Introducción a la programación competitiva (IPC)

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Training Camp 2019: UNC - FAMAF

Contenidos

Contenidos

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).

Contenidos

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 milllones.
- long long: hasta $2^{63} - 1$. Más de 10^{18}, pero menos que 10^{19}.

Contenidos

Aritmética modular

- A veces, en problemas donde una respuesta sería muy grande, para no tener que manejar enteros enormes se pide "módulo M".
- La aritmética "módulo M" consiste en hacer todas las cuentas tomando el resto de la división por M.
- Si $M > 0$ el resultado queda siempre entre $-M + 1$ y $M - 1$ inclusive.
- El resultado final de hacer las cuentas modulo M es el correcto, si solo hay +, - y multiplicaciones.
- Es decir:
  - (a+b) %M en lugar de a+b
  - (a*b) %M en lugar de a*b
  - (a-b) %M en lugar de a-b
- Ojo con la resta que puede generar negativos. $((x\,\%M) + M)\,\%M$ siempre lo deja positivo.

Contenidos

Fórmulas matemáticas fundamentales

- $1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- En general para progresiones aritméticas: *promedio* $\times$ *cantidad*, y el promedio siempre es $\frac{primero+ultimo}{2}$
- $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1} = \frac{1-x^{n+1}}{1-x}$
- $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

Contenidos

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]

Contenidos

Testing por fragmentos

- Cuando un programa va a tener que “calcular cosas independientes”, conviene escribirlas por separado y testearlas independientemente.
- Por ejemplo, supongamos que en un problema es útil tener una función $f(i, j)$ que calcula la suma de los elementos de un arreglo entre i y j.
- La función anterior tiene una consigna bien definida y tiene sentido testearla independientemente con valores de i, j para ganar confianza en que no tiene error.
- Una vez que ganamos confianza en la f, podemos revisar con cuidado el código en otras partes que usan f.
- Si testeando f encontramos un caso donde falla, ya sabemos que hay un bug **en la f:** con un caso que falla para todo el programa, no sabríamos dónde hay bugs.

TDD sobre los fallos

- Al encontrar un bug en el código o en la idea, que sabemos cómo solucionar, es conveniente buscar y escribir un caso de prueba donde el programa falle, **antes** de solucionar el bug.
- Es increíblemente común hacer esto, ir a solucionar el bug, volver a correr ¡¡Y descubrir que sigue dando mal!!
- Es frecuente que un caso que logra hacer saltar un bug, también haga saltar otros bugs, así que tener el caso ayuda.
- Regla muy útil: **no corregir el código hasta no tener un caso de prueba en el que el programa falle**.
- Excepción: si solucionar el código y mandar en ese problema particular es muy fácil (por ejemplo es poner un +1), pero buscar y armar un caso que rompa es difícil, puede ser razonable corregir y enviar.

Flags del compilador (C++)

Utilizar los flags indicados en `http://wiki.oia.unsam.edu.ar/cpp-avanzado/opciones-gcc`

Macro DBG (C++)

```
#define DBG(x) cerr << #x << " = " << (x) << endl
```

Contenidos

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

TreeSet

- set<int> en C++
- TreeSet<Integer> en Java
- En Python no hay. ¡Ojo! collections.OrderedDict **es otra cosa**
- 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
- “collections.OrderedDict” en Python **(pero 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

Contenidos

Suma de prefijos (Tabla aditiva 1D)

- Computamos una tabla con las sumas parciales de un arreglo
- Por ejemplo, para 1 3 10 15 computamos 0 1 4 14 29
- Nota: `partial_sum` (numeric) y su inversa `adjacent_differences` (numeric)
- Ahora restando podemos obtener en cualquier momento, cualquier subrango.

Contenidos

¿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(NULL);
    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(NULL);
    int x,y;
    cin >> x >> y;
    cout << x+y << "\n";
}
```

InputStreams, OutputStreams, Readers, Writers

- En Java existe la distinción entre los Streams (bytes) y los Readers / Writers (caracteres unicode).
- Aún siendo todo ASCII, para archivos de texto uno termina trabajando siempre con readers y writers porque tienen las funciones más cómodas.
- El “análogo” de cin y cout en Java es System.in y System.out.
- Sin embargo, hay que tener cierto cuidado ya que al operar con ellos directamente, no se bufferean las operaciones, y tenemos un problema de permanente flushing, similar al que ocurría en C++ con endl.
- Particularmente, hacer `System.out.println(x)` es exactamente como `cout << x << endl`, y queremos evitarlo.

Ejemplo típico de I/O con Java

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

class HelloWorld {
    public static void main(String [] args) throws Exception {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        long total = 0;
        for (int i = 0; i < n; i++) {
            long x = scanner.nextLong();
            total += x;
            System.out.println(total);
        }
    }
}
```

Esto es lento, porque no usa buffers, lee y escribe directamente.

Introduciendo Buffers

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

class HelloWorld {
    public static void main(String [] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));

        Scanner scanner     = new Scanner(br);
        PrintWriter printer = new PrintWriter(bw);
        int n = scanner.nextInt();
        long total = 0;
        for (int i = 0; i < n; i++) {
            long x = scanner.nextLong();
            total += x;
            printer.println(total);
        }
        printer.close(); // En codigo real, usar try-finally o try-with-resources
    }
}
```

¡¡Notar el close!! No se puede omitir. Al usar buffers, printer.println no imprime en el momento, y sin flushear al final pueden quedar cosas pendientes de escribir en la salida (se observa una salida “cortada”).

En versiones nuevas de Java...

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

class HelloWorld {
    public static void main(String [] args) throws Exception {
        Scanner scanner     = new Scanner(System.in);
        PrintWriter printer = new PrintWriter(System.out);
        int n = scanner.nextInt();
        long total = 0;
        for (int i = 0; i < n; i++) {
            long x = scanner.nextLong();
            total += x;
            printer.println(total);
        }
        printer.close(); // En codigo real, usar try-finally o try-with-resources
    }
}
```

En versiones nuevas, esto “zafaría”, gracias a que Scanner y PrintWriter usan buffers internos. Notar que usar System.out y System.in directamente sin envolverlos nunca usan buffers.
No obstante, la versión anterior es la jugada segura todo terreno. Si el rendimiento de E/S puede importar, **siempre usar buffers**.

Más eficientes, pero más incómodos

Podemos evitar por completo PrintWriter y Scanner y resolver todo con BufferedWriter y BufferedReader:

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

class HelloWorld {
    public static void main(String [] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));

        int n = Integer.valueOf(br.readLine());
        long total = 0;
        for (int i = 0; i < n; i++) {
            long x = Long.valueOf(br.readLine());
            total += x;
            bw.write(String.valueOf(total));
            bw.newLine();
        }
        bw.close(); // En codigo real, usar try-finally o try-with-resources
    }
}
```

La diferencia entre PrintWriter y BufferedWriter no es muy grande (En casos como el ejemplo, < 10 %).
La diferencia entre Scanner y BufferedReader es potencialmente muy grande (puede ser un 50 %). Otra función a evitar en estos casos es `String.split`, que es bastante lenta.

No todos los python son iguales

Python 2 $\neq$ Python 3
Tienen algunas diferencias en una de las formas eficientes de E/S.

Instrucciones de entrada

Python2:

1. input() vs raw_input()
2. raw_input() devuelve un string con la siguiente línea de `stdin`.
3. input() interpreta la siguiente línea como una expresión y devuelve su resultado.
4. ¿Cuál es más eficiente?

Instrucciones de entrada (cont.)

Python3: Solo hay input(). No hay más raw_input().

1. Pero ahora input() es lo que en python2 era el raw_input()
2. Si uno quisiera el “viejo input()”, en python3 se hace con eval(input())

Contenidos

Maximum subarray sum

- Se tiene un arreglo de números enteros, positivos y negativos.
- ¿Cuál es el subarreglo de mayor suma?

Movimiento de bloques

- Tenemos bloques indistinguibles, ubicados en ciertas posiciones iniciales de un arreglo.
- Queremos llevarlos a una configuracion final de los bloques.
- Puede haber más de un bloque en la misma casilla en cualquier momento.
- ¿Cuál es la mínima cantidad de movimientos necesaria?