Introducción a la Programación Competetiva

Competitive Programming for Newbies

Matias Ramos

Universidad Tecnológica Nacional - Facultad Regional Santa Fe

Training Camp 2024

Gracias Sponsors!

Organizador

Diamond

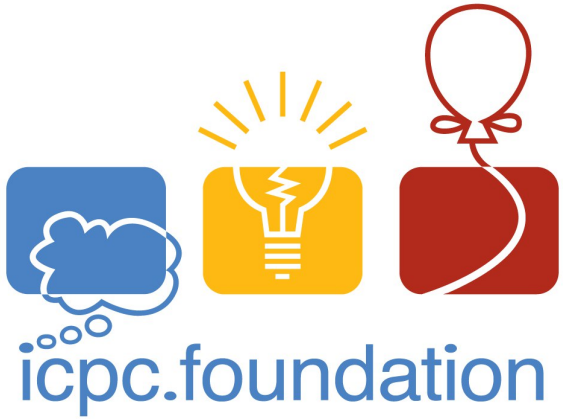
Gold

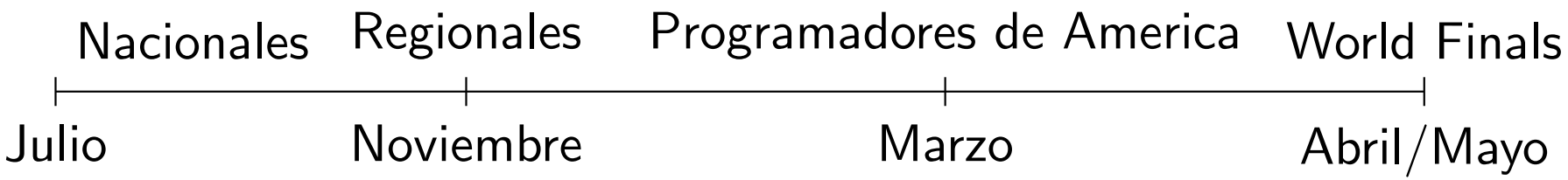
Temas

1. El Camino de ICPC
2. Juez Online - Codeforces
3. Análisis de Complejidad
4. Entrada y Salida
5. Estructuras fundamentales
6. Funciones clave (C++)
7. Overflow

Temas

El Camino de ICPC





- 3 personas
- 1 PC
- 10-14 problemas
- 5 horas

Temas

Codeforces

- Juez online ruso.
- Funciona como test de caja negra, obtenemos una respuesta casi instantanea.
- Tiene rondas casi semanales, sistema de rating para usuarios similar al ELO.
- Usaremos este juez para todas las simulaciones.

Submit y respuestas

Veamos algunos submits míos para conocer algunas respuestas.

Temas

Cantidad de operaciones

¿Cuántas operaciones "entran en tiempo"?

- Hasta 10^7 : ¡Todo OK!
- Entre 10^7 y hasta 10^9: "Tierra incógnita". Puede cambiar mucho según el costo de las operaciones.
- Más de 10^9: Casi con certeza total será demasiado lento.

Lo anterior asume:

- Hardware no extremadamente viejo.
- Límites de tiempo del orden de "segundos" (ni minutos, ni milésimas).

funcion O

- La complejidad del tiempo de un algoritmo se anota como $O(\dots)$
- Los tres puntos pueden ser una función.
- La variable n normalmente se refiere al tamaño del input.

Algunos ejemplos

```
for (int i = 1; i <= n; i++) {
 //codigo
}
```

$O(n)$

```
for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= n; j++) {
 //codigo
 }
}
```

$O(n^2)$

Estimando eficiencia

input size	required time complexity
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ or $O(n)$
n is large	$O(1)$ or $O(\log n)$

Temas

¿Por qué conviene hacer eficiente la E/S?

- En problemas de complejidad lineal o similar, las operaciones de E/S pueden insumir un porcentaje importante del tiempo total de ejecución, que es lo que se mide en la mayoría de las competencias.

¿Por qué conviene hacer eficiente la E/S?

- En problemas de complejidad lineal o similar, las operaciones de E/S pueden insumir un porcentaje importante del tiempo total de ejecución, que es lo que se mide en la mayoría de las competencias.
- Aún si los tiempos elegidos por el jurado son generosos, y es posible con una solución esperada resolver el problema aún con mecanismos de E/S ineficientes, usar formas eficientes de hacer E/S nos permitirá siempre "zafar" con programas más lentos que si no lo hiciéramos así.

¿Por qué conviene hacer eficiente la E/S?

- En problemas de complejidad lineal o similar, las operaciones de E/S pueden insumir un porcentaje importante del tiempo total de ejecución, que es lo que se mide en la mayoría de las competencias.
- Aún si los tiempos elegidos por el jurado son generosos, y es posible con una solución esperada resolver el problema aún con mecanismos de E/S ineficientes, usar formas eficientes de hacer E/S nos permitirá siempre "zafar" con programas más lentos que si no lo hiciéramos así.
- Existen diferencias muy simples y pequeñas en la forma de realizar E/S en los programas, que generan grandes diferencias en el tiempo total insumido por estas operaciones. Conocer estas diferencias es entonces obtener un beneficio relevante con muy poco esfuerzo.

Funciones printf y scanf

- En C plano, la forma de E/S más utilizada son las funciones printf y scanf. Estas funciones son eficientes, y es la forma recomendada de realizar entrada salida en este lenguaje.
- Ejemplo:

```
#include <stdio.h>
int main() {
    int x,y;
    scanf("%d%d", &x, &y);
    printf("%d\n", x+y);
}
```

Funciones printf y scanf

- En C++, las mismas funciones scanf y printf siguen disponibles, y siguen siendo una opción eficiente para aquellos que estén acostumbrados o gusten de usarlas.
- Ejemplo:

```
#include <cstdio>
using namespace std;
int main() {
    int x,y;
    scanf("%d%d", &x, &y);
    printf("%d\n", x+y);
}
```

Streams cin y cout

- La forma elegante de hacer E/S en C++ es mediante los streams cin y cout (Y análogos objetos fstream si hubiera que manipular archivos específicos en alguna competencia).
- Ejemplo:

```
#include <cstdio>
using namespace std;
int main() {
    int x,y;
    cin >> x >> y;
    cout << x+y << endl;
}
```

Por defecto en casos usuales, cin y cout son lentos

- La eficiencia relativa de cin y cout vs scanf y printf dependerá del compilador y arquitectura en cuestión.
- Dicho esto, en la mayoría de los compiladores y sistemas usuales utilizados en competencia, cin y cout son por defecto mucho más lentos que scanf y printf.
- Veremos algunos trucos para que cin y cout funcionen más rápido. Con ellos, en algunos sistemas comunes funcionan más rápido que printf y scanf, pero la diferencia es muy pequeña.
- En otras palabras, aplicando los trucos que veremos a continuación, da igual usar cin y cout o printf y scanf, ambas son eficientes.

Primera observación: endl

- El valor "endl" no es solo un fin de línea, sino que además ordena que se realice un flush del buffer.
- De esta forma, imprimir muchas líneas cortas (un solo entero, un solo valor Y/N, etc) realiza muchas llamadas a escribir directamente al sistema operativo, para escribir unos poquitos bytes en cada una.
- Solución: utilizar `\n` en su lugar. Esto es un sencillo caracter de fin de línea, que no ejecuta un flush del buffer.
- Ejemplo:

```
#include <cstdio>
using namespace std;
int main() {
    int x,y;
    cin >> x >> y;
    cout << x+y << "\n";
}
```

Segunda observación: sincronización con stdio

- Por defecto, cin y cout están sincronizados con todas las funciones de stdio (notablemente, scanf y printf). Esto significa que si usamos ambos métodos, las cosas se leen y escriben en el orden correcto.
- En varios de los compiladores usuales esto vuelve a cin/cout mucho más lentos, y si solamente usamos cin y cout pero nunca scanf y printf, no lo necesitamos.
- Solución: utilizar `ios::sync_with_stdio(false)` al iniciar el programa, para desactivar esta sincronización. Notar que si hacemos esto, ya no podemos usar printf ni scanf (ni ninguna función de stdio) sin tener resultados imprevisibles.
- Desactivar la sincronización también puede tener efectos al utilizar más de un thread. Esto no nos importa en ICPC.

Segunda observación: sincronización (ejemplo)

Esta optimización tiene efectos muy notorios, típicamente reduce el tiempo de ejecución a la mitad en varios jueces online comunes.
Ejemplo:

```
#include <cstdio>
using namespace std;
int main() {
    ios::sync_with_stdio(false);
    int x,y;
    cin >> x >> y;
    cout << x+y << "\n";
}
```

Tercera observación: dependencia entre cin y cout

- Por defecto, cin está atado a cout, lo cual significa que siempre antes de leer de cin, se fuerza un flush de cout. Esto hace que programas interactivos funcionen como se espera.
- Cuando solo se hacen unas pocas escrituras con el resultado al final de toda la ejecución, esto no tiene un efecto tan grande.
- Si por cada línea que leemos escribimos una en la salida, este comportamiento fuerza un flush en cada línea, como hacía endl.
- Solución: utilizar `cin.tie(nullptr)` al iniciar el programa, para desactivar esta dependencia. Notar que si hacemos esto, tendremos que realizar flush de cout manualmente si queremos un programa interactivo.

Tercera observación: dependencia (ejemplo)

```
#include <cstdio>
using namespace std;
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int x,y;
    cin >> x >> y;
    cout << x+y << "\n";
}
```

Ejemplo final con las 3 técnicas

- Eliminar sincronización con stdio
- Eliminar dependencia entre cin y cout
- No utilizar endl

```
#include <cstdio>
using namespace std;
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int x,y;
    cin >> x >> y;
    cout << x+y << "\n";
}
```

Temas

Vector

- vector<int> en C++, con push_back y pop_back
- ArrayList<Integer> en Java, con .add y .remove(list.size()-1)
- list en Python (listas usuales como [1,2,3]), con .append y .pop
- acceso con lista[i] o lista.get(i)
- Sirven como pila
- Las operaciones anteriores son $O(1)$ (amortizado)

Queue

- queue<int> en C++, con push, front y pop
- ArrayDeque<Integer> en Java, con .add, .getFirst y .remove
- collections.deque en Python, con .append, deque[0] y .popleft
- Sirven como cola
- Las operaciones anteriores son $O(1)$ (amortizado)

Deque

- deque<int> en C++, con push_front, push_back, pop_front y pop_back
- ArrayDeque<Integer> en Java, con .addFirst, .addLast, .removeFirst y .removeLast
- collections.deque en Python, con .appendleft, .append, .popleft y .pop
- acceso con lista[i] (no se puede en java!!)
- Sirven como cola de dos puntas
- Las operaciones anteriores son $O(1)$ (amortizado)

HashSet

- unordered_set<int> en C++
- HashSet<Integer> en Java
- set en Python
- Permiten insertar, borrar y consultar pertenencia en $O(1)$

HashMap

- unordered_map<int,int> en C++
- HashMap<Integer,Integer> en Java
- dict en Python
- Permiten insertar, borrar y consultar pertenencia en $O(1)$
- Son casi iguales a los HashSet, pero guardan un valor asociado a cada elemento

Set

- set<int> en C++
- TreeSet<Integer> en Java (googlear docs de NavigableSet)
- En Python no hay. ¡Ojo!
  collections.OrderedDict es otra cosa (LinkedHashMap de Java)
- Permiten insertar, borrar, consultar pertenencia y hacer `s.lower_bound` o `s.upper_bound` en $O(\lg N)$

TreeMap

- map<int,int> en C++
- TreeMap<Integer,Integer> en Java (googlear docs de NavigableMap)
- No confundir "collections.OrderedDict" (que no es..)
- Permiten insertar, borrar, consultar pertenencia y hacer `m.lower_bound` o `m.upper_bound` en $O(\lg N)$
- Son casi iguales a los TreeSet, pero guardan un valor asociado a cada elemento

Temas

Funciones clave (C++)

- `sort` (algorithm) [begin, end]
- `lower_bound` , `upper_bound`, `equal_range` (algorithm) [begin, end, val]. ¡¡NO USAR CON SET Y MAP!!
- `find` (algorithm) [begin, end, val]
- `max_element`, `min_element` (algorithm) [begin, end]

Temas

Overflow

- Si uno no presta atención, es extremadamente común tener errores por culpa del overflow de enteros.
- Es importante acostumbrarse a siempre revisar las cotas de todas las entradas, y calcular los posibles valores máximos de los números que maneja el programa. Suele ser multiplicar cotas de la entrada.
- Ante la duda preferir tipos de 64 bits (long long en C++, long en Java) a tipos de 32 bits (int).
- Ojo con
  `long long mask = 1 << 33;`
  que está mal. Debería ser
  `long long mask = 1LL << 33;`
- int: hasta $2^{31} - 1 = 2.147.483.647$. Algo más de dos mil millones.
- long long: hasta $2^{63} - 1$. Más de 10^{18}, pero menos que 10^{19}.

Consultas

Pueden consultarme durante esta semana, o me pueden enviar un mail a:

- mramos@frsf.utn.edu.ar