Algoritmos y estructuras de datos III

Trabajo Práctico Nº1

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

Abril 2010

Índice

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 *mod* n

1.1. Explicación

Una solución factible al problema es utilizar un algoritmo recursivo basado en la técnica *Divide&Conquer*. La idea seria quedarnos cada vez con un problema más chico, dividiendo el exponente a la mitad y resolver recursivamente dicho problema hasta llegar a uno suficientemente chico (en este caso $n == 2$) para luego combinar las soluciones elevando al cuadrado lo ya calculado y obtener así una solución al problema original. En el caso donde n fuese impar se resuelve el problema para $n - 1$ a través del mismo procedimiento y luego se multiplica el resultado por b.

Para la resolución del problema decidimos utilizar un algoritmo iterativo frente a uno recursivo debido al uso que este último hace de la pila, lo cual implica repetidos accesos a memoria disminuyendo asi la performance del algoritmo.

En cada iteración el algoritmo divide el exponente a la mitad y eleva al cuadrado el acumulador (en la primer iteración b). Cuando llega a un exponente impar multiplica el resultado por el acumulador y continua elevando dicho acumulador. Cuando el exponente llega a uno repite lo anterior y termina. Es decir, el resultado es la productoria de los resultados parciales obtenidos en los exponentes impares.

Al tener un acumulador que nos guarda las potencias ya calculadas el algoritmo evita hacer calculos repetidos y logra minimizar la cantidad de multiplicaciones.

Tanto la versión recursiva como la iterativa toman módulo n luego de cada multiplicación para asegurarse que el resultado entra en el tamaño de la variable (suponiendo que $n-1 \times n-1$ entra).

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.

Sea $m = n$

while $m > 0$	$O(\log^3 n)$
if m es impar	$O(\log m)$
$tmp \leftarrow tmp * b$	$O(\log^2 n)$
$tmp \leftarrow tmp \ mod \ n$	$O(\log^2 n)$
$m \leftarrow \frac{m}{2}$	$O(\log m)$
$b \leftarrow b^2$	$O(\log^2 n)$
$b \leftarrow b \ mod \ n$	$O(\log^2 n)$
return tmp	

Cantidad de operaciones

$$\begin{array}{rlcl} \text{1er} & \text{iteración} & \rightarrow & m = n \\ \text{2da} & \text{iteración} & \rightarrow & m = \frac{n}{2} \\ \text{3er} & \text{iteración} & \rightarrow & m = \frac{n}{2^2} \\ \text{4ta} & \text{iteración} & \rightarrow & m = \frac{n}{2^3} \\ & \vdots & & \vdots \\ k\text{-ésima} & \text{iteración} & \rightarrow & m = \frac{n}{2^{k-1}} = 1 \end{array}$$

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

$$\begin{aligned} \frac{n}{2^{k-1}} &= 1 \\ n &= 2^{k-1} \\ \log n &= \log\ 2^{k-1} \\ \log n &= k-1 \Rightarrow k = \log(n)+1 \end{aligned}$$

es decir, hace $\log(n)+1$ iteraciones, cada una con una cantidad constante de operaciones.

Por lo tanto, la cantidad de operaciones que hace el algoritmo es del orden de $\log n$ y $\mathrm{O}(\log n) \subset \mathrm{O}(n)$.

Complejidad en el modelo logarítimico

Como $m \leq n$ y la función logaritmo es estrictamente creciente, $\log m \leq \log n$. De la misma manera, $\log b$ y $\log tmp$ están acotados por $\log n$. Por lo tanto, la complejidad en el modelo logarítmico es:
$\mathrm{O}(\log(n) * (\log(n) + \log^2(n) + \log^2(n) + \log(n) + \log^2(n) + \log^2(n))) =$
$= \mathrm{O}(\log(n) * (2\log(n) + 4\log^2(n))) = \mathrm{O}(2\log^2 n + 4\log^3 n)$

Luego, por definición,

$$\mathrm{O}(2\log^2 n + 4\log^3 n) = \mathrm{O}(max(2\log^2 n, 4\log^3 n)) = \mathrm{O}(\log^3 n).$$

Entonces, la complejidad del algoritmo resulta ser: $\mathrm{O}(log^3 n)$

En función del tamaño de la entrada

El tamaño de la entrada t es $\log n$ ya que b es acotado ($n = 2^t$). Entonces la complejidad del algoritmo en función del tamaño de entrada es: $\mathrm{O}(\log^3 2^t)$=$\mathrm{O}(t^3)$.

1.2. Detalles de la implementación

La siguiente tabla representa la correspondencia entre las variables de entrada (b y n) en cada iteración del algoritmo implementado:

iteración	1	2	3	...	k
n	n	$\frac{n}{2}$	$\frac{n}{2^2}$	...	$\frac{n}{2^{k-1}}$
b	b	b^2	b^4	...	$b^{2^{k-1}}$

Sean

$A_k = \frac{n}{2^{k-1}}$ bla bla bla, y

$Z_k = impar(A_k) * b^{2^{k-1}}$ [2] la sucesión que tiene los valores de b correspondientes a los n imp

entonces el cálculo hecho por el algoritmo está dado por

$$\prod_{i=1}^{k} Z_i = b^n$$

Además, luego de cada multiplicación toma el módulo ya que

$$b^k * b^{n-k} \ mod \ n = (b^k \ mod \ n) * (b^{n-k} \ mod \ n) \ \forall \ k \leq n^{[1]}$$

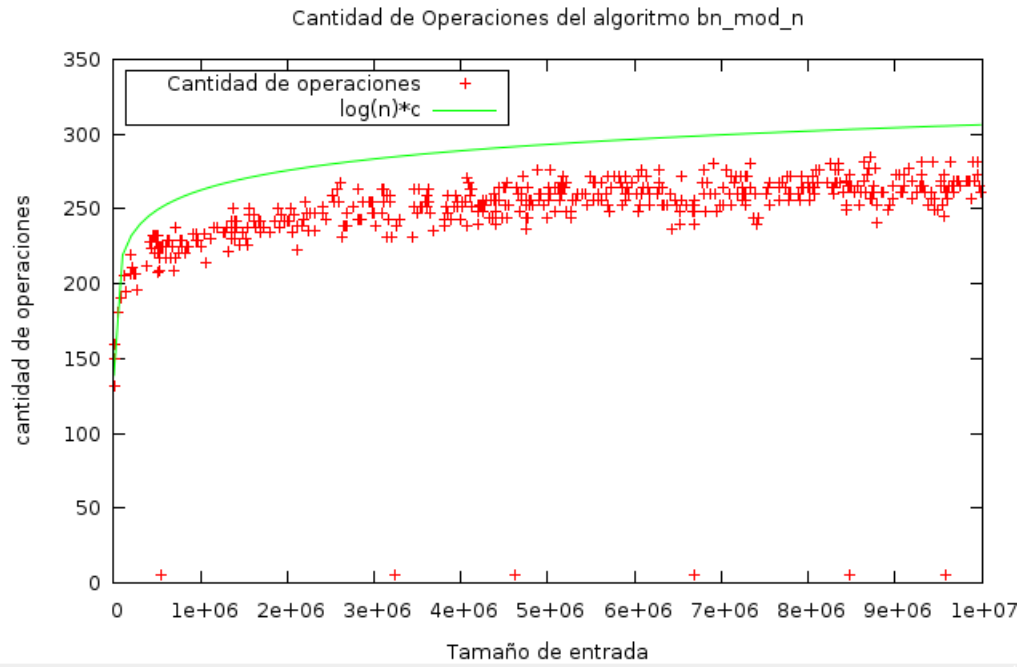
De esta manera, incluso si el cálculo de b^n es un número tan grande que no entra en el tamaño de la variable, se va a poder realizar sin problemas (suponiendo que $(n-1)^2$ entra en una variable).

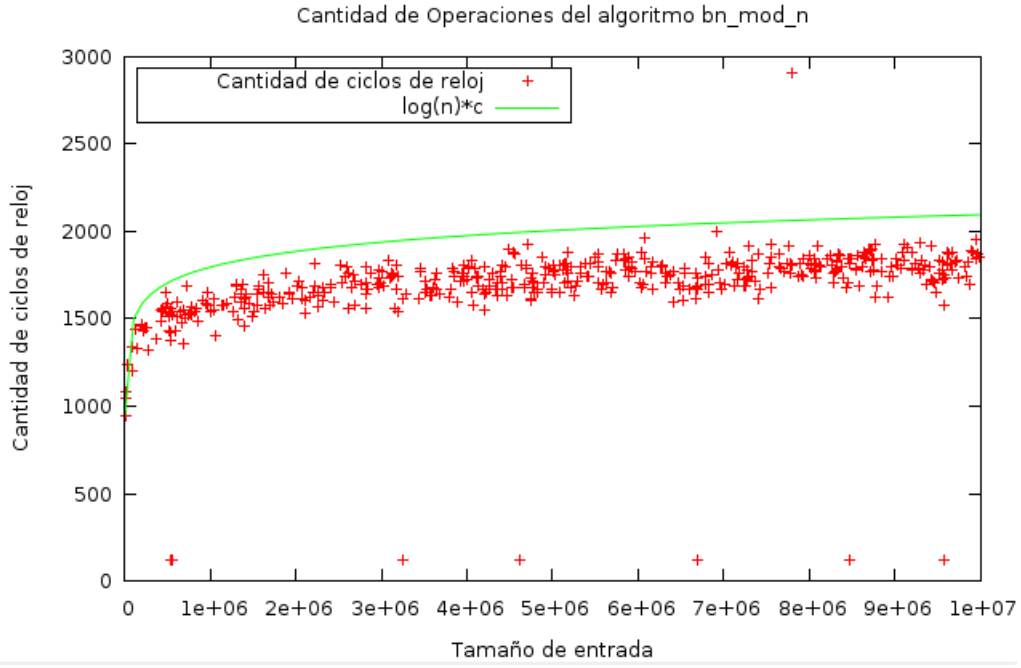
[1] impar se define como:

$$impar(x) = \begin{cases} 1 & \text{si } x \text{ es impar} \\ 0 & \text{sino} \end{cases}$$

[2] Por propiedades del módulo $x * y \ mod \ z = (x \ mod \ z) * (y \ mod \ z)$

1.3. Pruebas y Resultados





2. Problema 2

Se tiene n chicas cada de estas tiene k amigas, con $k < n$. Decidir si se puede formar una ronda que las contega a todas donde cada una de las chicas este de la mano de dos de sus amigas.

2.1. Explicación

El Algoritmo busca todas las formas de armar la ronda utilizando la técnica de backtracking. Para esto, el algoritmo recorre la matriz de relaciones uniendo a las chicas hasta que:

- forma la ronda (encuentra una combinación posible), o
- no puede armar la ronda de esa forma, y

saca la última chica que puso y vuelve a intentar formar la ronda poniendo otra amiga. Termina cuando encuentra una forma de armar la ronda o cuando prueba todas las posibles formas de armarla.

Además, el algoritmo utiliza algunas propiedades de la ronda para ser más eficiente:

2.1.1. Optimizaciones

- Verifica que cada chica tenga al menos dos amigas, de no ser asi podemos afirmar que no se puede formar la ronda ya que cada chica debe estar tomada de la mano de dos de sus amigas.
- A la vez, comprueba si todas son amigas de todas. Si eso sucede podemos afirmar que la ronda existe.
- Por otro lado, detecta si existen grupos independientes, es decir, sin conexiones entre sí. Si esto ocurre podemos afirmar que no se puede armar la ronda ya que esta debe incluirlas a todas.

2.1.2. Análisis de complejidad

Elegimos el modelo uniforme porque consideramos que las operaciones elementales son constantes porque todos los valores son acotados.

Sea n la cantidad de chicas.

El peor caso para el algoritmo es aplicar backtraking sobre una instancia *no_ronda* con una cantidad maximal de relaciones.

Para analizar la complejidad tomamos como peor caso, la instancia donde las relaciones sean máximas (todas con todas, cada chica tiene $n-1$ amigas) y el árbol se genere completamente sólo podando cuando se repite una amiga (nunca va a pasar por las optimizaciones y porque el algoritmo determinaria que se puede formar la ronda completando sólo la primer rama). En este caso se toma una chica y se generan todas las combinaciones posibles (sin repetir chicas). Para esto se elige la primera de donde se desprenden $n-1$ ramas posibles, de cada una de estas ramas se desprenden $n-2$, asi hasta llegar al nivel n.

Se ve que:

$$\prod_{i=1}^{n}(n-i) = (n-1)! \leq n!$$

Entonces la complejidad es: $O(n!)$.

En función del tamaño de la entrada

El tamaño de la entrada t es n^2 ($n = \sqrt{t}$). Entonces la complejidad del algoritmo es $O((\sqrt{t})!)$. El algoritmo es exponencial.

2.2. Detalles de la implementación

Elegimos arbitrariamente empezar la ronda por la chica *uno* (la primera segun el archivo de entrada) ya que a los efectos de verificar si es posible armar la ronda esta elección no tiene relevancia alguna.

Almacenamos las relaciones entre las chicas en una matriz de $n \times n$, donde n es la cantidad de chicas. Cada posición (i, j) de la matriz contiene un *uno*

si la chica i es amiga de j y un *cero* en caso contrario.

```
ronda_de_amigas(relaciones)
    if no_todas_tienen_al_menos_dos_amigas(relaciones)  or hay_más_de_un_grupo(relaciones)
        return false
    if todas_amigas_de_todas(relaciones)
        return true
    backtracking( relaciones )
```

- **no_todas_tienen_al_menos_dos_amigas:** para cada chica c el algoritmo inicializa un contador de amigas en cero y recorre todas las chicas preguntando si son amigas de c, si es asi incrementa dicho contador. Al finalizar el recorrido verifica que el contador sea mayor o igual a dos, es decir, que la chica c tenga al menos dos amigas. Si no es asi el algoritmo termina y devuelve falso.

- **hay_más_de_un_grupo:** para determinar si existe más de un grupo el algoritmo corre un bfs a partir de la primer chica (la primera segun el archivo de entrada). El algoritmo busca todas las amigas que todavia no hayan sido vistas considerandolas del mismo grupo. Repite este paso para cada una de las chicas alcanzadas (amigas de alguna anterior). Si el algoritmo termina y hay chicas que no fueron alcanzadas se las considera de otro grupo y por lo tanto la ronda no va a poder formarse.

  La complejidad es n^2, donde n es la cantidad de chicas. Porque el peor caso es cuando hay sólo un grupo ya que el algoritmo alcanza todas las chicas y para cada una de estas busca entre todas las chicas sus amigas.

- **todas_amigas_de_todas:** a la vez que determina si existe una chica que tiene menos de dos amigas (no_todas_tienen_al_menos_dos_amigas) utiliza ese contador para saber si cada chica tiene $n-1$ amigas, es decir, es amiga de todas las demas. Si es asi, el algoritmo termina y devuelve verdadero.

El algoritmo de backtracking recorre las chicas, para cada una de estas chequea si es amiga de la última chica que se agrego a la ronda y si todavia

no pertenece a la misma. Si es asi la agrega y repite este procedimiento (avanza). Sino significa que recorrió todas las chicas y ninguna cumple ambas condiciones por lo que comienza a retroceder.

Cuando retrocede, saca la última chica que agrego a la ronda (la cual identificaremos con la letra a, además llamamos b a la actual última chica en la ronda (la anterior a la que sacó)) y prosigue la busqueda desde la chica a de la amiga de b que ocupara la posición recientemente desocupada en la ronda. Si no hay una chica que puede ocuparla, es decir, b no tiene más amigas el algoritmo sigue retrocediendo.

Avanzando, si llega a meter a todas las chicas a la ronda y la primer chica es amiga de la última, encontró una forma de armar la ronda, termina y devuelve verdadero.

Retrocediendo, si llega a la primer chica, termina y devuelve falso.

2.3. Pruebas y Resultados

3. Problema 3

Dada una lista de ingresos y otra de egresos que contienen los horarios de ingreso y egreso de cada uno de los programadores de una empresa respectivamente, determinar la mayor cantidad de programadores que estan simultáneamente dentro de la empresa.

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 parcial de cantidad de programadores en simultaneo.

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 costo constante.

Sea n la cantidad de programadores, j el indice dentro de la lista de ingresos y k el indice dentro de la lista de egresos.

```
programadores_en_simultaneo(ingresos, egresos)
    max, tmp, j, k ← 0
    while (j < n)                                          O(n)
        if (ingresos[j] ≤ egresos[k])
            tmp ← tmp + 1
            j ← j + 1
        else
            if (tmp > max)
                max ← tmp
            tmp ← tmp − 1
            k ← k + 1
    if (tmp > max)
        max ← 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, tambien 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 tenemos un costo constante de operaciones, que no modifican la complejidad en el análisis asintótico. La complejidad algorítmica entonces, es $\mathrm{O}(n)$.

En función del tamaño de la entrada

Como los elementos de ambas listas son de tamaño acotado, el tamaño de la entada es proporcional a la longitud de las listas (n). Entonces la complejidad del algoritmo es $\mathrm{O}(t)$, donde t es el tamaño de la entrada. El algoritmo es lineal.

3.2. Detalles de la implementación

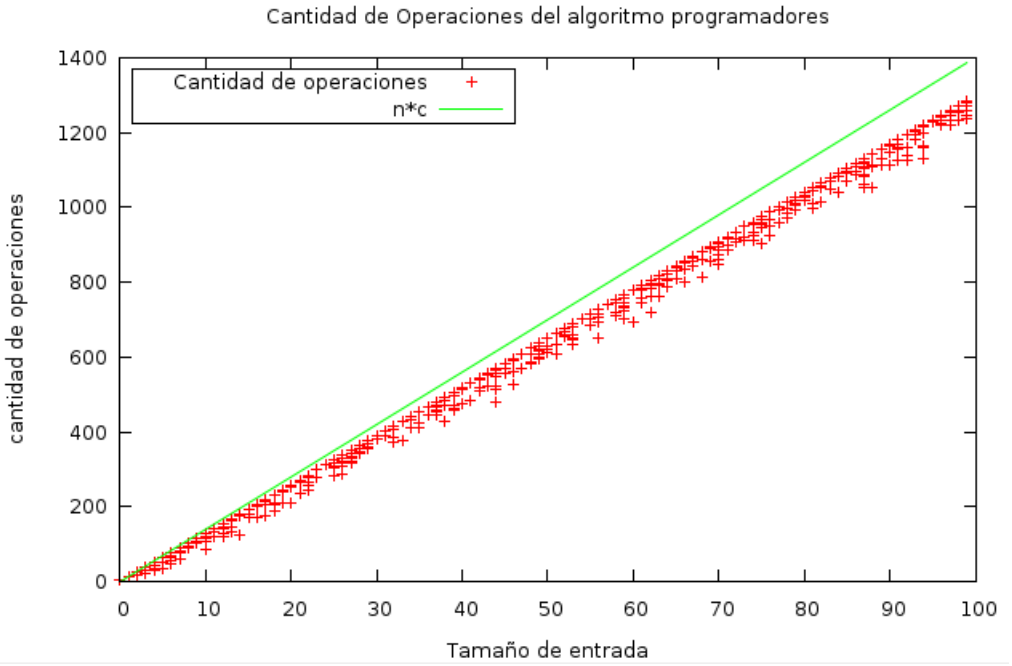
Guardamos los horarios de ingreso de todos los programadores (de la misma forma que estan en el archivo de entrada, es decir, en orden creciente) en un arreglo de *strings* (los cuales representan un horario en formato "HH:MM:SS") de tamaño n, donde n es la cantidad de programadores. Además guardamos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

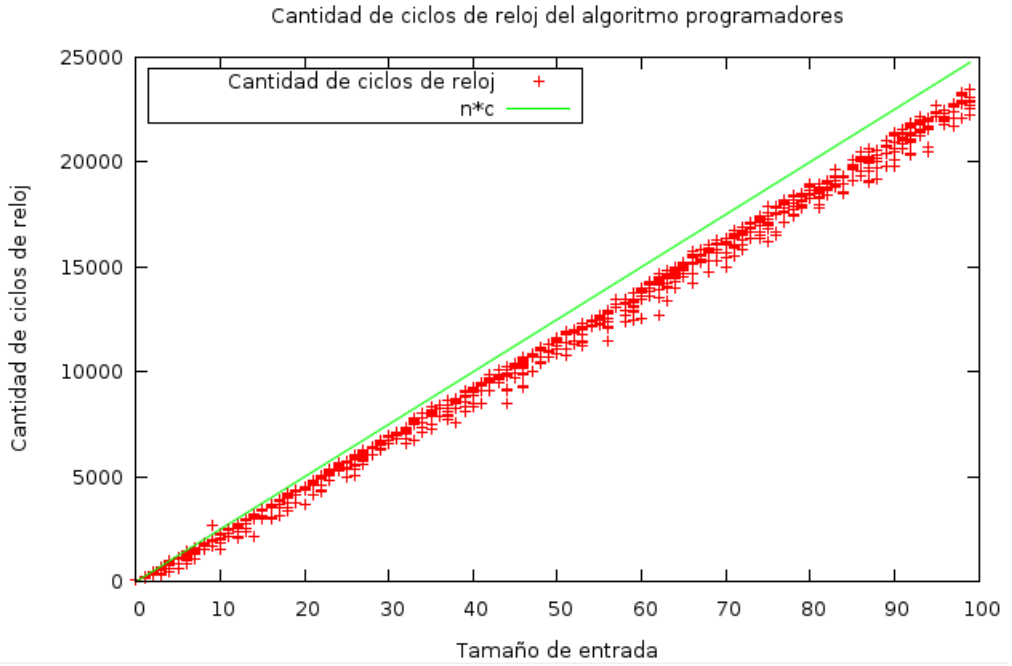
A medida que vamos recorriendo los arreglos *ingresos* y *egresos* necesitamos decidir si el horario de ingreso del programador j es anterior o posterior al horario de egreso del programador i, esto lo hacemos comparando los *strings* por menor o igual (que el horario de ingreso del programador j sea el mismo que el horario de egreso del programador i significa que ambos estuvieron en simultáneo en la empresa jústamente 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 estan simultáneamente dentro de la empresa.

3.3. Pruebas y Resultados

Para medir tiempos y cantidad de operaciones generamos instancias aleatorias con a lo sumo 100 programadores.


Cantidad de Operaciones del algoritmo programadores
Cantidad de operaciones
n*c
cantidad de operaciones
Tamaño de entrada
0
200
400
600
800
1000
1200
1400
0
10
20
30
40
50
60
70
80
90
100


Cantidad de ciclos de reloj del algoritmo programadores
Cantidad de ciclos de reloj
n*c
Cantidad de ciclos de reloj
Tamaño de entrada
0
5000
10000
15000
20000
25000
0
10
20
30
40
50
60
70
80
90
100

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` ses 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