

ALGORITMOS Y ESTRUCTURAS DE DATOS III

# Trabajo Práctico N°1

Carla Livorno 424/08 carlalivorno@hotmail.com  
Completen!!!

Abril 2010

# Índice

<b>Introducción</b>	<b>2</b>
<b>1. Problema 1</b>	<b>2</b>
1.1. Explicación . . . . .	2
1.1.1. Análisis de complejidad . . . . .	2
1.2. Detalles de la implementación . . . . .	3
1.3. Pruebas y Resultados . . . . .	3
<b>2. Problema 2</b>	<b>4</b>
2.1. Explicación . . . . .	4
2.1.1. Análisis de complejidad . . . . .	4
2.2. Detalles de la implementación . . . . .	4
2.3. Pruebas y Resultados . . . . .	4
<b>3. Problema 3</b>	<b>5</b>
3.1. Explicación . . . . .	5
3.1.1. Análisis de complejidad . . . . .	5
3.2. Detalles de la implementación . . . . .	6
3.3. Pruebas y Resultados . . . . .	7
<b>4. Compilación y ejecución de los programas</b>	<b>8</b>
<b>5. Conclusiones</b>	<b>8</b>

# Introducción

Este trabajo tiene como objetivo la aplicación de diferentes técnicas algorítmicas para la resolución de tres problemas particulares, el cálculo de complejidad teórica en el peor caso de cada algoritmo implementado, y la posterior verificación empírica.

El lenguaje utilizado para implementar los algoritmos de todos los problemas fue C/C++

## 1. Problema 1

Dados  $b, n \in \mathbb{N}$  calcular  $b^n \bmod n$

### 1.1. Explicación

Para la resolución del problema, usamos la técnica de *Divide & Conquer* para tratar de minimizar la cantidad de multiplicaciones.

#### 1.1.1. Análisis de complejidad

Esta vez, elegimos un modelo logarítmico para analizar el algoritmo, ya que las operaciones que aplicamos, en teoría, dependen del logaritmo del número en cuestión, o dicho de otra forma, del tamaño de la entrada. No obstante, en los resultados muchas de ellas tienen costo uniforme por trabajar con números de tamaño acotado (`unsigned long long int`) para simplificar la implementación.

Por esto también analizamos un poco la complejidad obtenida bajo el modelo uniforme.

```

while  $m > 0$ 
  if  $m$  es impar
     $tmp \leftarrow tmp \cdot c$ 
     $tmp \leftarrow tmp \bmod n$ 
   $m \leftarrow \frac{m}{2}$ 
   $c \leftarrow c^2$ 
   $c \leftarrow c \bmod n$ 

```

1 iteración  $m = n$   
 2 iteración  $m = \frac{n}{2}$   
 3 iteración  $m = \frac{n}{2^2}$   
 4 iteración  $m = \frac{n}{2^3}$   
 k iteración  $m = \frac{n}{2^{k-1}} = 1$

Como el algoritmo termina cuando  $m = 1$  entonces,

$$\frac{n}{2^{k-1}} = 1$$

$$n = 2^{k-1}$$

$$\log_2 n = \log_2 2^{k-1}$$

$$\log_2 n = k - 1 \Rightarrow k = \log_2(n) + 1$$

Entonces, el algoritmo hace  $\log_2(n) + 1$  iteraciones, cada una con una cantidad constante de operaciones. Por lo tanto, la cantidad de operaciones que hace el algoritmo es del orden de  $\log_2 n$ .

## 1.2. Detalles de la implementación

El algoritmo primero calcula  $c = b \bmod n$  para

## 1.3. Pruebas y Resultados

## **2. Problema 2**

### **2.1. Explicación**

#### **2.1.1. Análisis de complejidad**

### **2.2. Detalles de la implementación**

### **2.3. Pruebas y Resultados**

## 3. Problema 3

### 3.1. Explicación

Para cada instancia tenemos una lista que contiene para cada programador su horario de ingreso a la empresa y otra con su horario de egreso. Además, tenemos guardado en cada momento la cantidad máxima de programadores en simultáneo.

Dado que ambas listas se encuentran ordenadas, nuestro algoritmo las recorre decidiendo a cada momento si se produce un ingreso o un egreso, es decir, si el horario que sigue en la lista de ingresos es anterior a la de egresos implica que hubo un ingreso, en caso contrario un egreso.

Cuando una persona ingresa a la empresa se incrementa el contador de la cantidad de programadores en simultáneo en el horario actual. Así, cuando se produce un egreso se compara si la cantidad de programadores dentro de la empresa previo a dicho egreso es mayor a la máxima cantidad de programadores en simultáneo hasta el momento, de ser así, actualizamos el máximo.

Luego se descuenta el recientemente egresado del contador “cantidad de programadores en simultáneo”.

Este procedimiento se repite hasta haber visto todos los ingresos, lo que nos garantiza tener el máximo correspondiente, ya que a partir de ese momento sólo se producirían egresos.

#### 3.1.1. Análisis de complejidad

Elegimos el modelo uniforme para analizar la complejidad de este algoritmo porque el tamaño de los elementos es acotado y por lo tanto todas las operaciones elementales son de tiempo constante.

Sea  $n$  la cantidad de programadores.

```

programadores_en_simultaneo(ingresos, egresos)
    max, tmp, j, k  $\leftarrow$  0
    while (j < n)                                     O(n)
        if (ingresos[j]  $\leq$  egresos[k])
            tmp  $\leftarrow$  tmp + 1
            j  $\leftarrow$  j + 1
        else
            if (tmp > max)
                max  $\leftarrow$  tmp
            tmp  $\leftarrow$  tmp - 1
            k  $\leftarrow$  k + 1
    if (tmp > max)
        max  $\leftarrow$  tmp
    return max

```

Cada programador tiene un ingreso y un egreso, por lo tanto la lista de ingresos y la lista de egresos tienen longitud  $n$ .

El algoritmo en todos los casos recorre completamente la lista de ingresos, por lo que el peor caso es cuando el último ingreso y el último egreso corresponden al mismo programador, ya que para registrar éste último ingreso, también tuvo que recorrer toda la lista de egresos. Por este motivo, podemos inferir que a lo sumo se realizan  $2n - 1$  iteraciones. En cada una de estas iteraciones tenemos un costo constante de operaciones, que no modifican la complejidad en el análisis asintótico. La complejidad algorítmica entonces, es  $O(n)$ .

### 3.2. Detalles de la implementación

Guardamos los horarios de ingreso de todos los programadores (de la misma forma que están en el archivo de entrada, es decir, en orden creciente) en un arreglo de *strings* (los cuales representan un horario en formato “HH:MM:SS”) de tamaño  $n$ , donde  $n$  es la cantidad de programadores. Además guardamos otro arreglo del mismo tamaño con los horarios de egreso.

A medida que vamos recorriendo los arreglos *ingresos* y *egresos* necesitamos decidir si el horario de ingreso del programador  $j$  es anterior o posterior al horario de egreso del programador  $i$ , esto lo hacemos comparando los *strings*

por menor o igual (que el horario de ingreso del programador  $j$  sea el mismo que el horario de egreso del programador  $i$  significa que ambos estuvieron en simultáneo en la empresa justamente en ese horario ya que se considera que un programador permanece dentro de la empresa desde su horario de ingreso hasta su horario de egreso, incluyendo ambos extremos). Si la comparación resulta verdadera significa que el programador  $j$  ingresa a la empresa por lo que incrementamos el contador de programadores en simultáneo en ese horario. En caso contrario lo decrementamos ya que el programador  $i$  egresa. Antes de decrementar dicho contador verificamos si la cantidad de programadores en simultáneo previo al egreso de  $i$  es mayor a  $max$  (máxima cantidad de programadores en simultáneo calculada hasta el momento) y de ser necesario actualizamos  $max$ .

Una vez que terminamos de recorrer la lista de ingresos, actualizamos  $max$  ya que desde el último egreso visto se pueden haber producido nuevos ingresos. Una vez hecho esto tenemos determinada la mayor cantidad de programadores que están simultáneamente dentro de la empresa.

### 3.3. Pruebas y Resultados



## 4. Compilación y ejecución de los programas

Para compilar los programas se puede usar el comando **make** (Requiere el compilador **g++**). Se pueden correr los programas de cada ejercicio ejecutando `./bn_mod_n`, `./ronda_de_amigas` y `./programadores` respectivamente.

Los programas leen la entrada de `stdin` y escriben la respuesta en `stdout`. Para leer la entrada de un archivo `Tp1EjX.in` y escribir la respuesta en un archivo `Tp1EjX.out` se puede usar:

```
./(ejecutable) <Tp1EjX.in >Tp1EjX.out
```

Para medir los tiempos de ejecución: `./(ejecutable) time`

Para contar la cantidad de operaciones: `./(ejecutable) count`. Devuelve la cantidad de operaciones de cada instancia.

## 5. Conclusiones