Algoritmos y estructuras de datos III

Trabajo Práctico Nº1

Carla Livorno	424/08	carlalivorno@hotmail.com
Daniel Grosso	694/08	dgrosso@gmail.com
Diego Raffo	423/08	enanodr@hotmail.com
Mariano De Sousa Bispo	389/08	marian_sabianaa@hotmail.com

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 sería 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 cuando el exponente sea 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 procedimiento mencionado 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 que disminuyen la performance del algoritmo, y la posibilidad de *stack overflow*.

La versión iterativa de la solución funciona de la siguiente manera: en cada iteración el algoritmo divide el exponente a la mitad y eleva al cuadrado b. Cuando llega a un exponente impar, multiplica el resultado parcial (en la primer iteración 1) por b. Cuando el exponente es 0, termina, siendo el resultado parcial el valor del resultado final. Es decir, el resultado es la productoria de los resultados parciales obtenidos en los exponentes impares. Al acumular en b las potencias calculadas el algoritmo evita repetir cálculos y logra disminuir 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).

1.2. Detalles de la implementación

- La función recibe como parámetros por copia n y b.
- El resultado parcial de la función se guarda en la variable *tmp*, inicializada en 1.
- **Caso base:**
  - Cuando n es menor que 3, entra al caso base $b < 2$ ya que al comienzo la función toma $b = b \ mod \ n$ ($b \ mod \ 1 = 0$ y $b \ mod \ 2$ es *cero* si b es par y *uno* si es impar)
  - Cuando b es *uno* o *cero*, entra al caso base $b < 2$ y el resultado es b (porque $1^n \ mod \ n = 1$ y $0^n \ mod \ n = 0$).
- Antes de iniciar el ciclo, se crea una copia (m) de n para preservar su valor original.
- **Ciclo:**
  - **Caso condicional:** si m es impar multiplicamos *tmp* por el valor de b.
  - **En cada iteración:**
    - Se divide a la mitad m tomando la parte entera.
    - Se eleva al cuadrado b.
  - Luego de cada multiplicación toma módulo n ya que

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

    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).

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

1.3. Análisis de complejidad

Elegimos el modelo logarítmico para analizar el algoritmo, ya que las operaciones que aplicamos dependen del logaritmo de n, 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 $m = n$ y tmp inicializado en 1.

Como $b \leq n$ y la función logaritmo es estrictamente creciente, $\log b \leq \log n$. De la misma manera, $\log tmp$ está acotado por $\log n$.

Sean las siguientes complejidades en el modelo logarítmico:

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

Analizaremos a continuación la cantidad de operaciones, complejidad en el modelo logarítmico y complejidad en función del tamaño de la entrada.

1.3.1. Cantidad de operaciones

Para determinar la cantidad de operaciones que realiza el algoritmo analizamos el valor m en cada iteración debido a que la cantidad de iteraciones del ciclo depende de cuántas veces sea necesario dividir a m a la mitad para que sea igual a 0.

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

Como el ciclo termina cuando $m = 0$, en la última iteración $m = 1$. Por lo tanto, hallamos k tal que $m = 1$ para así obtener la cantidad total de iteraciones:

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

es decir, el algoritmo hace $\log(n) + 1$ iteraciones, cada una con una cantidad de operaciones acotadas por una constante c. La cantidad máxima de operaciones se da en las iteraciones en las que m es impar porque entra al caso condicional. Luego, la cantidad de operaciones que hace el algoritmo es $c * ((\log n) + 1)$. Por lo tanto, la cantidad de operaciones es del orden de $\log n$ y $\mathrm{O}(\log n) \subset \mathrm{O}(n)$.

1.3.2. Complejidad en el modelo logarítmico

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,

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

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

1.3.3. Complejidad 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: $O(\log^3 2^t)$=$O(t^3)$. El algoritmo es polinomial.

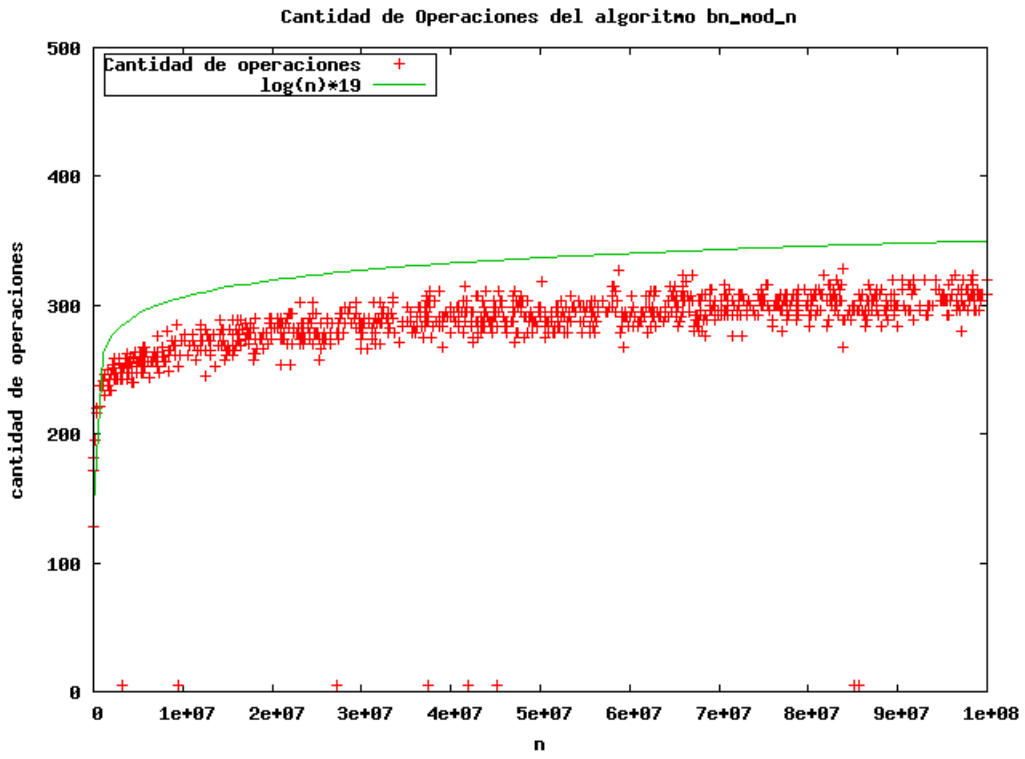
1.4. Pruebas y Resultados

Para probar correctitud tenemos un generador de instancias random que nos devuelve dos archivos, en uno `test.in` la instancia elegida (b n) y otro `test.out` el resultado ($b^n \mod n$). De esta forma, con una comparación de archivos (comando `diff test.out bn_mod_n.out`) podemos saber para cada instancia si el resultado obtenido por nuestro algoritmo se condice con el resultado correspondiente a esa instancia en el archivo `test.out`.

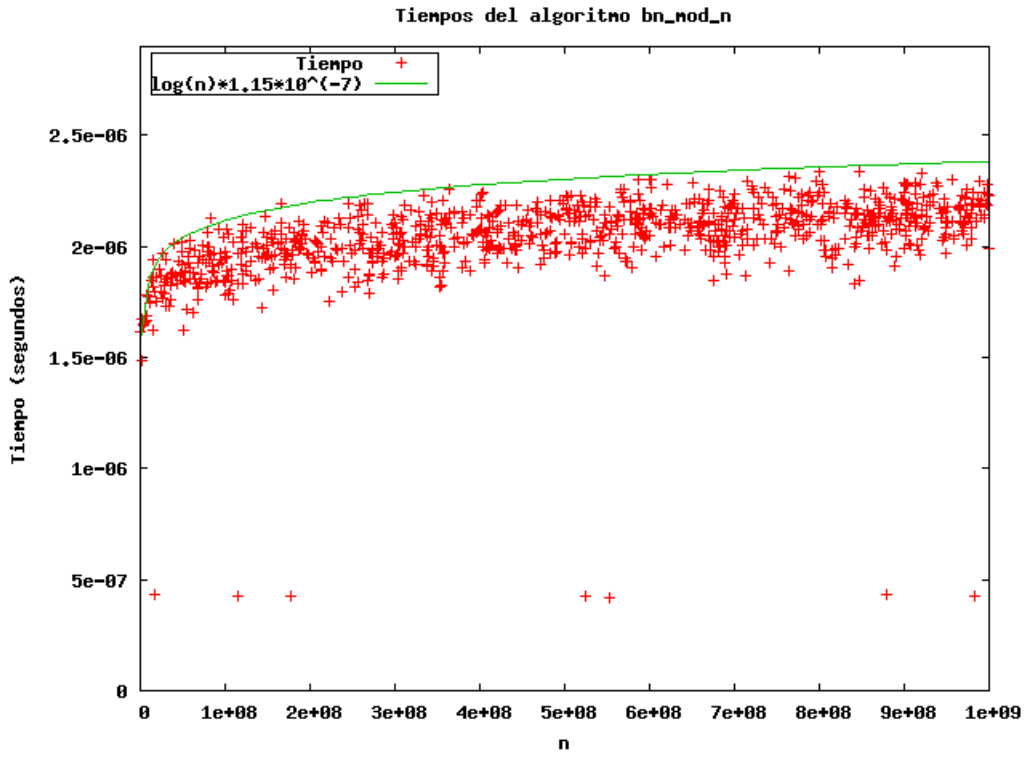
Para llevar a cabo las pruebas de este algoritmo en cuanto a operaciones realizadas por instancia, y tiempo en segundos transcurridos, generamos números en forma aleatoria, con b comprendido entre 0 y 200 (porque b está acotado), y n entre 1 y 10^7 (ya que esta cantidad de instancias posibles permite contrastar el comportamiento del algoritmo con los resultados teóricos).

En los siguientes gráficos cada instancia está representada con $+$. También está representada en color verde la función $log(n) * c$, ya que es la cota teórica previamente calculada con una constante aproximada calculada empíricamente para cada gráfico. A continuación se muestra la cantidad de

operaciones realizadas en función de n (donde c es 19).



El siguiente gráfico muestra el tiempo transcurrido (en segundos) en función de n (donde c es $1{,}15 * 10^{-7}$)



En los gráficos anteriores se ve cómo la cantidad de operaciones y el tiempo que tardó en ejecutar cada instancia están acotados por la función logaritmo (multiplicada por la constante correspondiente en cada caso), con n como parámetro. Los casos cercanos al cero en los ejes que representan al tiempo y la cantidad de operaciones, corresponden al caso trivial del ejercicio (caso base), en los que no hubo necesidad de iterar para llegar al resultado.

Como medir el tiempo no es una herramienta muy precisa, se pueden ver algunos *outliers* por encima de la curva del logaritmo. Aun así, los resultados coinciden con las predicciones en cuanto a la complejidad.

En el siguiente gráfico se muestran instancias en las que en cada iteración entra al caso condicional junto con las instancias que en ninguna de las

iteraciones entra al caso condicional. Las instancias fueron generadas de la siguiente manera:

- En ambos casos se tomó $2 < b < 200$ aleatoriamente. El valor máximo es 200 ya que b está acotado y el valor mínimo fue elegido para evitar generar casos base y así poder ver únicamente los casos en cuestión.

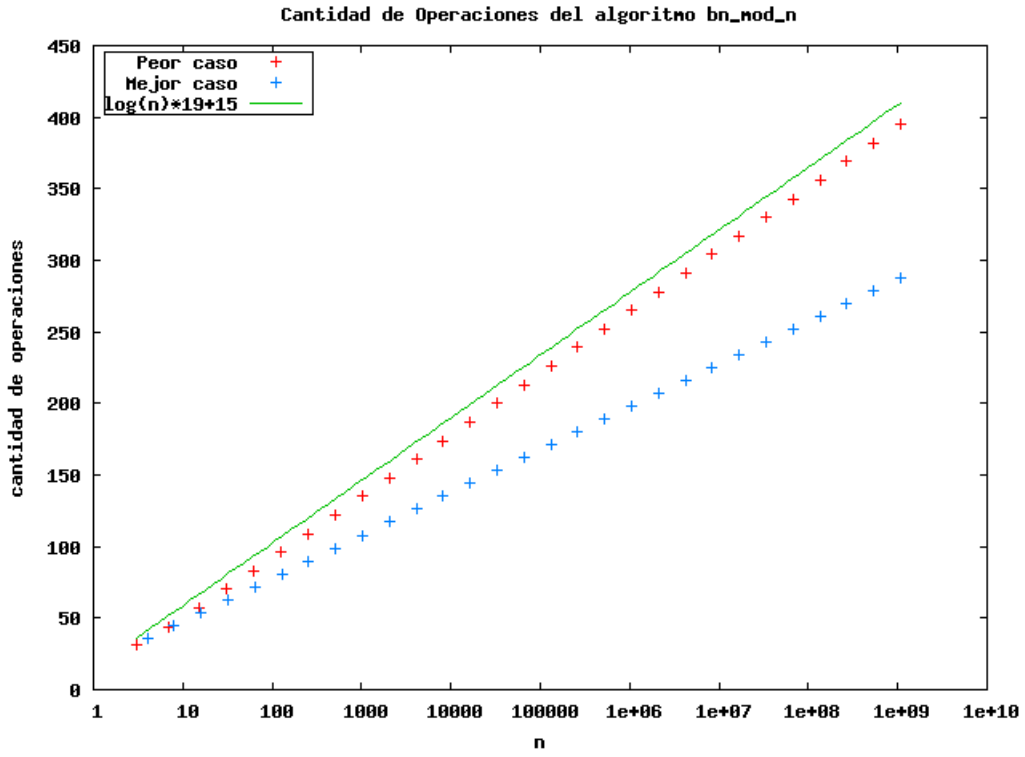
- Para las instancias que entran siempre al caso condicional se eligieron los n dados por $2^i - 1 \quad \forall\ i \in \mathbb{N}, 2 < i < 31$ debido a que al hacer la división entera de n por 2, siempre se obtiene un nuevo n impar. Esto hace que sea el peor caso. Elegimos 31 como cota superior para i porque es el máximo valor tal que $2^{31} \times 2^{31}$ entra en la variable y 2 como cota inferior para evitar los casos bases.

- Para las instancias que nunca entran al caso condicional se eligieron los n dados por $2^i \quad \forall\ i \in \mathbb{N}, 2 < i < 31$ debido a que al hacer la división entera de n por 2, siempre se obtiene un nuevo n par. Esto hace que sea el mejor caso excluyendo los casos base. La cota para i fue elegida con el mismo criterio que para generar los n impares.

Con esta prueba se espera poder contrastar ambos casos. Tanto el mejor como el peor caso difieren en una constante que se desprecia en la complejidad teórica pero que en la práctica puede significar una diferencia de rendimiento importante a medida que n crece.

El eje del gráfico que corresponde al valor de n está en escala logarítmica para poder apreciar mejor el gráfico dado que los valores de n tomados crecen exponencialmente. Cada instancia que entra siempre al caso condicional está representada con $+$. A su vez, las instancias que nunca entran al caso condicional están representadas con $+$. También está graficada en color verde la función $log(n) * 19 + 15$, ya que es la cota teórica previamente calculada

con una constante aproximada calculada empíricamente.



Observamos en el gráfico que para cada n impar elegido, hay un n par consecutivo. Cuanto mayor es n, mayor es la diferencia entre la cantidad de operaciones entre esas dos instancias. Esto se debe a que dichos n impares corresponden al peor caso del algoritmo y los pares al mejor. Asintóticamente, la diferencia entre mejor y peor caso resulta significativa a la hora del procesamiento.

`Observación:` Notamos que los resultados correspondientes a las instancias que nuncan entran al caso condicional son siempre *uno* o *cero*. Más específicamente el resultado es siempre *cero* para una instancia con b par y

siempre *uno* para una instancia con b impar.

1.5. Conclusiones

En este ejercicio nos encontramos con un problema donde debíamos operar valores enteros muy grandes, esto hizo que no pudiésemos considerar constante el costo de las operaciones elementales. Tuvimos así que analizar la complejidad en el modelo logarítmico. Al analizarla, tuvimos que diferenciarla en función de n y del tamaño de entrada (lo que se hizo en todos los ejercicios), con la particularidad de que en función de n resultó logarítmico, y en función del tamaño de la entrada, polinomial.

Las mediciones de tiempo y cantidad de operaciones se correspondieron con las complejidades teóricas calculadas. Esto indica que el modelo elegido fue adecuado.

Además, pudimos observar que en este caso la limitación del algoritmo no está dada por el tiempo de ejecución sino por el rango de valores admitidos ya que a partir de cierto n hay que trabajar con números más grandes de lo que se puede almacenar en la variable. Dicho esto, nos parece un buen algoritmo porque como cualquier otro que opera con variables numéricas en algún punto hace $overflow$ para una entrada 'suficientemente grande', y este puede devolver un resultado para todas las instancias -donde esto no ocurre- con un costo temporal muy bajo.

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 contenga 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 une 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 así 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. Dicho de otra manera, detecta si existen al menos dos chicas que no pueden ser unidas ya sea directamente o pasando por otras chicas. Si esto ocurre podemos afirmar que no se puede armar la ronda ya que esta debe incluirlas a todas.

2.2. Detalles de la implementación

Elegimos arbitrariamente empezar la ronda por la chica *uno* (la primera según el archivo de entrada) ya que a los efectos de verificar si es posible armar la ronda esta elección no tiene relevancia alguna. Esto es porque la equivalencia entre *dos* rondas no esta dada por la chica desde la que se inicie, sino por el hecho de que cada chica este de la mano de las mismas *dos* amigas.

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 así 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 así 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 según el archivo de entrada). El algoritmo busca todas las amigas que todavía no hayan sido vistas considerándolas 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 (ya que las relaciones entre las chicas están implementadas con una matriz), 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 demás. Si es así, 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 todavía no pertenece a la misma. Si es así 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 búsqueda 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. Análisis de complejidad

Elegimos el modelo uniforme porque consideramos que las operaciones elementales son constantes ya que 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, o dicho de otra manera, una instancia en que la cantidad de relaciones es tal que al agregar una relación más a cualquier chica, se forma una ronda. Es el peor caso ya que

no se resuelve por las optimizaciones y debe probar todas las combinaciones con las podas correspondientes.

Dado que la cantidad de ramas que se desprenden en cada nivel del árbol (teórico) no es constante, es decir, depende de la chica que el algoritmo agrega, encontrar una cota ajustada para la complejidad de este algoritmo es complicado. Por este motivo, para analizar la complejidad tomamos como peor caso un caso hipotético (sabemos que va a ser resuelto por las optimizaciones pero nos sirve para este análisis). Dicho caso es la instancia donde las relaciones son máximas (todas con todas, cada chica tiene $n-1$ amigas) y el árbol se genera completamente sólo podando cuando se repite una chica. 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 $n-2$, así sucesivamente hasta llegar al nivel n.

De lo anterior se deduce que la cantidad de combinaciones es:

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

Entonces la complejidad en el modelo uniforme es: $O(n!)$.

En función del tamaño de la entrada

Sea t el tamaño de la entrada, n la cantidad de chicas, i la cantidad de relaciones de cada una de ellas y j la chica con la cual se relaciona, entonces:

$t = \log n + \sum_{i=1}^{n}(\log i + \sum_{j=1}^{i} \log j)$

Dado que un valor ocupa como mínimo *un* bit se puede acotar inferiormente $\log n$, $\log i$ y log j por *uno*.

Entonces,

$t \geq 1 + \sum_{i=1}^{n}(1 + \sum_{j=1}^{i} 1) = 1 + n + n * i > n.$

Es decir, t es $\Omega(n)$. Entonces la complejidad del algoritmo es $O(t!)$. El algoritmo es factorial.

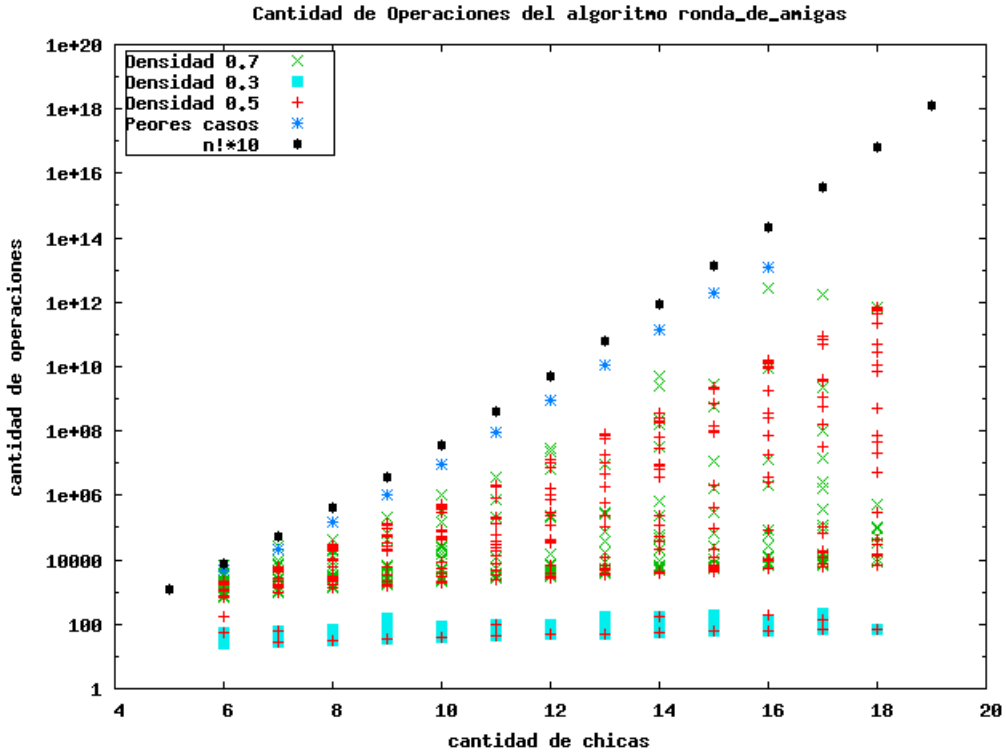
2.4. Pruebas y Resultados

Las entradas para probar correctitud están en los archivo `rondas.in`, `no_rondas.in`, `alguna_menos_dos_amigas.in`, `todas_con_todas.in`, `mas_de_un_grupo.in`. Fueron elegidas para probar el comportamiento del algoritmo en todas sus variantes y probar la eficiente detección de las 'no_rondas' cuando hay alguna chica con menos de dos amigas o cuando hay grupos aislados y de las 'rondas' cuando todas son amigas de todas.

Para analizar la complejidad temporal del algoritmo (tiempo y cantidad de operaciones) contamos con un generador aleatorio de rondas, donde para cada número de chicas n variamos la densidad de las relaciones (que es el porcentaje de la cantidad máxima de relaciones posibles que se incluye en la instancia generada). Generamos 20 casos por cada n desde 6 a 18 chicas, variando la densidad entre 0.3, 0.5 y 0.7.

Se hizo énfasis en la generación de casos promedio y peores casos (donde el algoritmo no resuelve el problema mediante las optimizaciones). Para esto se tomo en cuenta las posibles *no_rondas*, con un máximo aproximado (calculado empíricamente) de relaciones tal que al agregarle una relación más a cualquier chica, se pueda formar una ronda. También forzamos a que en sólo dos instancias por cada n hubiera chicas sin amigas o con sólo una amiga, donde el algoritmo optimizado se ahorra el intentar construir soluciones.

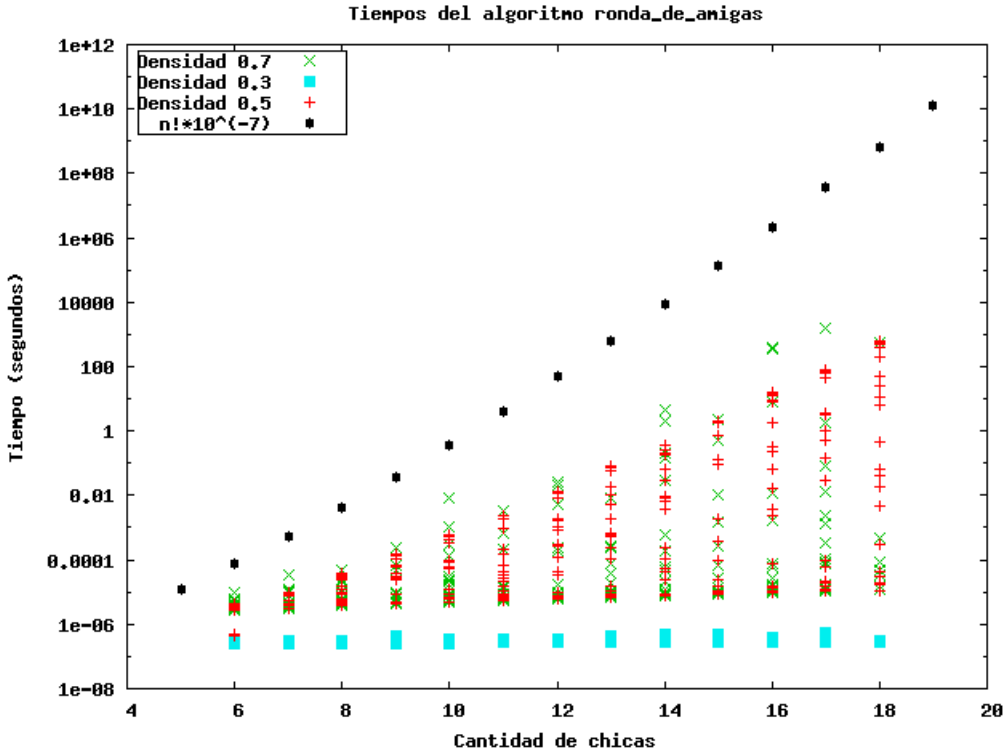
En los siguientes gráficos las instancias con densidad 0.3 están representadas con $\blacksquare$, las instancias con densidad 0.5 con $+$, las que tienen densidad 0.7 con $\times$ y los peores casos con $*$. Además, graficamos la función $c*n!$ con $\bullet$, ya que es la cota teórica previamente calculada con una constante aproximada calculada empíricamente para cada gráfico. Los gráficos están en escala logarítmica para poder apreciar mejor la función.



En este gráfico se muestra la cantidad de operaciones en función del tamaño de la entrada, es decir, en función de la cantidad de chicas y la cantidad de relaciones entre ellas.

Ovservamos que para un n fijo, la función crece (la cantidad de operaciones aumenta) conforme aumenta la cantidad de relaciones entre las chicas.

Dicha función se ve acotada por la función factorial, cuyo parámetro de entrada es la cantidad de chicas n. Vemos también que los casos que considerábamos como los peores aparecen por encima del resto de las instancias. Los casos cercanos al cero en el eje cantidad de operaciones representan las instancias donde las optimizaciones del algoritmo no permitieron recorrer inútilmente el árbol para poder concluir que no hay solución posible.



En este se grafica el tiempo transcurrido para cada instancia. La apreciación también es muy similar, ya que nuevamente aparece acotada por la función factorial, y se ven más abajo los casos donde se dieron lugar a las optimizaciones.

En general, podemos ver que la complejidad teórica analizada resulta ser una cota menos ajustada conforme crece n.

En ambos gráficos se puede observar que en el eje de cantidad de amigas para una entrada de *dieciocho*, existen menos puntos de densidad 0,7. Realizamos menos casos porque el costo (en tiempo) de procesamiento del algoritmo era del orden de horas para cada instancia. Tampoco se realizó la prueba de los peores casos correspondientes a n igual a *diecisiete* y *dieciocho* en cuanto a cantidad de operaciones por lo dicho anteriormente. Igualmente predecimos que de hacerse los casos y graficarlos, los puntos coincidirían con el comportamiento esperado.

Al generar las rondas de forma aleatoria, para valores menores que seis -con las densidades propuestas-, nuestro algoritmo de generación de rondas no podía realizar la tarea. Por lo tanto, esos casos no fueron incluídos.

2.5. Conclusiones

En este problema nos encontramos con un algoritmo de complejidad factorial que a partir de cierto tamaño de entrada 'relativamente chico' se torna inútil, ya que la ejecución para dichas instancias es extremadamente lenta y en la práctica muchas veces no es aceptable. Esto significa que el problema no está 'bien' resuelto para tamaños de entrada no necesariamente grandes, notándolo en las pruebas: a partir de 16 chicas en casos 'malos' el algoritmo requería de varias horas para devolver una respuesta.

Con respecto a las mediciones de tiempo y cantidad de operaciones estas se correspondieron con las complejidades teóricas calculadas lo que indica que el modelo elegido fue adecuado.

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 están 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. Este comportamiento se ve reflejado en el siguiente pseudocódigo:

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

```
programadores_en_simultaneo(ingresos, egresos)
    max, tmp, j, k ← 0
    while (j < 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
```

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. 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.

Como cada programador tiene un ingreso y un egreso, tanto la lista de ingresos como 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 tenemos un costo constante de operaciones, que no modifican la complejidad en el análisis asintótico. La complejidad algorítmica en el modelo uniforme es $O(n)$.

En función del tamaño de la entrada

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

3.4. Pruebas y Resultados

Las entradas utilizadas para probar correctitud están en el archivo `pruebas.in`. Fueron elegidas para probar casos bordes como son: un solo programador, primero ingresan todos y luego empiezan a egresar, un ingreso se produce exactamente al mismo horario que un egreso, cada ingreso seguido del egreso correspondiente y por último ingresos y egresos mezclados.

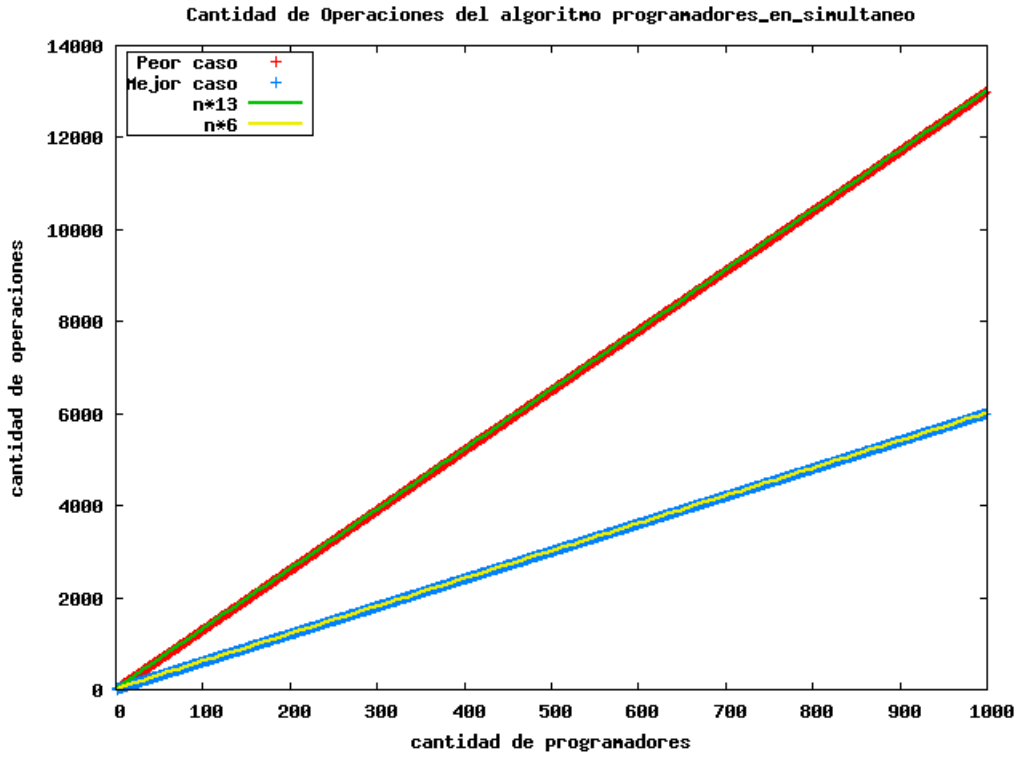
Por un lado, decidimos contar la cantidad de operaciones realizadas por el algoritmo en el mejor y pero caso. Para esto generamos las instancias de la siguiente manera:

- En ambos casos, elegimos un número máximo de programadores(m) ($m = 1000$ porque nos pareció suficiente para constrastar estos casos) y creamos una instancia $\forall\ n, 0 \leq n < m$ siendo n la cantidad de programadores para esa instancia.
- Para las instancias que corresponden al mejor caso ingresan todos los programadores y luego egresan.
- Para las instancias que corresponden al peor caso los programadores que entran salen antes de que haya un nuevo ingreso, es decir, nunca va a haber más de un programador en simultáneo.

El objetivo de esta prueba es comparar ambos casos. Uno de ellos es el mejor caso para el algoritmo y esta acotado inferiormente por $6 * n$ y el otro, el peor caso para el algoritmo se encuentra acotado superiormente por $13 * n$. A pesar de que el costo del algoritmo es siempre n, se espera ver diferencias significativas en la cantidad de operaciones realizadas en uno u otro caso ya que en el primero de ellos es mínima porque al producirse todos los ingresos el algoritmo termina sin recorrer la lista de egresos y en el segundo máxima ya que para poder registrar el último ingreso tuvo que recorrer todos los egresos.

Cada instancia correspondiente al peor caso se representa con $+$ mientras que cada instancia que corresponde al mejor caso está representada con $+$. También está graficada en color verde la función $13 * n$, ya que es la cota superior teórica previamente calculada y en amarillo la función $6 * n$ que es

la cota inferior calculada ambas con una constante calculada empíricamente.



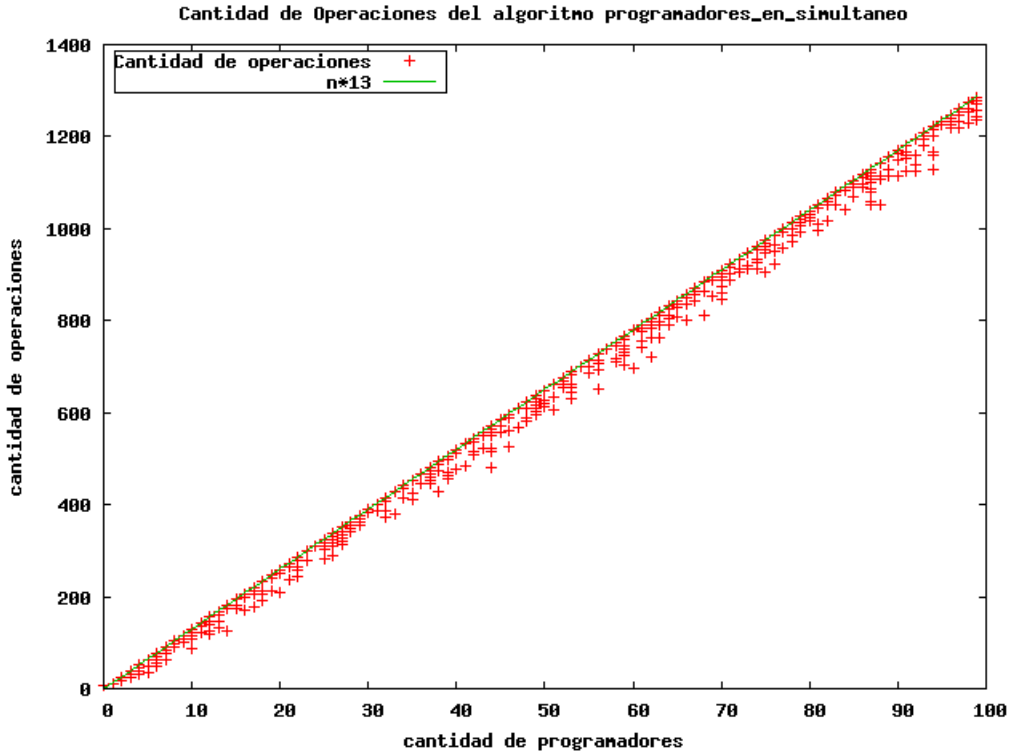
Observamos que se cumple lo esperado, es decir, corroboramos empíricamente que lo analizado previamente tiene coherencia, ya que la diferencia en la cantidad de operaciones que realiza el algoritmo para resolver un problema cuya característica es la del mejor caso con n programadores se hace cada vez más significativa respecto a las del peor caso para un mismo n a medida que dicho n crece.

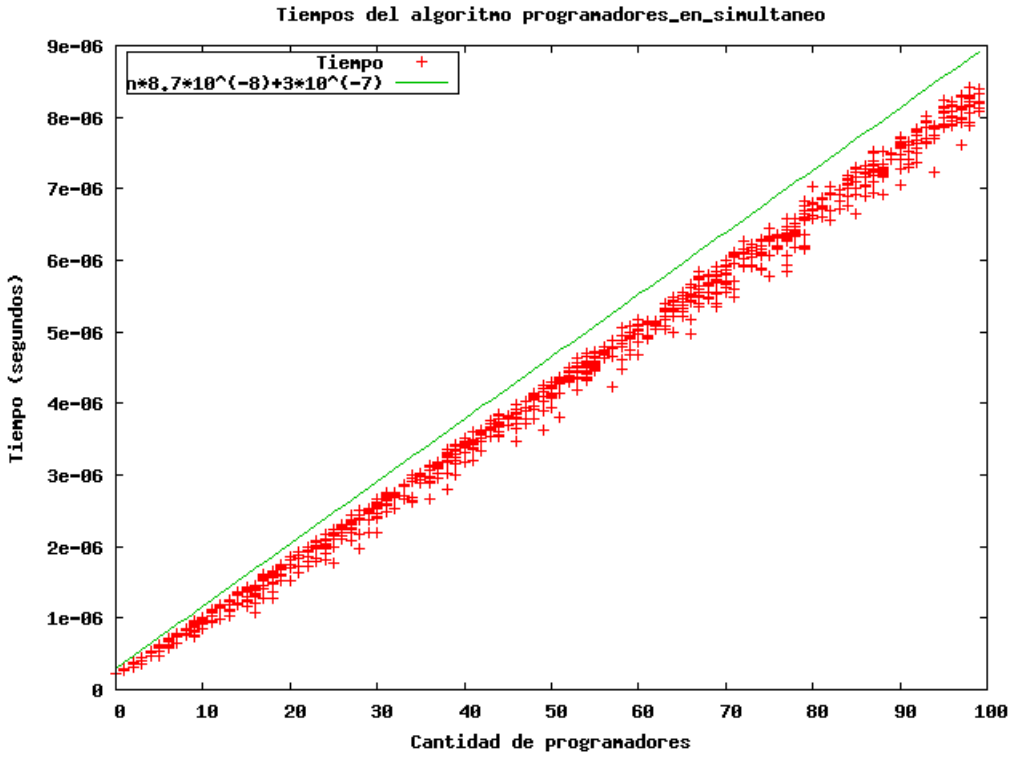
Para los gráficos que siguen utilizamos un generador, que dado un número máximo de programadores n, genera instancias aleatorias con a lo sumo n programadores. Corrimos el generador con n igual a *cien* dado que no necesitamos instancias demasiado grandes para poder apreciar el comportamiento

del algoritmo.

En los siguientes gráficos esperamos ver el comportamiento general del algoritmo en cuanto a cantidad de operaciones como en tiempo transcurrido por instancia.

Las instancias se representan con $+$, además en el gráfico de cantidad de operaciones aparece en color verde la función $13 * n$ mientras que en el de tiempos aparece graficada en el mismo color $8,7 * 10^{-8} * n + 3 * 10^{-7}$ por los motivos ya mencionados.





En el primer gráfico se ve la función que dado el número de programadores, muestra la cantidad de operaciones realizadas por el algoritmo mientras que en el segundo se observa el tiempo consumido para procesar cada instancia, ambos en función de la cantidad de programadores. En los dos gráficos se puede observar que la función se encuentra acotada por una recta, con lo cual se ve sin problemas que se trata de un algoritmo que tiene 'costo' lineal.

Lo que observamos en los gráficos tiene coherencia con la complejidad teórica calculada. En general, vemos que el algoritmo se comporta como $\Theta(n)$ ya que como mínimo recorre sólo toda la lista de ingresos y como máximo recorre tanto la lista de ingresos como la de egresos. Es decir, el costo del algoritmo esta acotado inferior y superiormente por $c * n$, con c constante.

3.5. Conclusiones

Este problema nos pareció particularmente sencillo desde el punto de la implementación. Pudimos resolverlo con un algoritmo simple que tiene costo lineal tanto en función del tamaño de la entrada como de la cantidad de programadores. Se considera entonces un problema 'bien' resuelto desde el punto de vista computacional.

Además, las mediciones de tiempo y cantidad de operaciones se correspondieron con las complejidades teóricas calculadas. Por lo que el modelo elegido fue adecuado.

4. Anexo

Correctitud ejercicio 1

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

$A_k = \frac{n}{2^{k-1}}$ la sucesión con los valores de n en la iteración k, y

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

entonces el cálculo hecho por el algoritmo está dado por

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

Luego de cada multiplicación toma módulo n ya que

$$b^k * b^{n-k} \ mod \ n = ((b^k \ mod \ n) * (b^{n-k} \ mod \ n)) \ mod \ n \ \forall \ k \leq n \text{[2]}$$

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] $par(x) = 1 - impar(x)$
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 $x * y \ mod \ z = ((x \ mod \ z) * (y \ mod \ z)) \ mod \ z$

5. Mediciones

- Para contar la cantidad aproximada de operaciones definimos una variable inicializada en *cero* la cual incrementamos luego de cada operación.
- Para medir tiempo tenemos una función que ejecuta el algoritmo por un mínimo de tiempo pasado como parametro, esto lo hacemos para obtener una mejor precisión. Una vez que se cumple el tiempo tenemos en una variable la cantidad de veces que se ejecutó el algoritmo, luego dividimos el tiempo medido por c.

6. 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`. Devuelve para cada instancia el tamaño seguido del tiempo trancurrido (en segundo) para procesar esa instancia.

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