6), sin embargo existen otros mecanismos más seguros y controlados para realizar todas estas cosas y por tanto deben ser utilizados sobre `define`. Donde no se puede sustituir su uso es en la creación de atajos que se aprovechan del mecanismo de sustitución textual (Lst. 3.3 y 3.7), dichos usos son los recomendados para la programación competitiva.

`define` también se ha llegado a utilizar para definir el valor de variables constantes inmutables, esta práctica es muy común al trabajar en lenguaje C, sin embargo en C++ es más seguro hacer dicha tarea con el modificador `const`, como se verá en la Sec. 3.3.

3.3 EL MODIFICADOR 'CONST'

Muy a menudo se utilizan en las soluciones valores numéricos que son los mismos sin importar el problema que se esté resolviendo, y que por tanto nos es conveniente tenerlos en nuestra plantilla. Tal es el ejemplo del valor máximo que puede tomar una variable de tipo entero y que es conveniente definirla como *infinito* en el tratamiento de ciertos problemas; éste mismo valor, pero para variable de tipo `long long`; el entorno épsilon con el cuál se puede realizar la comparación de números flotantes; o el valor por el cuál se solicita aplicar módulo a la solución de problemas cuya solución toma números demasiado grandes para ser presentados tal cual[1].

Para guardar estos valores de manera global (respecto a cualquier solución desarrollada) existe el mecanismo de utilizar variables globales y **constantes**, cuyo valor una vez definido no podrá ser mutado en tiempo de compilación, ni de ejecución.

Los identificadores de dichas variables por buenas prácticas y por convención se escriben siempre en mayúsculas.

1 Dicho valor es $10^9 + 7$, no seleccionado al azar, sino por ser un número primo muy grande, y por tanto su módulo tiende también a ser un número único identificable y comparable por los jueces en línea a la respuesta esperada.

¿Y qué es lo que vuelve a estas variables especiales? Su declaración como `const` es lo que le indica al programa que no pueden ser mutadas. La sintaxis para la declaración de constantes es la siguiente:

```
const tipo_dato IDENTIFICADOR = valor;
```

En el Lst. 3.8 se muestra la declaración directa de constantes, y los ejemplos más recurrentes en problemas de tipo programación competitiva.

```
1 // Sintaxis: const tipo_dato IDENTIFICADOR = valor;
2 const int MOD = (int)1e9+7;
3 const int INF = INT_MAX;
4 const long long LLINF = LLONG_MAX;
5 const double EPS = DBL_EPSILON;
```

Listado 3.8: Declaración de variables constantes más utilizadas en plantillas para problemas de tipo programación competitiva.

3.4 CONFIGURACIÓN PARA ENTRADA Y SALIDA

Quizá la primera complicación con la que se encuentra quien se esté iniciando en la programación competitiva será el de tratar con la entrada y salida de información que el programa debe de procesar.

Ya escogí un problema para resolver, leí cuidadosamente los detalles e implementé una solución que, según mi criterio, pareciera ser correcta, ¿cómo puedo probarla?, ¿quién me va a decir que, en efecto, lo que hice es correcto? El mecanismo

mediante el cuál se lleva a cabo ésto es a partir de los jueces en línea.

Los problemas que vayas a resolver están planteados por las distintas plataformas dedicadas a ello (Véase Sec. 4.1), y de la misma forma que con el problema se plantea la solución a los problemas.

La solución codificada (el archivo código fuente) debe de *subirse* o *mandarse* en un apartado específico que existe en cada **problema** (o **contest**, en su defecto). Dicho apartado varía según la interfaz y el sitio que se esté consultando, algunas veces se puede copiar y pegar todo el código, otras sin embargo se solicita forzosamente seleccionar un archivo, pero en cualquier caso se hace lo mismo: someter tu programa a un proceso de pruebas exhaustivo.

A la acción de mandar una solución a un juez en línea para que te dé un veredicto (Véase Sec. 6.1) comúnmente también se le llama hacer *submit*[2]. Una vez que se hace submit de un código, a éste le se harán varias pruebas (casos de prueba) llamados *test cases*, y en cada test case se ejecutará la solución con inputs conocidos (y rutinarios) para el juez en línea, donde se conoce la respuesta correcta que se espera produzca la solución.

Por lo tanto, la manera en que se presentan los resultados de la solución (output) importan, deben ser reconocibles por el juez en línea para que pueda ser reconocida como exitosa.

La indicación es utilizar la entrada y salida estándar: *stdin* y *stdout*. Ésto describe la manera en que presentar el flujo

2 Literal traducción del inglés: enviar.

de entrada y salida de información; son los canales por los cuales un programa recibe los datos de entrada y envía los resultados.

La entrada suele proporcionarse en un archivo, o directamente como flujo de datos, mientras que la salida suele presentarse por medio de la consola, o a un archivo redirigido. En alto nivel: el teclado y la impresión en pantalla. Muy probablemente los programas que hayas desarrollado hasta ahora utilizaban entrada y salida estándar[3], es posible, de hecho, ejecutar el programa e ir escribiendo en la terminal y por el teclado toda la información que necesita (aunque por obvias razones hacer ésto al momento de probar una solución no es recomendado), y se mostrará en la pantalla de la terminal el resultado. Siempre que la lógica sea correcta, hacer ésto funcionará para un juez en línea.

La eficiencia con que se maneja la lectura y escritura de datos es bastante relevante cuando se debe procesar gran cantidad de información, como es en la mayoría de casos cuando de programación competitiva se trata, por lo tanto es crucial establecer métodos optimizados.

En el caso de C++ se utiliza `cin` para la entrada, y `cout` para la salida, sin embargo es relevante recordar que C++ inicialmente fue diseñado como una extensión de C, por lo que C++ es un superset de C y los elementos existentes en C también existen para C++, incluyendo la entrada y salida. Entonces además de la IO de C++, también es posible utilizar las funciones de C `scanf` y `printf`. Ambos pares de funciones

3 Para C++ las sentencias de IO son *cin* y *cout*, sin olvidar que es un superset de C donde existe *scanf* y *printf*; para Python *input()* y *print*, para Java un objeto de tipo *Scanner* con *System.in*, y *System.out.print*.

están sincronizadas entre sí, de manera que nativamente puede utilizarse cualquiera de las cuatro funciones.

Ésto es bastante cómodo al momento de desarrollar en C++, sin embargo en el ámbito de la programación competitiva resulta ser una desventaja, ya que esta atadura alenta terriblemente la entrada y salida de información que en la mayoría de lo problemas es de por sí suficiente grande.

Para resolver este problema se puede romper la sincronización entre ambos pares de funciones a costa de escoger forzosamente una de ellas mediante la instrucción

```
ios_base::sync_with_stdio(0);
```

Ésto es, si decides trabajar con `cin` y `cout`, no existe la posiblidad de utilizar `scanf` y `printf`. Ésto por sí mismo no parece un gran problema, sin embargo las funciones de C son en general más rápidas que las de C++, aún así, las de C++ son mucho más rápidas de utilizar al escribir código, y mucho más fáciles de usar, ya que no entra en detalles del tipado de datos. Estas formas de optimización es conveniente encapsularlas en una función a la que llamar en cada solución e, inevitablemente, añadirla a la plantilla. La función en específico se muestra en el Lst. 3.9.

¿Por qué es más rápido utilizar `cin` y `cout`? La razón es que nativamente `cin` está atada (*tied*) a `cout`. Lo que significa que antes de cada entrada de `cin` se vacía el buffer de `cout` (*flush*). Hacer `cin.tie(0)` rompe dicha relación, evitando que se limpie el buffer cada vez y, por tanto, optimizando la entrada. Similar a lo anterior, hacer `cout.tie(0)` rompe su relación con `cin` y cualquier otro flujo de entrada.

```
1 void fastIO() {
```

```
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
}
```

Listado 3.9: Función de optimización de la entrada / salida en C++.

Es precisamente la acción de que se limpie el buffer la responsable del último ejemplo propuesto en el Lst. 3.7 sobre el uso de `define`. Pues cada vez que se escribe un salto de línea con `endl` se limpia el buffer que, de nuevo, consume tiempo en su uso repetido para la salida. Es precisamente esa la diferencia entre utilizar `endl` y su antecesor `'\n'`, también es la razón por la cuál es preferida la segunda opción. Aún así es más rápido de escribir `endl`, y también se vuelve más entendible el código C++ y por eso es propuesto el atajo.

En primer contacto es bueno conocer la forma de hacer *fastIO*, sin embargo se vuelve un elemento crucial para cierto tipo de problemas donde la entrada y salida es considerablemente grande (mayor al promedio). Tal es el caso del problema GYM - 105200 B (CF)[5], donde una solución correcta que implementa el *fastIO* no es capaz de lograr el *Accepted* sin él, pues da error de *Time Limit Exceed* (Véase Sec. 6.1) en el test case número tres. Ambos submits se pueden ver en la Fig. 3.

submissions

#	When	Who	Problem	Lang	Verdict	Time	Memory
294508202			E - Earthquake	C++20 (GCC 13-64)	Time limit exceeded on test 3	1500 ms	26700 KB
294508157			E - Earthquake	C++20 (GCC 13-64)	Accepted	468 ms	34000 KB

Figura 3: Submit de una misma solución con y sin *fastIO*.

Como se habló antes, las entradas muchas veces vendrán de archivos, y aún cuando se trate de las muestras triviales a la solución del problema resulta ineficiente simplemente copiar el input completo y pegarlo en la terminal después de ejecutar, para ello existen mecanismos más sofisticados.

Uno de ellos es leer la entrada de un archivo que será llamado de forma genérica `input.txt`, y escribir el resultado en otro llamado `output.txt`. La manera de hacerlo es dirigiendo el flujo de IO abriendo el archivo de input para lectura a partir de stdin, y el de output como (sobre)escritura con stdout. Adicionalmente, si se acomoda el programador para trabajar así puede dejar el código sin necesidad de modificarlo al hacer sumbit mediante un `#ifndef`, como se muestra en el Lst. 3.10.

```
1 void setIO() {
2     #ifndef ONLINE_JUDGE
3         freopen("input.txt", "r", stdin);
4         freopen("output.txt", "w", stdout);
5     #endif
6 }
```

Listado 3.10: Definiendo la entrada y salida del programa en archivos.

Otra manera de trabajar de una manera más cómoda los inputs es con banderas de compilación en terminal. Para ésto hay que entender primero cuál es la sintaxis para compilar y ejecutar un archivo con extensión `.cpp`, para los ejemplos partiremos del supuesto que nuestro programa se encuentra en el archivo `solucion.cpp`, y que todas las instrucciones se ejecutan desde el mismo directorio en el que se encuentra este archivo.

La sintaxis básica para compilar el programa mediante `gcc` es con la siguiente sintaxis:

```
g++ codigo_fuente.cpp -o ejecutable.exe
```

Donde se da la indicación que se hace una llamada a g++ (compilador), para que compile el archivo con extensión `.cpp`, y a través de la bandera `-o` se indica que el archivo inmediato a la bandera será el nombre del archivo ejecutable resultante de la compilación. La ejecución se hace mediante la sintaxis:

```
./ejecutable.exe
```

Por convención se suele utilizar el mismo nombre del código fuente, pero con la extensión propia[4] para el archivo ejecutable. De manera que la sintaxis mínima en terminal para compilar y ejecutar un programa sería la siguiente:

```
g++ solucion.cpp -o solucion.exe
./solucion.exe
```

De manera adicional, existen ciertas banderas que pueden resultar útiles al momento de compilar. La bandera `-Wall` (warning all) hace que se muestren todos los warnings en pantalla; utilizar una versión específica del compilador con `g++-N`, donde `N` es una versión del compilador g++, por ejemplo la importación de la STL es una característica del compilador 13.

4 En sistema operativo Windows la extensión debe ser `.exe`, mientras que en macOS y Linux no es necesaria una extensión, puede especificarse únicamente como `-o salida`, aunque también se suele utilizar la extensión `.out`.

De manera similar a como se hizo en el Lst. 3.10, e independientemente del compilador (son mecanismos relativos al entorno en línea de comandos) se le puede dar la indicación al momento de la ejecución de que se tome el input a partir de un archivo, y se escriba en otro. Ésto se hace mediante los operadores de redirección < y >. Se puede redirigir el contenido de un archivo `input.txt` como entrada estándar del programa mediante <, y redirigir la salida estándar el programa al archivo `output.txt` mediante >.

Con estas notas adicionales, el ejemplo de compilación y ejecución en terminal resulta:

```
g++-13 -Wall -o solucion.exe solucion.cpp
./solucion.exe < input.txt > output.txt
```

Es importante que el programador competitivo conozca estas herramientas y esté acostumbrado a trabajar con ellas debido a que las competiciones difieren entre contest y contest dependiendo de quién las organice, el entorno donde se encuentre, la máquina en la que se esté realizando... todas estas características, y por tanto también el entorno de trabajo se vuelven volátiles. Ésto puede demostrar ser una desventaja para los competidores que confían demasiado en entornos de desarrollo integrados (IDEs) caprichosos y de lujo para depurar. ¡Podría ser una buena idea practicar a programar con únicamente un editor de textos y un compilador!

En la Fig. 4 podemos apreciar un fragmento de un problema en donde como dato de entrada (input) se brinda un número `t`, cuyo valor puede estar tomar valores desde 1 hasta 10,000. Esto implica que para que nuestro algoritmo sea calificado como correcto debe ofrecer una respuesta válida en 10,000

Input
Each test contains multiple test cases. The first line contains the number of test cases t ($1 \le t \le 10\,000$).

Each test case consists of a single integer n ($1 \le n \le 10^9$)

Figura 4: Ejemplo de un *input* convencional sacado del problema 1788B.[5]

ocasiones (pese a que el intervalo pueda ir de 1 hasta 10 000 en la gran mayoría de casos se probará bajo el límite superior). Posterior a leer el número de *test cases*, debemos además de leer para cada uno de ellos los datos a procesar.

Ya que este formato es muy popular, podemos utilizar la estructura de código mostrado en el Lst. 3.11 como plantilla, ésta nos será útil siempre que nos enfrentemos a un problema donde tengamos un número de casos de prueba mayor a uno.

```
1  void solve() {
2      // Implementa tu solución en un entorno limpio
3  }
4
5  int main() {
6      int testCases;
7      cin >> testCases;
8
9      while( testCases-- ) { // while( testCases > 0 ) {
       testCases -= 1 ... }
10         // Leer datos del problema
11         solve();
12     }
13
14     return 0;
```

```
15 }
```

Listado 3.11: Plantilla para varios casos de prueba.

El código presentado en el Lst. 3.11 comienza en la función *main*. Dentro se declara una variable de tipo entero que almacenará el número de test cases (siempre será el primer elemento a leer de un problema del tipo descrito). Posteriormente procedemos a leer el valor con ayuda de la palabra reservada *cin*. Una vez que en memoria contamos con nuestro número de casos de prueba, utilizamos un bucle de repetición *while* cuya condición de ejecución consistirá en validar si el número de *test cases* es mayor a cero, decrementando una unidad a la variable cada vez.

Dentro de nuestro ciclo *while* el bloque de instrucciones a ejecutar consiste en llamar a una función cuyo nombre y definición construiremos de acuerdo a nuestras necesidades. Es aquí donde se ejecutará el algoritmo planteado para solucionar el problema.

Es importante mencionar que el grado de personalización de un plantilla puede llegar a ser bastante alto y dependerá enteramente de los gustos y necesidades de cada programador.

Bien pues, después de enlistar y explicar brevemente los elementos básicos para la creación de una plantilla para programación competitiva, no queda más que juntarlos todos para crear la plantilla propuesta de este documento, misma que se puede ver en el Lst. 3.12.

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

```
3  typedef long long ll;
4  typedef long double ld;
5  typedef pair<int,int> ii;
6  typedef pair<ll,ll> pll;
7  typedef vector<int> vi;
8  typedef vector<ll> vll;
9  typedef vector<ii> vii;
10 typedef vector<vi> vvi;
11 typedef vector<vii> vvii;
12 #define fi first
13 #define se second
14 #define pb push_back
15 #define all(v) v.begin(),v.end()
16 #define rall(v) v.rbegin(),v.rend()
17 #define sz(a) (int)(a.size())
18 #define fori(i,a,n) for(int i = a; i < n; i++)
19 #define endl '\n'
20 const int MOD = 1e9+7;
21 const int INF = INT_MAX;
22 const long long LLINF = LLONG_MAX;
23 const double EPS = DBL_EPSILON;
24 void fastIO() {
25     ios_base::sync_with_stdio(0);
26     cin.tie(0);
27     cout.tie(0);
28 }
29 void setIO() {
30     #ifndef ONLINE_JUDGE
31         freopen("input.txt", "r", stdin);
32         freopen("output.txt", "w", stdout);
```

```
33      #endif
34  }
35
36  void solve() {
37
38  }
39
40  int main() {
41      fastIO();
42      setIO();
43      int testCases;
44      cin >> testCases;
45      while( testCases-- ) {
46
47          solve();
48      }
49      return 0;
50  }
```

Listado 3.12: Plantilla completa con todos los elementos mínimos para la resolución de problemas tipo programación competitiva en C++.

El uso de una plantilla propia es una herramienta muy poderosa para la resolución efectiva y veloz de problemas en la programación competitiva, aún así, la construcción de ésta no se trata de únicamente copiar lo que se ve en otras personas para sí, sino de familiarizarse con ella. Una plantilla se vuelve una compañera, y la cercanía con ella incrementa conforme más problemas se resuelvan a su lado.

Es por ésto que el programador debe de saber exactamente qué hace cada elemento de su plantilla, entender qué hacen las instrucciones que utiliza en ella permitirá su modificación y evolución conforme se vaya conociendo, y suprimirá posibles errores de depuración de código cuando las soluciones propuestas no sean aceptadas por un juez.

Nosotros aquí damos las herramientas básicas que se pueden utilizar para el programador que ya tenga poca (o mucha) experiencia en la programación competitiva, así como para que aquel programador que ya conozca el lenguaje de trabajo (propuesto C++) a profundidad por causa de la experiencia pueda brincar a trabajar con una plantilla y pueda agilizar su aprendizaje a partir de la experiencia de los demás. Sin embargo, el programador que es novato, que no conoce el lenguaje y que no ha tenido experiencia no debe sentirse presionado ni abrumado por dominar al instante todo este conocimiento, lo más importante a fin de cuentas es lograr algo funcional para sí mismo, no utilizar los más elementos encontrados de forma aleatoria. Nuestro consejo para aquellas personas es que comiencen poco a poco, con el pasar de los problemas el atleta identificará las necesidades que la plantilla soluciona, y con el paso del tiempo inevitablemente aprehenderá el conocimiento necesario.

4

COMPETICIONES O 'CONTESTS'

Un **contest** se define como:[1]

> Competencia estructurada en la que los participantes deben resolver una serie de problemas (problemset) dentro de un tiempo limitado, utilizando sus habilidades en programación y pensamiento lógico.

Como toda disciplina, se puede identificar un ciclo sencillo de aprendizaje: Aprender y practicar. (Cumpliendo este ciclo puedes dominar presumiblemente cualquier disciplina)[2]

No obstante para la programación competitiva se tiene un ciclo más específico: Estudia, practica, vuelve a practicar (practica de nuevo y aún más) y, finalmente, **compite**. Somos competidores, naturalmente nuestro objetivo final es competir entre los mejores.

1 El término de competencia, concurso y contest se usa indistintamente a lo largo de este libro para referirse a lo mismo.

2 *Les funciona para los atletas, les funciona a los músicos, te funcionará a ti*.[1]

Los contests (en general, las competencias), por ende, son donde nos desenvolvemos, son la última parte del bello ciclo de nuestro entrenamiento, allí es donde demostramos de lo que somos capaces, donde aplicamos todo lo aprendido en nuestros días de arduo esfuerzo practicando.

Existen competencias abiertas para todo público, usualmente en línea en plataformas como las que se mencionarán posteriormente, donde el nivel de los problemas es variable, puede ser una competencia donde se espere una dificultad media, o puede ser uno donde solo los mejores programadores son capaces de sacar al menos un problema dentro del repertorio ofrecido.

Existen también las competencias mundiales, donde el número de participantes se distribuye en equipos, los cuales son sometidos a un proceso de concurso largo que va desde etapas regionales hasta las más selectas para así decidir que equipo será el que pase a las finales.

Como club, tenemos una meta clara y general, poder entrenar atletas que representen a nuestra universidad en la ICPC.

La ICPC (*International Collegiate Programming Contest*) es el concurso de programación competitiva por excelencia; el más reconocido y cuyo formato es utilizado por muchos otros contests alrededor del mundo.

Es un concurso anual a nivel Universitario[3], el cuál cuenta con un proceso de selección de equipos conformados por tres integrantes, los cuáles deben destacar y quedar en una buena

3 Cada equipo representa a su Universidad, por lo que es usual que cada Universidad someta a todos sus equipos a un proceso interno de entrenamiento y de selección para así mandar a sus equipos más fuertes a competir.

posición en todas las etapas que conforman su proceso. Se destacan las etapas regionales (toman lugar usualmente entre los meses de Octubre y Diciembre), los ganadores pasaran a la final mundial (*World Finals*) y competirán el año siguiente (entre Abril y Julio).

La competencia dura 5 horas, cada equipo cuenta con una sola computadora, por lo que la organización y el trabajo en equipo es primordial. Las soluciones son aceptadas (AC) si y solo si cumplen con todos los casos de prueba de manera eficiente (Véase Sec. 5), esto da al equipo +1 punto.

Un equipo tiene una penalización (usualmente +20 minutos al tiempo final) por cada envío erróneo (no Accepted) de un problema.

Los equipos son **rankeados** por el número de problemas resueltos, si se empatan con otro, irá primero el que menos tiempo de penalización tenga, y si empata de nuevo con otro equipo, irá quien haya subido antes su último envió exitoso.

Más allá de la universidad, existen una cantidad extensa de contests abiertos a todos los niveles, la mayoría en línea, en donde los atletas pueden desenvolverse, empresas como Facebook, Google, IEEE y Yandex organizan concursos, tales como Facebook Hacer Cup, Google Code Jam, IEEEXtreme, entre otros. Codeforces, por ejemplo, es una de las plataformas de programación competitiva más activas (Véase Sec. 7)[4], organizando competencias casi semanalmente y dividiendo a los participantes en divisiones según su experiencia.

4 También es la plataforma predilecta del CPCFI.

Aunque la última etapa del proceso sea competir entre nosotros, no debemos olvidar que durante la parte previa, somos un equipo, como club y como programadores que con buena voluntad, estamos dispuestos a ayudarnos entre nosotros.

Nuestro objetivo es llegar al ICPC, pero antes hay mucho camino por recorrer.

Como se dijo en la introducción de este libro y en muchos otros libros de programación competitiva:

> La programación competitiva es un deporte mental
> [...]

Y tal cuál un atleta, los programadores competitivos deben entrenar regularmente y mantenerse en forma, practicar es pues una acción inherente a nuestra preparación.

4.1 DÓNDE PRACTICAR

Para convertirse en competitivo y desarrollarse *bien* en competiciones de programación, debes de ser capaz de clasificar con confianza frecuentemente problemas que ya se resuelto antes, y de los cuáles se tiene la seguridad que se es capaz de resolver de nuevo (y de una manera rápida).

Aprender programación competitiva requiere una enorme cantidad de trabajo, y es importante mencionar que la cantidad de problemas resueltos no es tan importante como la calidad de los problemas. Es tentador seleccionar aquellos problemas que se ven bonitos y fáciles de resolver, y saltarse los problemas que se ven difíficles y tediosos. Sea cual sea el

caso, la forma de mejorar las habilidades propias es enfocarse en el segundo tipo de problemas.

Otra observación importante es que la mayoría de contests pueden ser resueltos utilizando algoritmos sencillos y cortos, la parte difícil sin embargo es inventar dichos algoritmos. La programación competitiva no es acerca de memorizar algoritmos complejos y oscuros, sino de aprender habilidades de resolución de problemas y maneras de encontrar aproximaciones a problemas difíciles utilizando herramientas simples.

Para mantenernos en forma, utilizamos los siguientes sitios de confianza, como atletas del CPCFI creemos firmemente que el éxito es el resultado del continuo esfuerzo en uno mismo.

4.1.1 *Codeforces*

Codeforces es el sitio predilecto donde se imparten las lecciones y claramente también el curso propedéutico. Contiene una miríada de ejercicios disponibles para resolver, y continuamente se suman al archivo más y más. Ofrece un ranking calculado a partir de contests llamados *Divs*, por supuesto que su propio juez virtual y proporciona estadísticas de actividad y rendimiento según se resuelvan problemas.

Al ser Codeforces el sitio principal de trabajo, se puede encontrar una descripción más detallada en la Sec. 7.

4.1.2 *CSES*

CSES Problem Set provee una colección de problemas que pueden ser utilizados para practicar programación competitiva.

Los problemas han sido organizados por orden de dificultad y por tema.

CSES es una herramienta muy poderosa para los estudiantes de programación competitiva, y en especial para los que se están iniciando en ella, debido a que permite seguir la resolución lineal de los problemas para aprender desde los algoritmos y conceptos más básicos, hasta los más avanzados. El juez que tiene no utiliza tantos test cases como en Codeforces, por ejemplo, sino que engloba casos críticos en donde puede haber un punto de fallo para la solución. Tras cada submission también es totalmente transparente con los veredictos y el contenido de los test cases, una vez que encuentra un veredicto distinto de AC para cualquier test case, no se detiene ahí como lo harían el resto de plataformas, sino que ejecuta todos los demás para poder mapear el conjunto de casos en los que la solución falla.

4.1.3 *Leetcode*

Leetcode es un sitio bastante reconocido por ofrecer problemas de tipo programación competitiva que han aparecido en entrevistas técnicas laborales, inclusive se pueden consultar las preguntas que se han hecho de las empresas de tecnología más grandes en específico. El cuerpo de problemas que contiene es vasto, está bien organizado y categorizado por temas y dificultad.

Leetcode es un sitio cuya visión no se limita únicamente a la programación competitiva, su interfaz es amigable con el usuario y definitivamente vale la pena visitarlo para precticar.

La metodología del CPCFI para el aprendizaje de programación competitiva se centra en una selección de problemas en Codeforces, con algunos problemas de CSES complementarios. Bajo la instrucción y selección de problemas adecuada, es posible practicar los conceptos más básicos en CSES para practicarlos en problemas más complejos dentro de Codeforces. De manera complementaria Leetcode es un buen complemento.

Aún así, existen muchísimos sitios que proveen problemas de tipo programación competitiva, y según la bibliografía que se consulte se mostrarán unos sitios u otros.

USACO Guide, por ejemplo, es un sitio excelente para el atleta de programación competitiva, provee material vasto y variado sobre temas afines, explicación e implementación de algoritmos y estructuras de datos, listas de ejercicios relacionados a un tema que engloba otros sitios de programación competitiva y claro, su propio conjunto de problemas clasificado por dificultad.

La siguiente es una lista de otros sitios populares con problemas de programación competitiva de distintas dificultades:

- UVa Online Judge. (Universidad de Valladolid, España)
- ACM-ICPC Live Archive.
- CS Academy.
- USACO. (*USA Computing Olympiad* training program)
- AtCoder.
- CodeChef.

5

ANATOMÍA DE LOS PROBLEMAS EN CONTESTS

Cuando comenzamos en la Programación Competitiva tenemos que acostumbrarnos a un formato en el cual son presentados la mayoría de los problemas independientemente de la competencia o el jurado en línea en el cual nos encontremos participando. Dicha estructura se describe a continuación.

5.1 DESCRIPCIÓN

La descripción de un problema proporciona el contexto y los objetivos. Explica qué representa el problema y cuál es la tarea que se espera resolver, a menudo mediante un escenario que ayuda a entender la situación. En esta sección se detallan:

- El objetivo final del problema (qué hay que calcular, construir o validar).
- La lógica o reglas básicas que deben seguirse.
- Ejemplos o detalles adicionales que aclaran los requisitos.

Ejemplo: "Dado un arreglo de enteros, encuentra el subarreglo contiguo que tenga la suma más alta."

5.2 RESTRICCIONES (CONSTRAINTS)

Las restricciones definen los límites del problema y ayudan a determinar la eficiencia necesaria del algoritmo. Especifican el rango y los límites de los datos de entrada, como el tamaño máximo de los números o la cantidad de elementos. Estas restricciones permiten a los competidores prever la complejidad óptima para que su solución sea lo suficientemente rápida y eficiente.

Ejemplo: $1 < n < 10^5$

5.3 ENTRADA (INPUT)

La entrada describe los datos que se proporcionarán al programa. Define el formato y el tipo de información que el programa recibirá para resolver el problema, tales como:

- El número de casos de prueba o cantidad de datos.
- Los tipos de datos. (e.g. enteros, cadenas)
- La estructura de la entrada. (líneas, columnas, matrices, etc.)

Ejemplo: La primera línea contiene un entero `n` (el número de elementos en el arreglo). La segunda línea contiene `n` enteros que representan los elementos del arreglo."

Cuando comenzamos en la programación competitiva uno de los contratiempos con los que más frecuentemente nos encontramos es el cómo leer la entrada que nos brinda un problema, ya que en la mayoría de las ocasiones es un mecanismo al que no estamos acostumbrados, a continuación se enlista una manera simple que nos ayudará a cubrir este punto.

Dado que la entrada de un programa usualmente consiste en números (*int*) y/o cadenas (*strings*) que están separadas por espacios y saltos de línea, esta puede ser leída con las siguientes instrucciones descritas en el Lst. 5.1.

```
1 // Declaración de variables
2 int x, y;
3 string p;
4
5 // Lectura
6 cin >> x >> y >> p;
```

Listado 5.1: Declaración y lectura de variables enteras y cadenas en C++.

Al llevar a cabo esta secuencia nosotros podemos leer cualquier disposición de la entrada, siempre y cuando haya al menos un espacio o un salto de línea entre cada una de las variables. Es decir, el problema nos podría haber descrito la entrada de la siguiente manera:

Input: Se brindan 3 elementos, los dos primeros son dos enteros `1 <x, y <10,000`, mientras que el tercero es una `string p` donde la longitud de `p` es menor a 100 (esta input puede ser leída perfectamente por el código proporcionado anteriormen-

te).

La entrada podría lucir del siguiente par de maneras y funcionaría para ambas:

```
10 2000
CPCFI

10 2000 CPCFI
```

En algunos problemas puede llegar a ser importante optimizar nuestro algoritmo hasta el grado en el que incluso la manera en que leemos y escribimos la entrada y la salida respectivamente se puede volver crítica, es por esta razón que se recomienda utilizar las instrucciones mostradas en el Lst. 5.2.

```
1 ios::sync_with_stdio(0);
2 cin.tie(0);
3 cout.tie(0);
```

Listado 5.2: Instrucciones para eficientar la salida y entrada de datos estándar.

Dichas instrucciones (Lst. 5.2) se pueden indicar en el método principal antes de iniciar la lógica del programa, sin embargo, si se desea mantener un aspecto más limpio y claro es recomendable utilizar una función cuya llamada encapsule las instrucciones. El uso de este método se muestra en el Lst. 5.3.

```
1 fastIO() {
```

```
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
}

int main() {
    fastIO();
    // Your solution here
    return 0;
}
```

Listado 5.3: Instrucciones para eficientar la salida y entrada de datos estándar.

La primer línea nos permite desvincular el flujo de entrada/-salida de C++ y de C: `cin` y `cout` en el caso de C++, y `scanf` y `printf` para C. Mientras que la segunda evita la sincronización entre *cin* y *cout*. Ambas líneas en conjunto permiten que el proceso de lectura y escritura sea más rápido, sin embargo, una vez que se hace uso de estos comandos no es posible usar `printf` y `scanf` dado que pueden comportarse de manera inesperada.

Por último, algunos jueces en línea no indican de antemano el número de casos de prueba a los cuales se someterá nuestro algoritmo, es por esta razón que es conveniente presentar una manera de declarar un bucle que nos permita manejar este tipo de entrada, como lo muestra el Lst. 5.4, donde se declara un bucle que lee elementos desde el input uno tras otro hasta que ya no hay más.

```
while( cin >> a ) {
```

```
    // your logic here
}
```

Listado 5.4: Instrucciones para eficientar la salida y entrada de datos estándar.

5.4 SALIDA (OUTPUT)

La salida especifica lo que debe devolver el programa después de procesar la entrada. Indica el formato exacto y los datos que deben imprimirse, a menudo con ejemplos de salida deseada. Esta sección es clave para que el juez automático pueda verificar si el programa produce los resultados correctos en el formato adecuado.

Ejemplo:

Imprime un solo entero que represente la suma máxima de un subarreglo contiguo del arreglo dado.

En la Fig. 5 se muestra el enunciado completo de un problema típico de *Codeforces* donde podemos identificar los diferentes segmentos descritos con anterioridad.

En primer lugar tenemos las restricciones de complejidad tanto temporal como espacial que debe de cumplir nuestra solución (debe entregar una solución en un segundo y debe requerir como máximo 64Mb para su ejecución).

Lo siguiente es la descripción del problema, posteriormente nos es presentado el tipo de *input* que recibiremos, así como

A. Watermelon

time limit per test: 1 second
memory limit per test: 64 megabytes

One hot summer day Pete and his friend Billy decided to buy a watermelon. They chose the biggest and the ripest one, in their opinion. After that the watermelon was weighed, and the scales showed w kilos. They rushed home, dying of thirst, and decided to divide the berry, however they faced a hard problem.

Pete and Billy are great fans of even numbers, that's why they want to divide the watermelon in such a way that each of the two parts weighs even number of kilos, at the same time it is not obligatory that the parts are equal. The boys are extremely tired and want to start their meal as soon as possible, that's why you should help them and find out, if they can divide the watermelon in the way they want. For sure, each of them should get a part of positive weight.

Input
The first (and the only) input line contains integer number w ($1 \leq w \leq 100$) — the weight of the watermelon bought by the boys.

Output
Print `YES`, if the boys can divide the watermelon into two parts, each of them weighing even number of kilos; and `NO` in the opposite case.

Examples

input	Copy
8	
output	**Copy**
YES	

Note
For example, the boys can divide the watermelon into two parts of 2 and 6 kilos respectively (another variant — two parts of 4 and 4 kilos).

Figura 5: Enunciado completo del problema 4A-Watermelon.[5]

los límites del mismo. Esto es necesario para poder calcular la complejidad que debe mantener nuestra solución.

En la sección *output*, se especifica cómo debe de presentarse una respuesta después de que nuestro algoritmo procesó los *inputs*.

Por último, en la mayoría de los problemas se nos presenta un caso o un grupo de casos de ejemplo con los cuales podemos entender de mejor manera cómo se espera que nuestra solución se comporte, es fundamental que cuando leamos estos, pensemos en escenarios que no estén contemplados en ellos, ya que no considerar los casos extremos fácilmente puede hacer que nuestra solución no arroje un resultado incorrecto.

6

LOGRANDO EL 'ACCEPTED'

En cuanto se empiezan a hacer submissions las dificultades de la programación competitiva se hacen evidentes. Algunas veces las soluciones no satisfacen el problema porque se está cometiendo algún error de lógica, el problema se resolvió mal, el análisis no se hizo completo, las constraints no se verificaron correctamente, no se escogió el método adecuado, la implementación es incorrecta, no se conocen los elementos sintácticos para implementar el algoritmo, o peor aún, ni siquiera se sabe la razón por la que una solución no es correcta. Aunque todas estas dificultades enlistadas se superan conforme se adquiere experiencia, práctica y pericia, no significa que no existan formas de disminuir la incidencia en estos errores cuando se está resolviendo un problema.

La siguiente es una compilación de información, técnicas y recomendaciones que se aprenden fácilmente conforme se practican más y más problemas, y que ayudarán a amortiguar estos fracasos expresados como submissions fallidas. Visto desde afuera algunas soluciones pudieran parecer que funcionen por obra de un milagro, sin embargo esto no es así, una

solución difícil podrá ser realizada por cualquier programador dedicado con la suficiente práctica y habilidad.

6.1 TIPOS DE VEREDICTOS

El primer paso en resolver un error es comprender cuál es ese error, Como se ha mencionado antes, son los jueces en línea los que tienen la responsabilidad de evaluar nuestras soluciones y decidir nuestra suerte, al hacerlo nos responderán con veredictos *estándar* bien conocidos entre la comunidad, conocer estos veredictos será nuestra (generalmente) única pista para saber ¿por qué la solución no fue satisfactoria? Y con ello saber también qué elemento o elementos de la solución se deben de mejorar para un submit exitoso.

Accepted (AC)
Es la respuesta que todos esperamos obtener, significa que la solución proporcionada fue capaz de resolver el problema planteado. Para lograr el accepted el output de cada uno de los test case debe ser probado como válido. Una vez que se logra un accepted, se debe pasar al siguiente problema.

Wrong Answer (WA)
Aparece cuando una submission no tuvo ningún error de los mostrados adelante, el código se ejecutó de inicio a fin por los diferentes test cases, pero en alguno se encontró que el output resultante no es una solución válida al problema, y no es lo que esperaba el juez.

Después del AC es uno de los veredictos más recurrentes, muchas veces se tiene la noción en el propio código de qué

es lo que puede estar fallando, o a qué puntos de quiebre y debilidades es susceptible, si tienes un WA es buena idea empezar por ahí.

Usualmente cuando se está practicando, depende de la plataforma se puede ver el test case donde el algoritmo falla, sin embargo hay plataformas donde, así como en las competencias ésto no es una opción, y toca deducir el error de primera mano. Cuando lo anterior ocurre lo primero suele ser revisar la lógica de la solución y el funcionamiento del programa, que haga lo que tenga que hacer. Si el error del programa no se encuentra y todo parece estar correcto se pueden buscar puntos de quiebre al plantear test cases en valores críticos (si no se hizo previo a la submission) (Véase Sec. 6.2.

WA puede llegar a ser el veredicto más frustrante debido a que, si lo anterior falla y no es capaz de lograr identificar el problema, las pruebas realizadas deben ser más exhaustivas.

Time Limit Exceed (TLE)

Aparece cuando el algoritmo, para uno de los test case, se pasa en ejecución del tiempo límite establecido en las constraints (Véase Sec. 5.2), usualmente un segundo.

Obtener este veredicto no significa necesariamente que el algoritmo proporcione una solución incorrecta, es también muy común que el algoritmo llegue a una solución exitosa, solo que no lo hace en el tiempo permitido.

El TLE es muy común y aparece por diversas razones que en general se resumen en un mal análisis de complejidad. Algunos ejemplos de casos en los que aparece TLE son los siguientes:

- Se necesita optimizar una parte de la solución que ya funciona y está por arriba del límite por *poco*, e.g. aplicar búsqueda binaria en lugar de búsqueda lineal representa optimizar una complejidad $O(n)$ a $O(log_2 n)$ que en algunos casos puede ser suficiente para pasar.

- Se tendrá que replantear la solución completa por causa de haber seleccionado un algoritmo o método inadecuado, e.g. calcular el enésimo término de la serie de Fibonacci de manera recursiva puede ascender a una complejidad $O(2^n)$, mientras que con programación dinámica se puede reducir a $O(n)$.

- Se optó por el brute force. Implementar soluciones por fuerza bruta no es malo en sí, pero debe de hacerse con estricto cuidado en el análisis previo para verificar que, según las constraints, la complejidad de fuerza bruta puede pasar. Muchos problemas adhoc que aparecen en competiciones no avanzadas pueden realizarse por fuerza bruta, hacerlos así para terminar con ellos rápido es probablemente una buena idea.

- No se optimizó la entrada y salida de información. Como se vio en la Sec. 3.4, hay problemas con límites muy grandes que a su vez exigen una entrada y salida muy grandes (e.g. problemas de grafos o de grids) que pueden provocar un cuello de botella para la solución.

- Errores en la lógica del programa. Si existen otros errores en el código al hacer sumbit o correr casos de prueba previos al submit, puede terminar en TLE si. e.g. existe un bucle infinito.

Memory Limit Exceed (MLE)

Similar a TLE, MLE aparece cuando el algoritmo, para uno de los test case, se pasa en ejecución de la memoria límite establecida en las constraints. Aunque obtener este veredicto es poco común, puede suceder por plantear una ED grande e inadecuada al problema (reflejando una selección de estrategia equivocada) e.g. una programación dinámica de tres dimensiones que puede resolverse con dos; y también un poco más común por la pila de métodos en llamadas recursivas, siendo ésta la razón por la que se recomienda optar por algoritmos iterativos sobre recursivos siempre que sea posible.

Compilation Error (CE)

Obviamente hacer submit a ciegas es malo, a menos que se haya realizado un cambio muy menor en un código ya funcional, entonces y por falta de tiempo se puede considerar como una opción desesperada, sí existen las submissions de último minuto. De otra forma sería ilógico hacer sumbission de códigos que no compilan, en dado caso aparecerá CE.

Runtime Error (RE)

Como su nombre indica, aparece cuando ocurre un error en tiempo de ejecución, ésto ocurre por índices mal definidos (accede a una posición inexistente, *segmentation fault*), si explícitamente el valor de retorno es distinto de cero, a veces por causa de algún overflow, entre muchas otras razones.

Presentation Error (PE)

Muy poco común hoy en día, aunque hay competencias en las que aún se puede dar. Aparece cuando la solución es mos-

trada en formatos distintos de los solicitados, e.g. espacios en blanco o saltos de línea inesperados.

6.2 MAESTRO DEL 'TESTING'

Pensaste que resolviste un problema particular. Identificaste el tipo de problema, diseñaste el algoritmo correspondiente, verificaste que el algoritmo (con las estructuras de datos que utiliza) funcionará en tiempo (y con los límites de memoria) considerando la complejidad temporal y espacial e implementaste el algoritmo, pero tu solución sigue sin conseguir el AC. Entonces deberás de ser capaz de diseñar buenos, completos e inteligentes test cases por tu cuenta.

El ejemplo de input / output que viene en la descripción del problema es, por naturaleza, trivial y no debe de utilizarse como indicador de la correctitud del código. En lugar de eso, y de malgastar submissions acumulando penalty, debes de diseñar test cases *trickys*, astutos, que puedan determinar si tu solución es aceptable, o no.

Hacer éste análisis no es cosa únicamente de contingencia cuando una solución no consigue el AC, ni cosa de hacer testing a una solución desarrollada, es también especialmente útil cuando se está planeando el algoritmo solución al problema. Ésto es, si estás resolviendo el problema (antes de implementar nada) y dudas de una solución propuesta por tu compañero, o del rumbo que se le está dando al flujo mental, pensar en test cases especiales o peligrosos para hacer la prueba de escritorio o estudiar el comportamiento del algoritmo ahorrará mucho tiempo si encuentra que el algoritmo no

es capaz de dar con la solución, o hará que se deposite la confianza en que es una solución robusta la propuesta.

Algunos lineamientos que se pueden tomar como base para diseñar estos test cases propios son los siguientes:

- Evita la comparación manual para test cases largos, ésto no solo es inviable en algunos casos, sino que los humanos somos propensos a errores. Si no se puede realizar la comparación por máquina, busca la redundancia en conclusiones con compañeros lo máximo posible
- Para problemas con múltiple test cases, en una misma ejecución incluye dos copias idénticas de un test case de forma consecutiva. Ambas deben de producir el mismo output para la respuesta conocida. Hacer ésto ayuda a determinar, por ejemplo, si existen inicializaciones olvidadas.
- Incluye corner cases. Los corner cases son test cases en los que se prueba con valores críticos para las constraints dadas. Prueba con los valores mínimos y máximos establecidos que pueden aparecer, mucha veces los test cases más simples posibles son aquellos que pueden encontrar errores en la programación.
- Incluye test cases grandes. Sobre todo el problemas que rondan el límite de capacidad para un tipo de dato entero, donde puede ocurrir overflow en las variables y se deben de utilizar `long long` sobre `int`.

Dependiendo del contest puedes o no obtener puntos parciales por resolver un problema en cierto porcentaje de los

test cases, en ICPC y la mayoría de las competencias no, sino que solo conseguirás los puntos (tú y tu equipo) si logran que su solución pase todos los test cases secretos para ese problema.

Otra técnica bastante poderosa en la verificación de algoritmos son las llamadas pruebas de escritorio. Cuando hay problemas, se puede realizar la ejecución del código línea por línea según está escrito con el test case que no esté funcionando. De esta forma estudiarás si, en efecto, el código está haciendo lo que uno piensa que está haciendo. Es sorprendente la cantidad de ocasiones en la que hacer esta prueba de escritorio saca a la luz errores de lógica y de implementación que vuelve una solución en AC. Este proceso usualmente se realiza mediante un depurador en línea, sin embargo dicha herramienta no estará disponible en una competencia.

6.3 ALGUNOS TIPS

Escribe código más rápido

Si bien las competiciones de programación competitiva no son de *speed typing*, sí es una habilidad bien recompensaba para quien se lo toma en serio.

No son pocos los casos en los que la diferencia entre el puesto `i` y el puesto `i+1` fuera de unos pocos minutos, o casos en los que participantes pierden puntajes importantes por no poder programar una solución brute force a último minuto (en realidad, cualquier solución ya planteada en último minuto). Cuando entre equipos se están resolviendo exactamente los mismos problemas, la competencia está en la habilidad de

programar (producir código conciso y robusto), y la velocidad para escribir.

Si el programador competitivo decide tomar el camino de Codeforces (o cualquier otro sitio con contests) y comience a participar en *Divs* se encontrará con gente que es capaz de solucionar los cinco o seis problemas en tan solo doce minutos... claramente la velocidad para escribir de dichas personas es extraordinaria (así como su habilidad en resolución de problemas).

En cualquier caso, la mejor forma de utilizar el tiempo dentro de contests es resolviendo el problema y desarrollando el algoritmo, que tardar escribiendo la solución.

Un lugar para practicar y analizar la velocidad para escribir es el sitio: `https://monkeytype.com/`.

Haz análisis de complejidad

En este libro se mencionó la gran importancia de realizar el análisis de complejidad cuando se ataque un problema, sin embargo aún así ¡hay mucha gente que se lo salta! No es algo de condenar absolutamente, pues cuando no existe un dominio en lo algoritmos utilizados, y cuando las constraints son muy chicas se puede omitir sin peligro este paso, sin embargo, en soluciones mínimamente más complejas se vuelva una obligación realizar el análisis de complejidad para cada solución desarrollada.

La razón de dicha regla es muy simple de ver, y de vivir si se hace caso omiso a este aviso. Imagina que terminaste tu solución, agotaste una hora y media de tu tiempo en contest para implementar esa solución, la probaron, hicieron lo corner cases y todo va bien, hasta el momento en el que hacen submission y se da el veredicto TLE en el primer test case, pierden

media hora más haciendo técnicas de optimización y ninguna logra que pase del tercer test case... la técnicas utilizadas no fueron las adecuadas, por muy optimizado que esté el algoritmo, si es por complejidad no habrá forma de obtener AC si no replanteando toda la solución con una técnica más sofisticada.

Identifica el tipo de problema que estás resolviendo

Los problemas en ICPC están dados por conjuntos de problemas en temas recurrentes. Los temas en los que se clasifican dichos problemas pueden ser únicos, o a veces combinados, por no mencionar las técnicas básicas que terminan siendo parte de una solución mayor.

Conforme se vaya adquiriendo la experiencia en resolución de problemas, en base a ésto se puede realizar la clasificación propuesta en la Tab. 3

No.	Categoría	Confianza y velocidad de resolución esperada
A	He resuelto este tipo antes	Estoy seguro de que puedo resolverlo de nuevo (y rápido)
B	He visto este tipo antes	En este momento no sé cómo resolverlo aún
C	No he visto este tipo antes	

Tabla 3: Clasificación de los tipos de problemas que un competidor puede hacer según su experiencia previa.[1]

Para considerarse *competitivo* se debe ser capaz de clasificar con seguridad problemas como tipo A, y minimizar los problemas de tipo B. Ésto significa adquirir el suficiente conocimiento algorítmico para desarrollar tus habilidades en programación de tal forma que se consideren muchos problemas clásicos como fáciles.

Conoce tu lenguaje de programación

Existen varios lenguajes de programación soportados en ICPC, sin embargo con la voz de la experiencia nosotros recomendamos C++, el que lenguaje con el que se trabajó la totalidad del libro.[1]

La maestría en los lenguajes de programación utilizado y sus rutinas pre-fabricadas son de extrema ayuda para las competiciones de programación. Conocer y estar familiarizado con la sintaxis básica, los elementos disponibles, y las librerías de un lenguaje harán que la implementación no sea un problema por estas razones. Ésto cobra aún más relevancia cuando se está empezando en la programación competitiva.

Trabaja en equipo

La programación competitiva es (por si no se había notado aún) un ejercicio que se realiza por equipo. Si bien las habilidades individuales de cada uno de los miembros del equipo será lo que determine la efectividad del mismo, sí se requiere sincronización y acoplamiento para maximizar su potencial.

Si se empieza en la programación competitiva por cuenta propia será difícil hallar equipo para participar en competiciones importantes, sin embargo en un entorno dedicado a ello (como en el CPCFI) ésta es una facilidad más que se va a la lista, y entonces es necesario buscar con quién hacer equipo primero, y buscar cómo congeniar con tu equipo después.

La manera de formar un buen equipo es buscando que, entre los tres, las habilidades que tengan se complementen. Un equipo cuyos miembros los tres son expertos en gráficas

1 Otro lenguaje estudiado en la literatura es Java, debido a las librerías tan versátiles y cómodas que contiene, tal como BigInteger/BigDecimal, GregorianCalendar, Regex... además de que es muy fácil depurar debido a la JVM.

será muchísimo más susceptible a tener deficiencias cuando se enfrenten en competiciones en las que, naturalmente, los problemas son variados y entran en cualquier tema. Contrario a como sucedería en un equipo donde haya un experto en gráficas, otro en programación dinámica, y otro en matemáticas, se forma un conocimiento con conjunto integral pues, si bien es verdad que todos deben estar instruidos en todos los temas, especializarse en todo es una labor imposible. Tampoco hay que tener en cuenta que no porque alguien se especialice en un tema sea des-letrado de los otros, la forma de trabajar (programar, resolver los problemas...) de los tres seguirá siendo complementaria.

Algunos consejo para agilizar el flujo de trabajo entre los tres se enlistan a continuación:

- Practica programar en papel. Ésto es útil cuando un compañero está ocupando la computadora para programar una solución, dejar lo más en claro el algoritmo en papel hará que, cuando se desocupe la computadora se pueda simplemente escribir el código lo más rápido posible, en lugar de sentarse a pensar en él. Precaución: Compilar en papel es una tarea difícil.

- Si un compañero actualmente está codificando su algoritmo, aprovecha el tiempo preparando test cases para su código. (Véase Sec. 6.2)

- Hazte amigo de tu equipo. Ser compañeros cercanos fuera de las competiciones y prácticas hace que se conozcan mejor, empaticen y, eventualmente, se sincronicen. Entre más se conozcan más fácil será trabajar junto.

- Trabajen juntos. Depende mucho de las personas del equipo, las habilidades que tengan, y varios factores más la forma en que le acomode a un equipo organizarse efectivamente. Aún así, puede ser buena idea cuando se está iniciando no *dividir el trabajo* entre todos si no están seguro de lo que hacen todavía, sino plantear juntos la solución y pensar en el planteamiento inicial ayudará a llegar a una conclusión más robusta. De igual manera, si aún no existe un dominio del lenguaje implementar entre los tres la solución ayudará a asegurar que la idea que se quiere implementar esté clara, y evitar retrasos por problemas con la sintaxis.

7

INTRODUCCIÓN A CODEFORCES

De manera somera, Codeforces es un sitio web en donde se aloja un abanico de problemas relacionados con la programación competitiva además, en él tienen lugar competiciones relacionadas con el mismo ámbito.

Cuando ingresamos al sitio lo primero que debemos de hacer es dirigirnos a la esquina superior derecha para registrarnos en el botón que se muestra en la Fig 6.

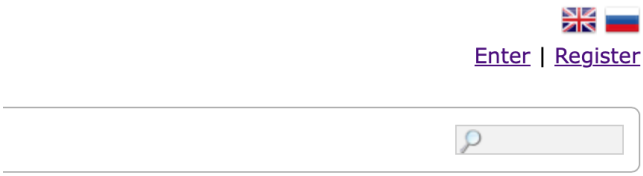


Figura 6: Inicio de sesión/registro en Codeforces. [5]

Una vez que contamos con una cuenta registrada e iniciamos sesión podemos acceder a todas las funciones disponibles del menú de la Fig. 7.

Figura 7: Menu principal de Codeforces. [5]

A continuación se describe brevemente cada una de las secciones:

1. **HOME**: Lleva a la página principal de Codeforces, donde puedes ver noticias, anuncios, y las próximas competencias. Es la página de inicio.

2. **TOP**: Muestra una lista de los usuarios mejor clasificados en Codeforces, junto con su puntaje de rating. Puedes explorar los programadores de mayor nivel y su rendimiento en la plataforma.

3. **CATALOG**: Proporciona una lista de artículos y tutoriales organizados en categorías. Estos artículos pueden abarcar desde algoritmos y estructuras de datos hasta consejos para concursos.

4. **CONTESTS**: Muestra una lista de competencias actuales, futuras y pasadas de Codeforces. Aquí puedes ver el calendario de competiciones, inscribirte en próximas competencias o revisar los resultados de concursos anteriores.

5. **GYM**: Permite acceder a una sección especial donde puedes practicar con problemas y concursos creados por otros usuarios o instituciones. Es ideal para entrenamientos en equipo y competencias privadas.

6. **PROBLEMSET**: Lista de todos los problemas disponibles en Codeforces, organizados por dificultad, etiquetas y popularidad. Desde aquí puedes elegir problemas específicos para resolver y mejorar tus habilidades.

7. **GROUPS**: Te permite unirte a grupos de usuarios o crear uno propio. Los grupos son útiles para organizar competiciones privadas, colaborar en entrenamientos y compartir problemas.

8. **RATING**: Muestra las clasificaciones de los usuarios según su rendimiento en competencias, con estadísticas de rating y progreso. Es útil para ver cómo se comparan tus habilidades con las de otros usuarios.

9. **EDU**: Acceso a la sección educativa de Codeforces, que ofrece lecciones y rondas educativas de programación competitiva para mejorar en algoritmos y técnicas comunes en concursos.

10. **API**: Permite a los usuarios y desarrolladores acceder a la API de Codeforces para recuperar información de problemas, usuarios y concursos. Esto es útil para crear herramientas y aplicaciones relacionadas con la plataforma.

11. **CALENDAR**: Muestra el calendario de todas las competencias programadas en Codeforces. Incluye tanto competencias oficiales como entrenamientos en el Gym, con fechas y horarios de inicio.

12. **HELP**: Lleva a una sección de ayuda donde se explican aspectos del uso de Codeforces. Puedes encontrar in-

formación sobre cómo funcionan las competiciones, el sistema de puntuación y cómo usar la plataforma.

Para comenzar a resolver problemas de tal manera que el nivel de dificultad sea bajo damos click derecho sobre problemset y posteriormente presionamos sobre el ícono mostrado en la Fig. 8. Lo siguiente que veremos será la lista de problemas ordenada de menor a mayor, debajo del símbolo de rayo aparecerá una puntuación cuyo valor mínimo ronda 800, como es de esperarse, mientras mayor sea ese número, mayor la dificultad.

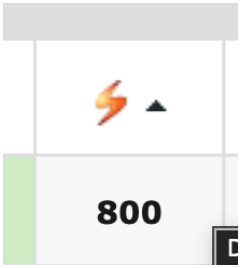


Figura 8: Filtrando por dificultad mínima el problemset de Codeforces. [5]

Una vez que seleccionamos un problema accederemos a la descripción cuya estructura ya tratamos en la sección anterior. Lo siguiente es implementar una solución en algún lenguaje de programación (preferentemente C++) y subir nuestra solución para que el jurado en línea sea quien emita un veredicto. Para completar esta tarea en la parte superior de la descripción buscamos una opción que indique *Submit* en donde aparecerá la interfaz mostrada en la Fig. 9.

Figura 9: Interfaz para hacer *submit* de un código en Codeforces. [5]

En dicho menú seleccionaremos el lenguaje en el cual está implementado nuestro algoritmo, posteriormente presionamos la opción que se encuentra en la esquina inferior izquierda llamada *choose file*, procedemos a seleccionar el archivo desde nuestra computadora y por último presionamos *submit.*

Al finalizar con el proceso anterior aparecerá e nuestra pantalla una lista de tareas que se están validando, en caso de que nuestro algoritmo cumpla con los requisitos aparecerá el siguiente mensaje:

4A - Watermelon	C++17 (GCC 7-32)	Accepted	30 ms	0 KB

Figura 10: Veredicto después un *submit*.

Somos muy conscientes que de las tareas modernas de informática requieren habilidades agudas en resolución de problemas y una creatividad tremenda, virtudes que no se pueden impartir a través de los libros. Recurrir a la bibliografía puede proveer conocimiento, pero el trabajo duro debe ser últimamente hecho por ti. Con práctica viene la experiencia, y con la experiencia viene la habilidad. Así que, ¡a seguir practicando!

Con esto hemos cubierto los temas indispensables para comenzar a practicar la Programación Competitiva, esperamos de corazón que este material les sea de utilidad.

BIBLIOGRAFÍA

[1] H. Felix y H. Steven, *Competitive Programming 3. The new Lower Bound of Programming Contests*. Morrisville, NC: USA: Lulu Independent Publish, 2013.

[2] A. Laaksonen, *Guide to Competitive Programming Learning and Improving Algorithms Through Contests* (Undergraduate Topics in Computer Science), 2nd. Helsinki, Finland: Springer, 2020, ISBN: 978-3-030-39356-4.

[3] A. Laaksonen, *Competitive Programmer's Handbook*. CSC Center for Science, 2018, vol. 5.

[4] B. Stroustrup, *The C++ Programming Language*, 4th. Michigan, United States: Addison-Wesley, 2013, ISBN: 978-0-321-56384-2.

[5] Codeforces Team, *Codeforces: Competitive Programming Platform*, 2024. dirección: https://codeforces.com/.

[6] Antti Laaksonen, *CSES Problem Set*, 2024. dirección: https://cses.fi/problemset/.

[7] Cppreference Contributors, *Cppreference: C++ Reference and Resources*, 2024. dirección: https://en.cppreference.com/w/.

[8] Cplusplus Contributors, *Cplusplus.com: The C++ Resources Network*, 2024. dirección: https://cplusplus.com/.