

ALGORITMOS Y ESTRUCTURAS DE DATOS III

Trabajo Práctico N°1

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	2
1. Problema 1	2
1.1. Explicación	2
1.2. Detalles de la implementación	3
1.3. Análisis de complejidad	4
1.3.1. Cantidad de operaciones	4
1.3.2. Complejidad en el modelo logarítmico	5
1.3.3. Complejidad en función del tamaño de la entrada	5
1.4. Pruebas y Resultados	6
2. Problema 2	10
2.1. Explicación	10
2.1.1. Optimizaciones	10
2.1.2. Análisis de complejidad	10
2.2. Detalles de la implementación	11
2.3. Pruebas y Resultados	13
3. Problema 3	16
3.1. Explicación	16
3.1.1. Análisis de complejidad	17
3.2. Detalles de la implementación	18
3.3. Pruebas y Resultados	18
4. Compilación y ejecución de los programas	22
5. Conclusiones	22

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

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 b es menor que 2, entra en un caso base y el resultado es $1 \bmod n$ ya que $1^n \bmod n = 1 \forall n > 1$ y $1 \bmod 1 = 0$.
 - Cuando n es 1, entra al caso base $b < 2$ ya que al comienzo la función toma $b = b \bmod n$ y $b \bmod 1 = 0$.
- 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

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

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 $x * y \bmod z = ((x \bmod z) * (y \bmod z)) \bmod z$

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 $m = n$, tmp inicializado en 1 y las siguientes complejidades en el modelo logarítmico:

```

bn_mod_n(b, n)
  while m > 0
    if m es impar          O(log m)
      tmp ← tmp * b        O(log2 n)
      tmp ← tmp mod n      O(log2 n)
    m ←  $\frac{m}{2}$               O(log m)
    b ← b2                 O(log2 n)
    b ← b mod n            O(log2 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.

1er	iteración	→	$m = n$
2da	iteración	→	$m = \frac{n}{2}$
3er	iteración	→	$m = \frac{n}{2^2}$
4ta	iteración	→	$m = \frac{n}{2^3}$
	⋮		⋮
k -ésima	iteración	→	$m = \frac{n}{2^{k-1}} = 1$

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 de 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 $O(\log n) \subset O(n)$.

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:

$$\begin{aligned}O(\log(n) * (\log(n) + \log^2(n) + \log^2(n) + \log(n) + \log^2(n) + \log^2(n))) &= \\ = O(\log(n) * (2 \log(n) + 4 \log^2(n))) &= O(2 \log^2 n + 4 \log^3 n)\end{aligned}$$

Luego, por definición,

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

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

1.3.3. Complejidad en función del tamaño de la entrada

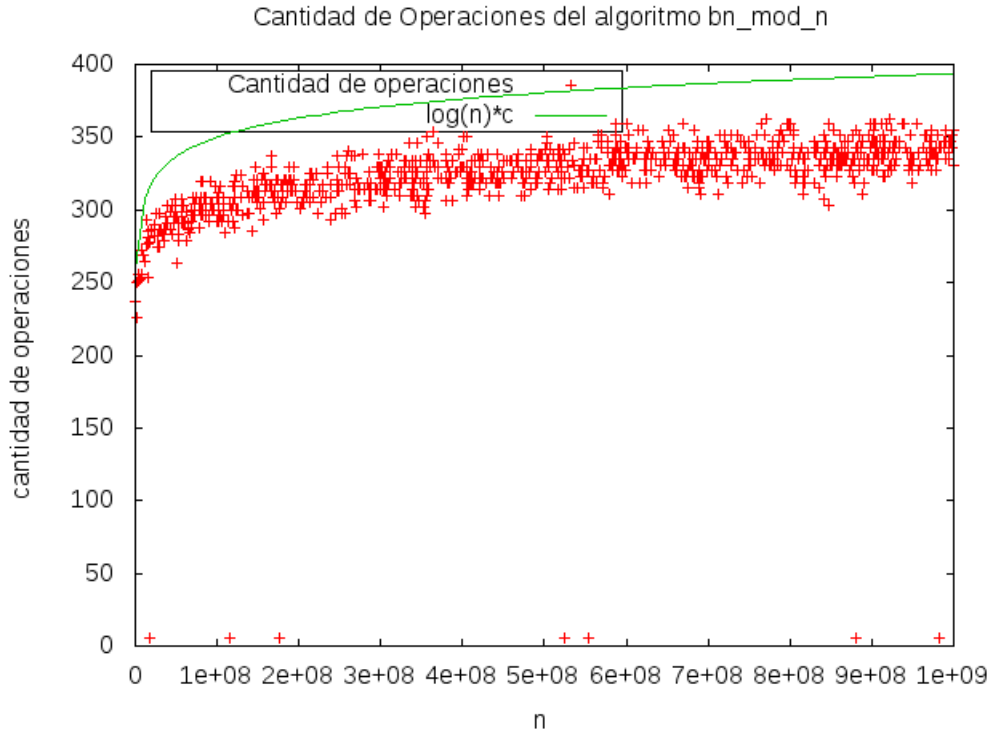
El tamaño de la entrada t es $\log n$ ya que b es acotado ($n = 2^t$). Entonces la complejidad del algoritmo en función del tamaño de entrada es: $O(\log^3 2^t) = O(t^3)$. El algoritmo es polinomial.

1.4. Pruebas y Resultados

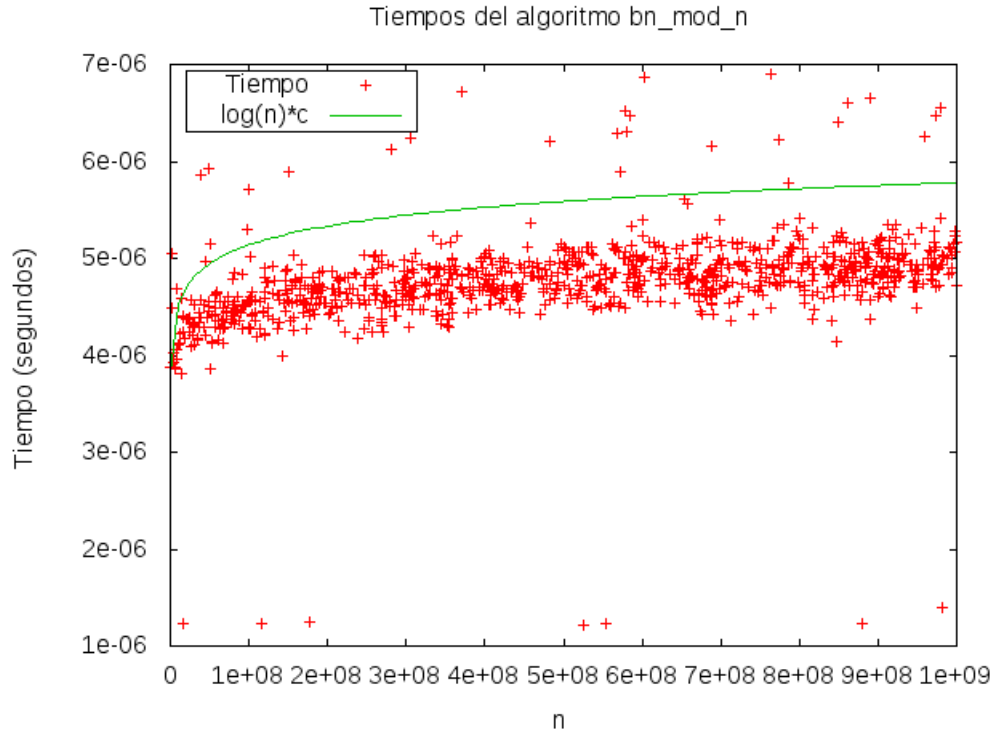
Para probar correctitud tenemos un generador de instancias random que nos devuelve dos archivos, en uno `test.in` la instancia elegida ($b \ n$) y otro `test.out` el resultado ($b^n \bmod n$). De esta forma, con una comparación de archivos (comando `diff test.out bn_mod_n.out`) podemos saber para cada 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 (esta cantidad de instancias posibles permite contrastar el comportamiento del algoritmo con los resultados teóricos).

En los siguientes gráficos cada instancia está representada con una cruz roja. 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 $2,79 * 10^{-7}$)



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.

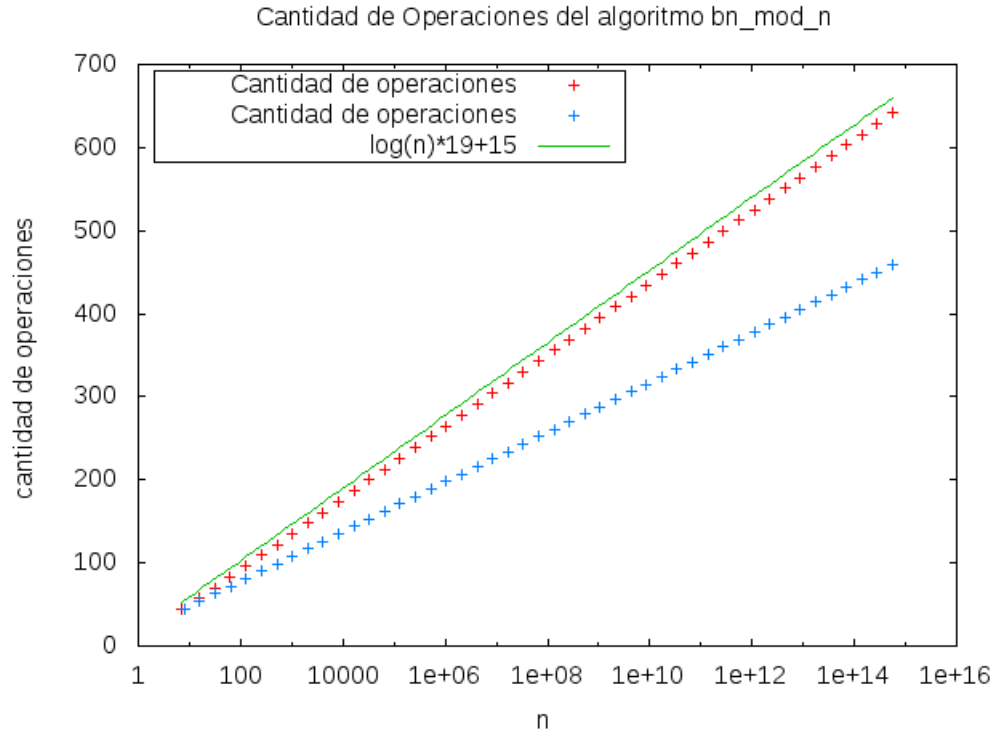
En el siguiente gráfico se muestran instancias en las que en cada iteración entra al caso condicional junto con las instancias que en ninguna de las iteraciones entra al caso condicional. Las instancias fueron generadas de la siguiente manera:

- En ambos casos se tomó $2 < b < 200$ aleatoriamente. El valor 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 $2^i - 1 \ \forall i \in \mathbb{N}, i < 50$ debido a que al hacer la división entera de n por 2, siempre se obtiene un nuevo n impar.
- Para las instancias que nunca entran al caso condicional se eligieron los n dados por $2^i \ \forall i \in \mathbb{N}, i < 50$ debido a que al hacer la división entera de n por 2, siempre se obtiene un nuevo n par.

El eje 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 una cruz roja. A su vez, las instancias que nunca entran al caso condicional están representadas con una cruz azul. 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. Con esta prueba se espera poder contrastar ambos casos, donde teóricamente hay solo una diferencia *cosntante*, pero que en la práctica nos brinda el mejor o peor caso para cierta cantidad de iteraciones.



Observamos que en el gráfico, cuanto mayor es n , mayor es la diferencia entre los valores de las muestras a pesar de que son consecutivos.

En general, podemos ver que la complejidad teórica analizada resulta ser una mejor cota conforme crece n . Por lo tanto, podemos afirmar que $\log n$ es una buena cota superior para la complejidad del algoritmo ya que los resultados prácticos coinciden con los cálculos teóricos.

2. Problema 2

Se tiene n chicas cada una de las cuales 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 recorre la matriz de relaciones uniéndolo 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. 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 backtracking 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 determinaría 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 $n - 2$, así 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 según 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 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 , 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 agregó a la ronda y si todavía

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

Cuando retrocede, saca la última chica que agregó 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.

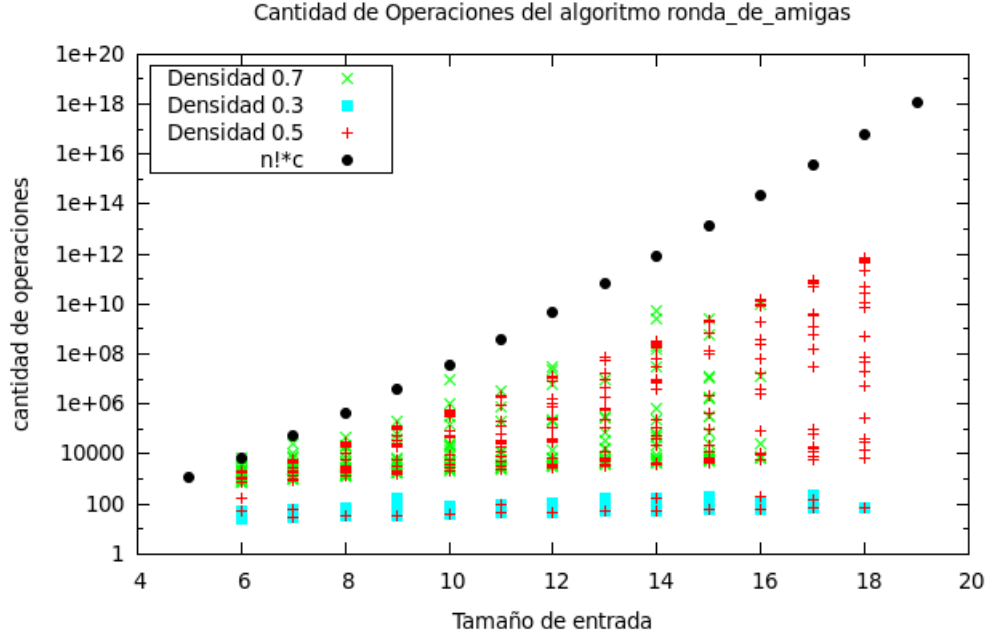
Retrocediendo, si llega a la primer chica, termina y devuelve falso.

2.3. Pruebas y Resultados

Las entradas para probar correctitud están en los archivos `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.

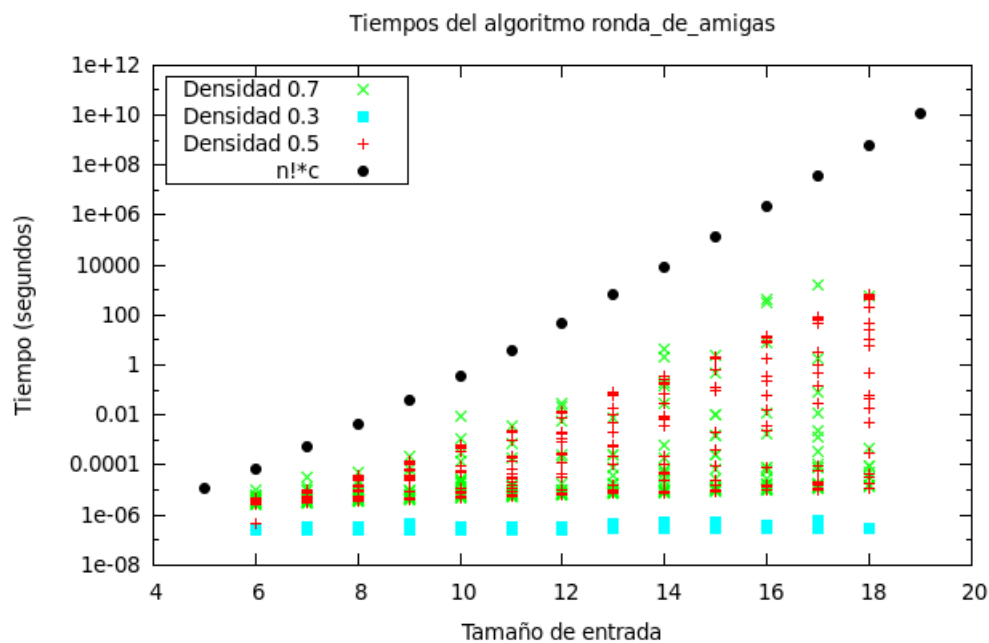
Se hizo énfasis en la generación de casos promedio y peores casos (donde el algoritmo no resuelve el problema mediante las optimizaciones). También forzamos a que en sólo dos instancias por cada n hubiera chicas sin amigas o con sólo una amiga, donde el algoritmo optimizado se ahorra el intentar construir soluciones.



En 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 parametro de entrada es la cantidad de chicas n . Se ven claramente en el piso del gráfico los casos en donde las optimizaciones del algoritmo no permitieron recorrer inutilmente 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.

Los gráficos están en escala logarítmica porque las funciones graficadas crecen extremadamente rápido y de esta manera se pueden apreciar mejor los valores.

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

Este es el pseudocódigo del algoritmo diseñado, que se explicará mas detalladamente a continuación.

```
programadores_en_simultaneo(ingresos, egresos)
(1)    $max, tmp, j, k \leftarrow 0$ 
(2)   while ( $j < n$ )
(3)       if ( $ingresos[j] \leq egresos[k]$ )
(4)            $tmp \leftarrow tmp + 1$ 
(5)            $j \leftarrow j + 1$ 
(6)       else
(7)           if ( $tmp > max$ )
(8)                $max \leftarrow tmp$ 
(9)            $tmp \leftarrow tmp - 1$ 
(10)           $k \leftarrow k + 1$ 
(11)   if ( $tmp > max$ )
(12)        $max \leftarrow tmp$ 
(13)   return  $max$ 
```

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 (lineas 2 y 3), 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(linea 4). 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(linea 7), de ser así, actualizamos el máximo.

Luego se descuenta el recientemente egresado del contador parcial de cantidad de programadores en simultaneo(linea 9).

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 índice dentro de la lista de ingresos y k el índice dentro de la lista de egresos.

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 tenemos un costo constante de operaciones, que no modifican la complejidad en el análisis asintótico. La complejidad algorítmica entonces, 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.

3.2. 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 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 estan simultáneamente dentro de la empresa.

3.3. Pruebas y Resultados

Las entradas utilizadas para probar correctitud estan 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 ultimo ingresos y egresos mezclados.

En los gráfico se verán, en rojo, las funciones que dado el número de programadores, muestran la cantidad de operaciones realizadas o el tiempo

transcurrido durante la ejecución del algoritmo para esa cantidad de programadores. Se busca con esto comprobar lo analizado anteriormente, que dicha función se pueda ver acotada por alguna recta, dibujada en color verde.

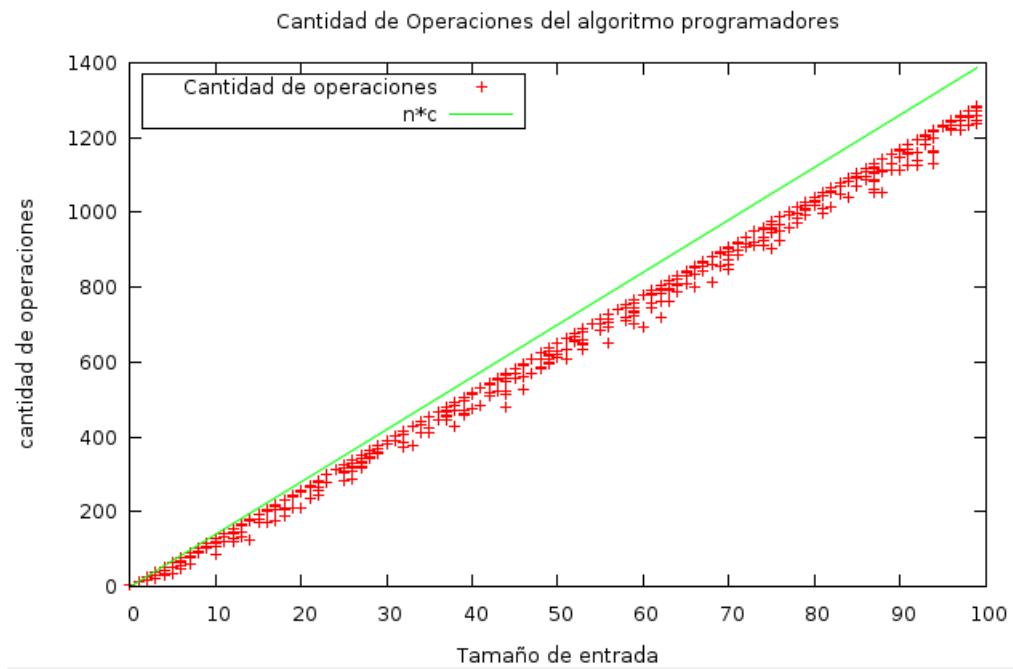
En los primeros dos gráficos se podrá ver comportamiento del algoritmo para un rango de 1 a n programadores que entran y salen constantemente, es decir, nunca habra mas de un programador en simultaneo. Para esto se cuenta con un generador que crea instancias de este tipo. La recta graficada se corresponde con Ax .

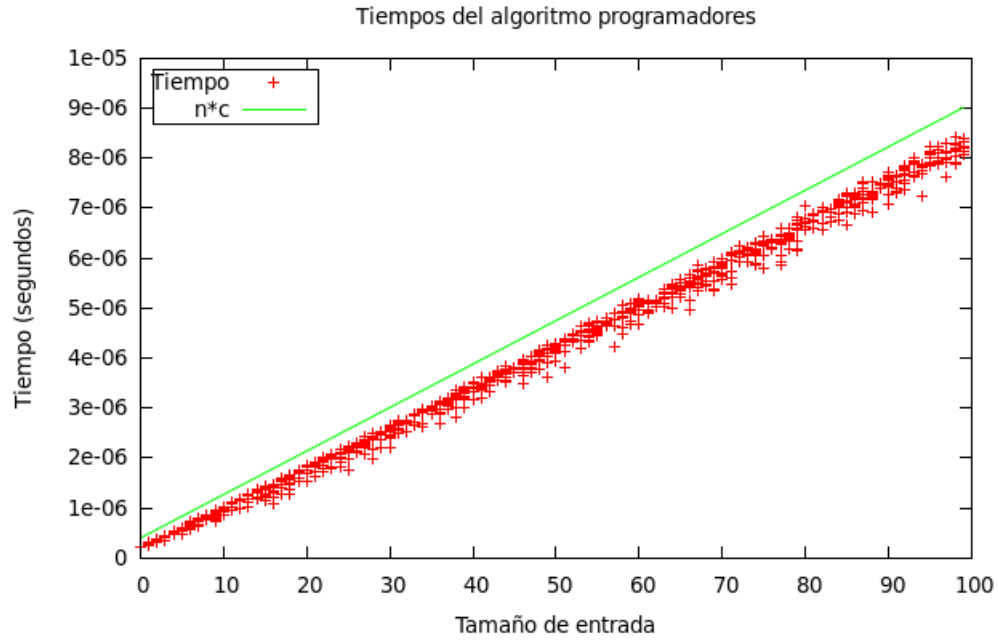
GRAFICO 1, GRAFICO 2

En los siguientes dos gráficos se mostraran casos donde primero ocurran todos ingresos y luego los egresos. Se espera ver lo que sería el peor caso, donde si bien la complejidad de mantendría lineal, se maximisa la cantidad de operaciones, ya que carla lo explicara ahora..... COMPLETAR. Para esto se usó un generador de instancias de este tipo, donde siempre el último en ingresar era el ultimo en irse.

GRAFICO 1, GRAFICO 2

Para los ultimos gráficos utilizamos un generador aleatorio de instancias, 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, esperando ver el comportamiento del algoritmo en los casos anteriores y casos promedio, y así poder cotrastarlos.





Se puede observar que ambos gráficos se encuentran acotadas por la rectas de los casos donde todos los ingresos precedían a los egresos. Con lo cual se corrobora empíricamente que el caso coincide con lo analizado previamente. En ambos gráficos, se observa que hay una distribución muy similar, sólo que en este se muestra el tiempo consumido para procesar cada instancia en función de la cantidad de programadores.

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 está acotado inferior y superiormente por $c * n$, con c constante.

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`. Devuelve para cada instancia el tamaño seguido de la cantidad de ciclos de reloj consumidos.

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

5. Conclusiones

Pudimos implementar las soluciones a los problemas propuestos utilizando diversas técnicas algorítmicas. En todos los casos las mediciones de tiempo y cantidad de operaciones se correspondieron con las complejidades teóricas calculadas. Esto indica que los modelos elegidos fueron adecuados.

En el problema *dos* nos encontramos con un algoritmo de complejidad exponencial 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 esto no es aceptable. Esto significa que el problema no está 'bien' resuelto para tamaños de entrada no necesariamente grandes.