Algoritmos y Estructuras de Datos III

Primer Cuatrimestre de 2007

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 1

Integrante	LU	Correo electrónico
Blanco, Matias	508/05	matiasblanco18@gmail.com
Freijo, Diego	4/05	giga.freijo@gmail.com
Giusto, Maximiliano	486/05	maxi.giusto@gmail.com

Palabras Clave

Complejidad algorítmica, modelo uniforme, modelo logarítmico.

Índice

1. Ejercicio 1

1.1. Introducción

Para comenzar a pensar este ejercicio, se descartó de lleno la opción de fuerza bruta. Se buscó una que podara la mayoría de los casos que no sirven. Donde se ahorran la mayoría de los casos es al encontrar un caso que no sirve, se siga buscando en ese mismo conjunto, un elemento más abajo.

Lo que se hizo en este algoritmo es primero colocar los primeros elementos de ambos conjuntos, lo cual esta garantizado que es la tupla más grande en valor. Luego, se elige arbitrariamente el primero de uno, con el segundo del otro. Se buscan los mayores a esta tupla, descartando al encontrar un caso menor, ya que si, por ejemplo A[i]+B[j] es menor al valor de referencia, A[i]+B[j+1] también lo será, por la precondición de que ambos conjuntos vienen ordenados. Luego de conseguir éstas sumas, se vuelve a buscar mayores, pero enéste caso con los próximos dos elementos del mísmo índice. Ésta vez se busca tanto las variaciones del conjunto B con el A, como las del A con el B. Luego, quedan generadas 3 listas con candidatos para ingresar. Se ordena ésta lista y se agregan a la lista final hasta que se llene. En caso contrario, que falten casos, se vuelve a implementar el procedimiento.

Para ordenar la lista, primero se tienen que ordenar las listas candidatas. Para lo primero se utilizó un Quick Sort por sobre el Selection Sort, ya que, aunque ambos tienen la misma complejidad en peor caso, el Quick Sort se comporta mucho mejor en casos promedio, como se puede ver en el gráfico de los resultados. Para el segundo, se utiliza un Merge Sort para crear una nueva lista ordenada desde las tres listas anteriores.

1.2. Detalles de implementación

Se creó la clase Datos para contener a los dos conjuntos iniciales y se creó la clase Instancia para contener ambos conjuntos y el n del tamaño.

Como detalle, se puede mencionar que el algoritmo se optimizó un poco más utilizando en algunos casos Linked Lists en vez de Array Lists, ya que las primeras tienen una complejidad de lectura constante y de inserción lineal. Ésto ahorró la lectura de los conjuntos iniciales, que se realiza en varias oportunidades. Los conjuntos, tanto temporales como el del resultado, siguen siendo ArrayLists, ya que en ellos se inserta más de lo que se lee, y en este caso, la inserción es constante.

El resto del algoritmo se realizó como se detalla en la introducción.

1.3. Pseudocódigos

Mayores(A, B): Devuelve mayores sumas entre A y $B \longrightarrow \mathrm{O}(n^2)$

Require: A y B ordenados

```
1: i ← 0
2: while n > 0 do
3:     ret.agregar(A[i], B[i])
4:     valor ← A[i] + B[i + 1]
5:     mayores ← BuscoMayor(valor, A, B, i)
6:     valor1 ← A[i + 1] + B[i + 1]
7:     menores ← BuscoMayor(valor1, A, B, i)
8:     menores1 ← BuscoMayor(valor1, B, A, i)
9:     Ordenar(mayores)
10:    Ordenar(menores)
11:    Ordenar(menores1)
12:    Ingresar.agregar(mayores, menores, menores1)
13:    OrdenarListas(ingresar)
14:    j ← 0
15:    while n > 0 do
16:        ret.agregar(ingresar(j))
17:        j + +
18:    end while
19:    i + +
20: end while
```

BuscoMayor(valor,A,B,i): Devuelve la lista de tuplas mayores a *valor* $\longrightarrow \mathrm{O}(n^2)$

Require: A y B ordenados

```
1: k ← 0
2: while k ≤ i do
3:     j = i + 1
4:     while j < tamaño(A) do
5:         if A[j]+B[k] ≥ valor then
6:             ret.agregar(A[j], B[k])
7:             j + +
8:         else
9:             break
10:        end if
11:    end while
12:    k + +
13: end while
14: return ret
```

Los Algoritmos Ordenar y OrdenarListas corresponden a los habituales algoritmos Quick Sort y Merge Sort, cambiando los enteros por la lista de tuplas. Sus complejidades respectivas en peor caso son x^2 y $x * log(x)$ respectivamente.

1.4. Analisis de complejidad

Modelo Uniforme

1.4.1. Mayores

Tamaño de la entrada = 2 * n = x con n cantidad de elementos del conjunto.

La complejidad de Mayores es de O($4n^2 + n + n * log(n)$). Los n^2 corresponden a las 3 veces que llama al algoritmo BuscoMayor, la $n * log(n)$ corresponde al Merge Sort para ordenar a los candidatos y la n restante es del agregado de los candidatos a la lista de retorno Ret.

El resto de las operaciones son basicas, o, en el caso del costo del acceso a las listas, las cuales son Linked Lists, que es constante.

En funcin del tamaño de la entrada, la complejidad quedaria O(x^2).

1.4.2. BuscoMayor

Este algoritmo recorre i veces la lista B desde $n - i$ hasta n. En el peor caso es que i sea $n/2$, lo cual hace recorrer $n/2 * n/2$ veces.

Por lo tanto, la complejidad del algoritmo es O(n^2). Las restantes operaciones realizadas son O(1) dentro del algoritmo.

En función del tamaño de la entrada, la complejidad queda cuadratica, O(x^2).

1.5. Resultados

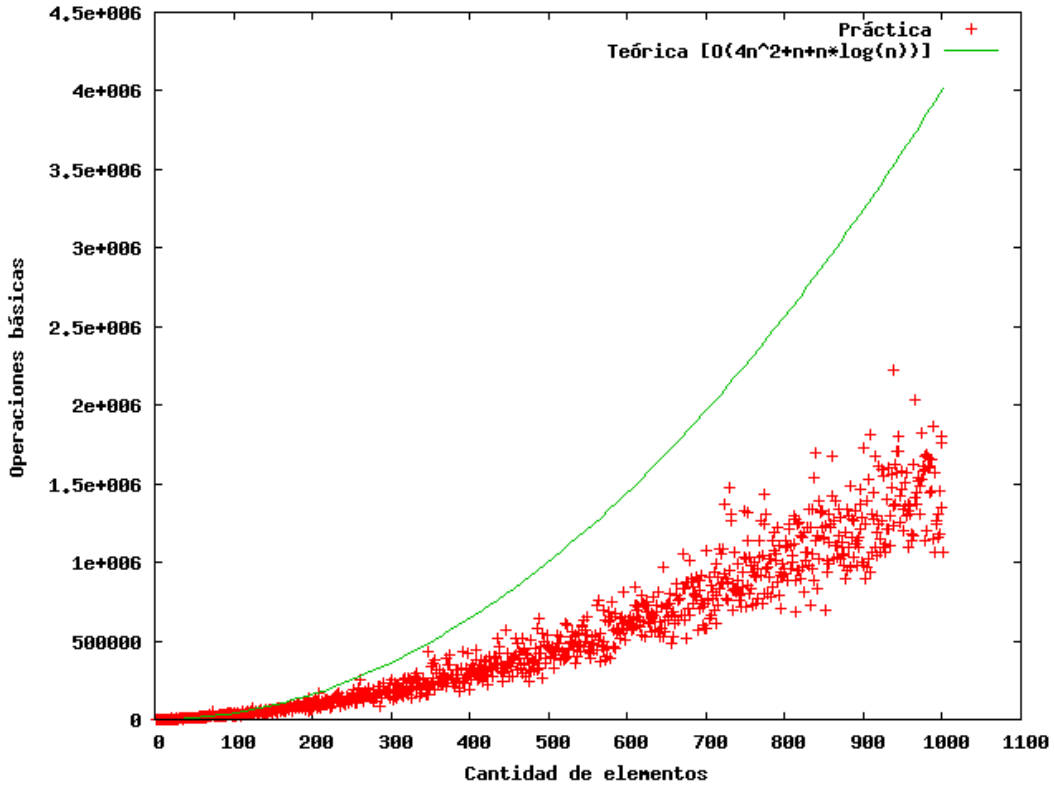

Figura 1: Cantidad de operaciones básicas del algoritmo completo

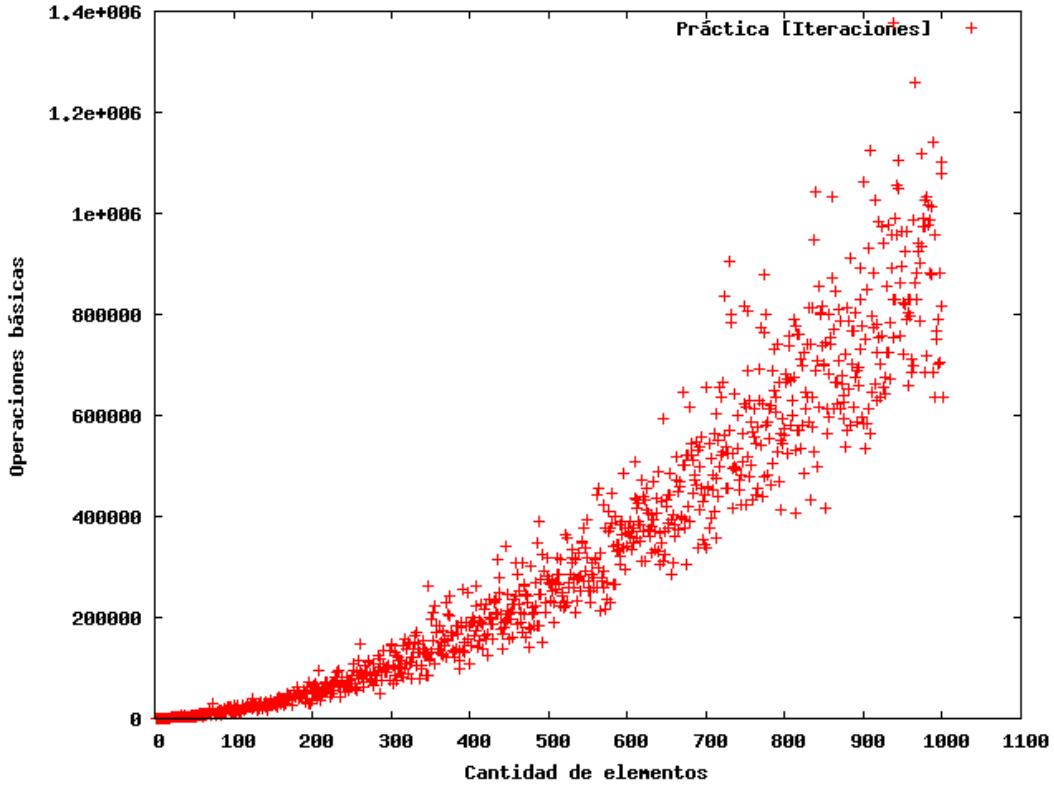

Figura 2: Cantidad de operaciones básicas del algoritmo Valor

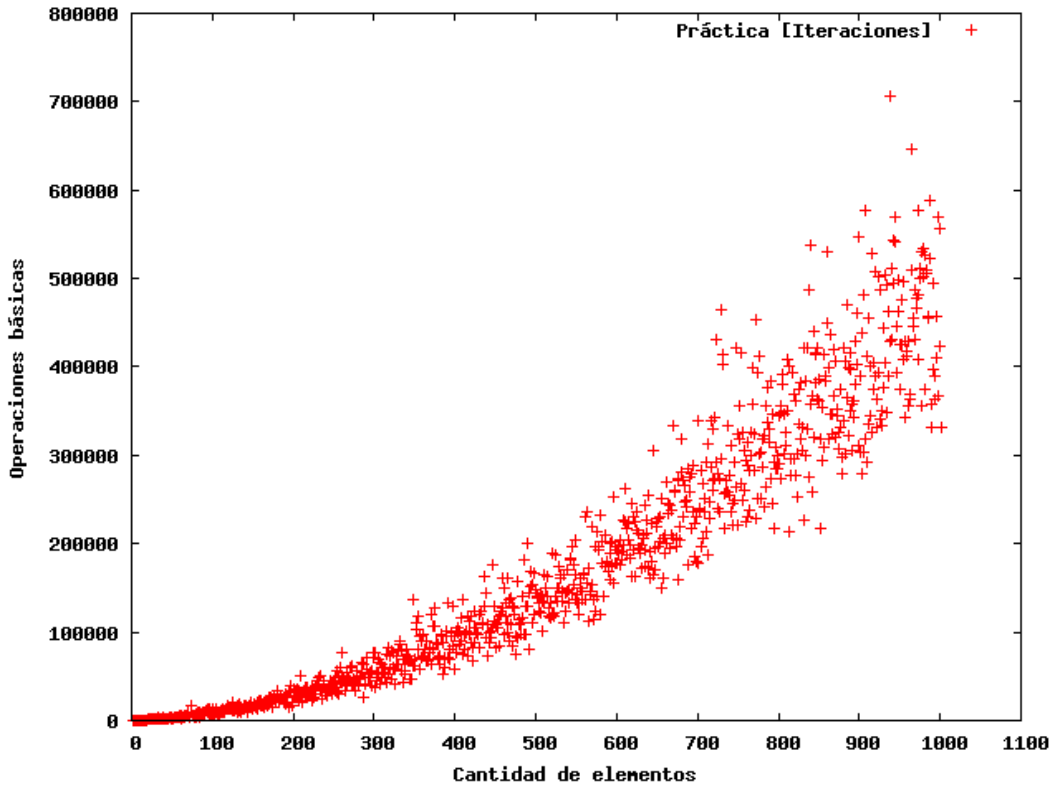

Figura 3: Cantidad de operaciones básicas del algoritmo Ordenar

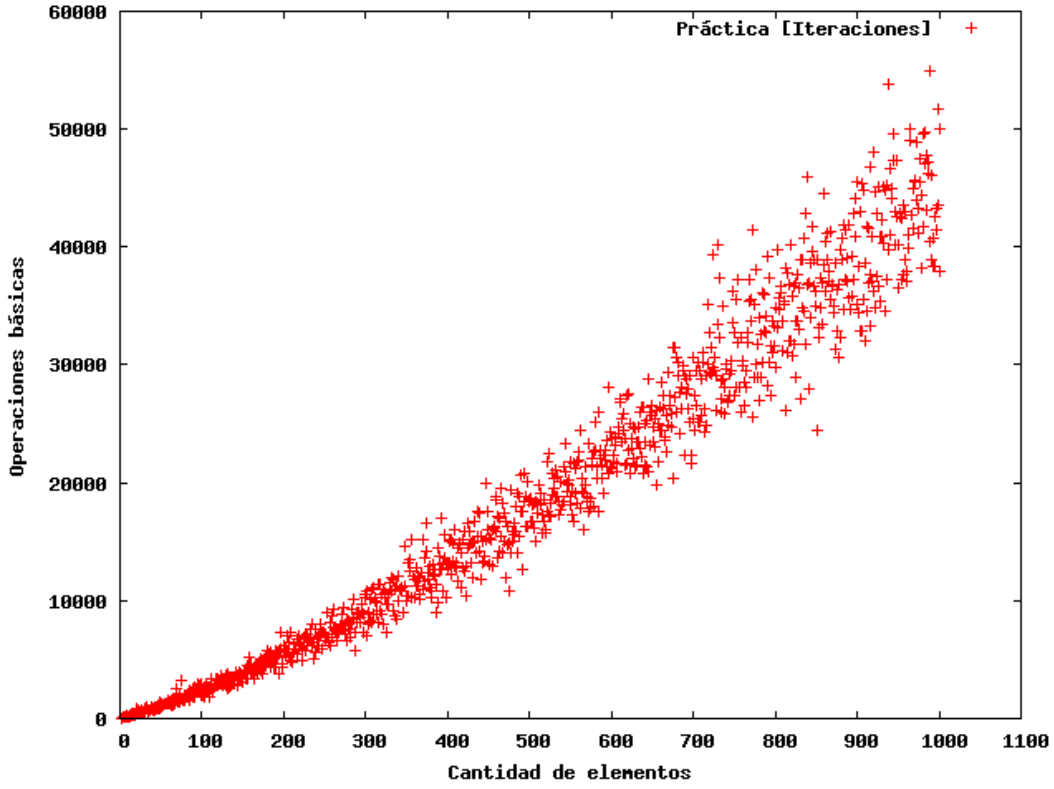

Figura 4: Cantidad de operaciones básicas del algoritmo OrdenarListas

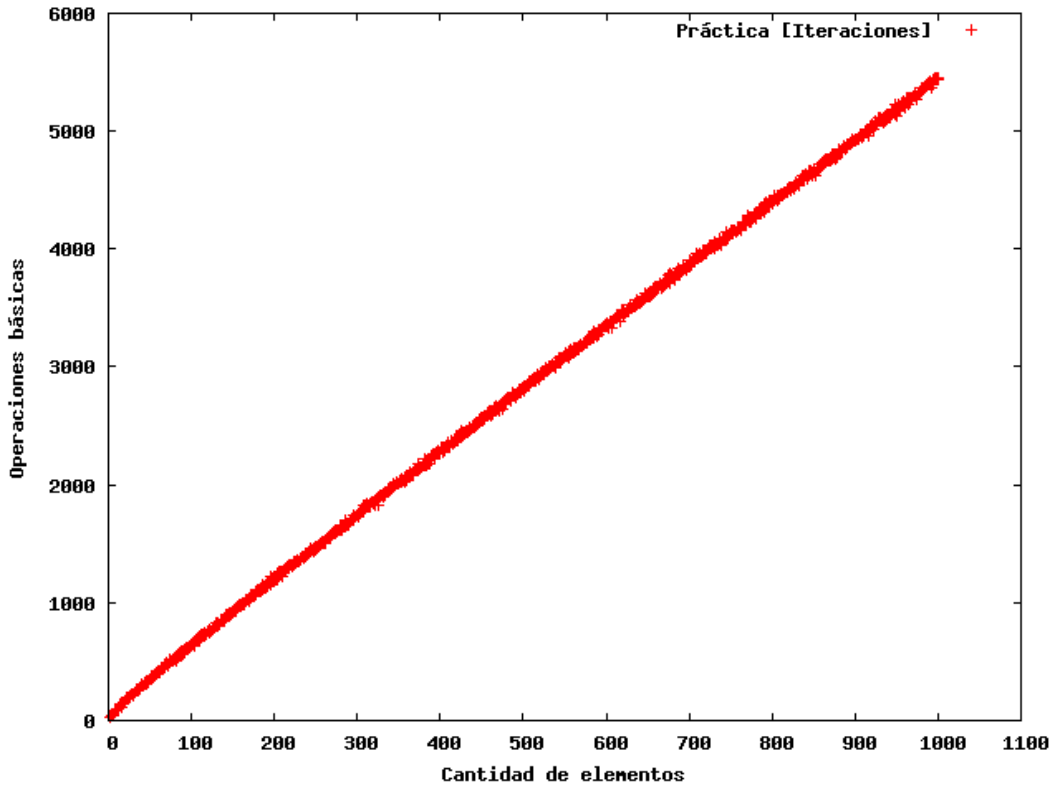

Figura 5: Cantidad de operaciones básicas del algoritmo Mayores

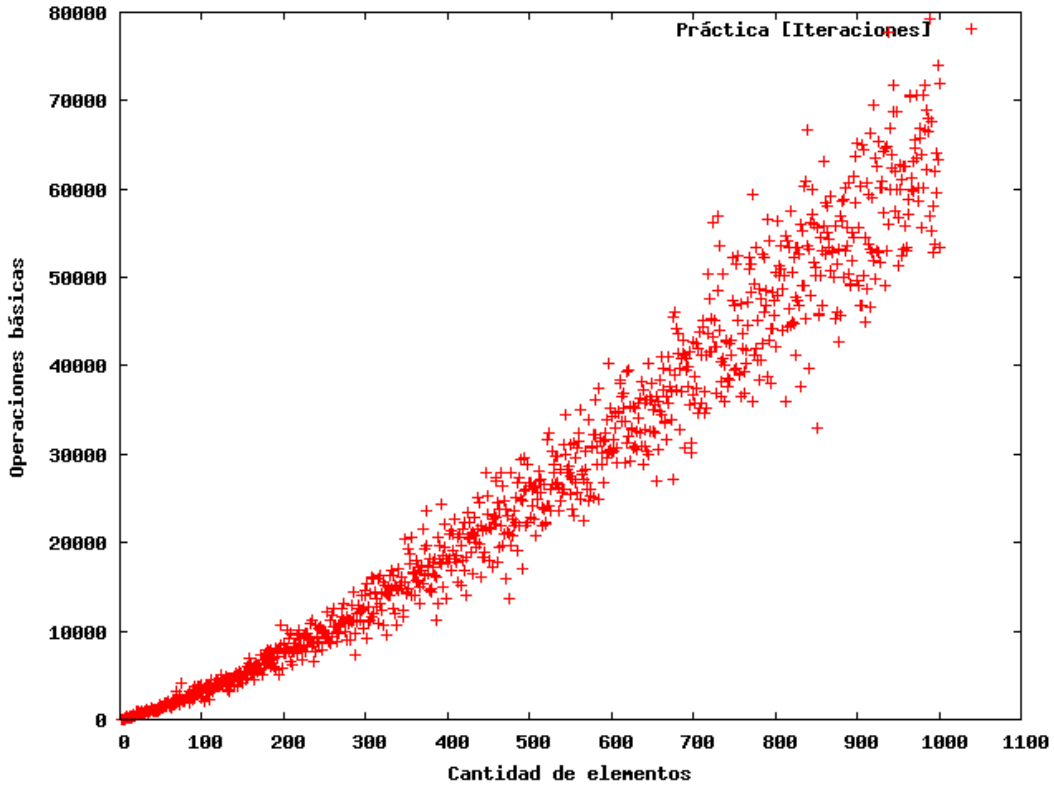

Figura 6: Cantidad de operaciones básicas del algoritmo BuscoMayor

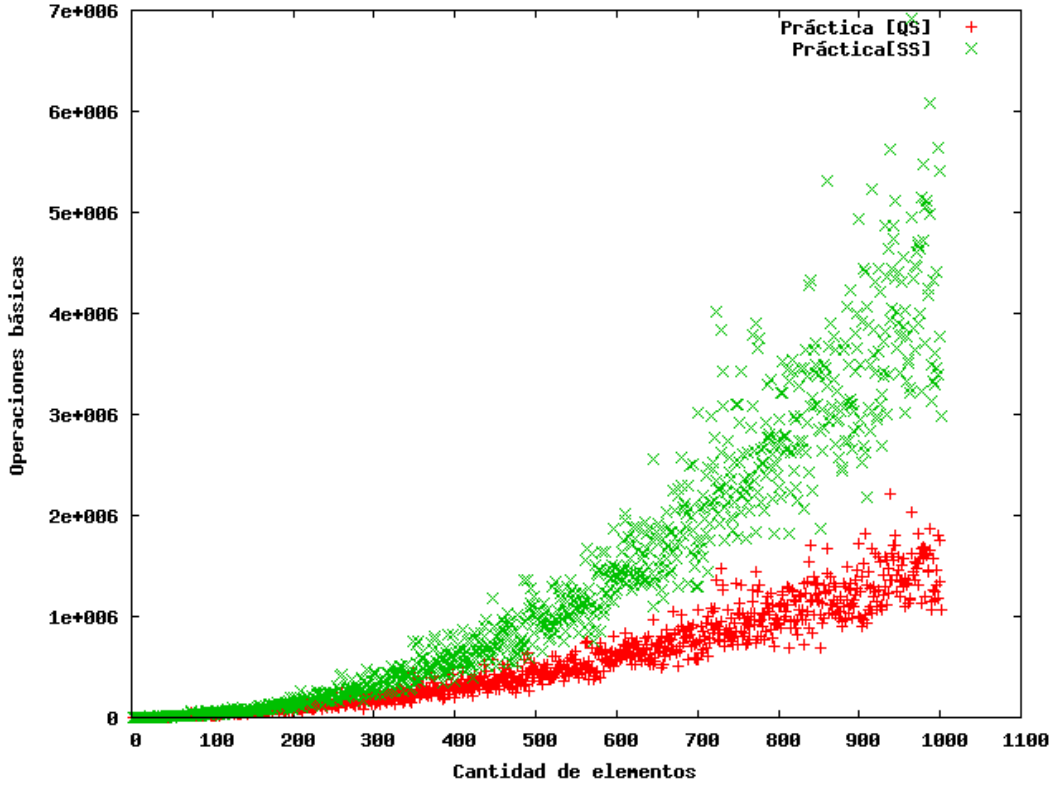

Figura 7: Comparación de Cantidad de operaciones básicas del algoritmo Quick Sort contra Selection Sort

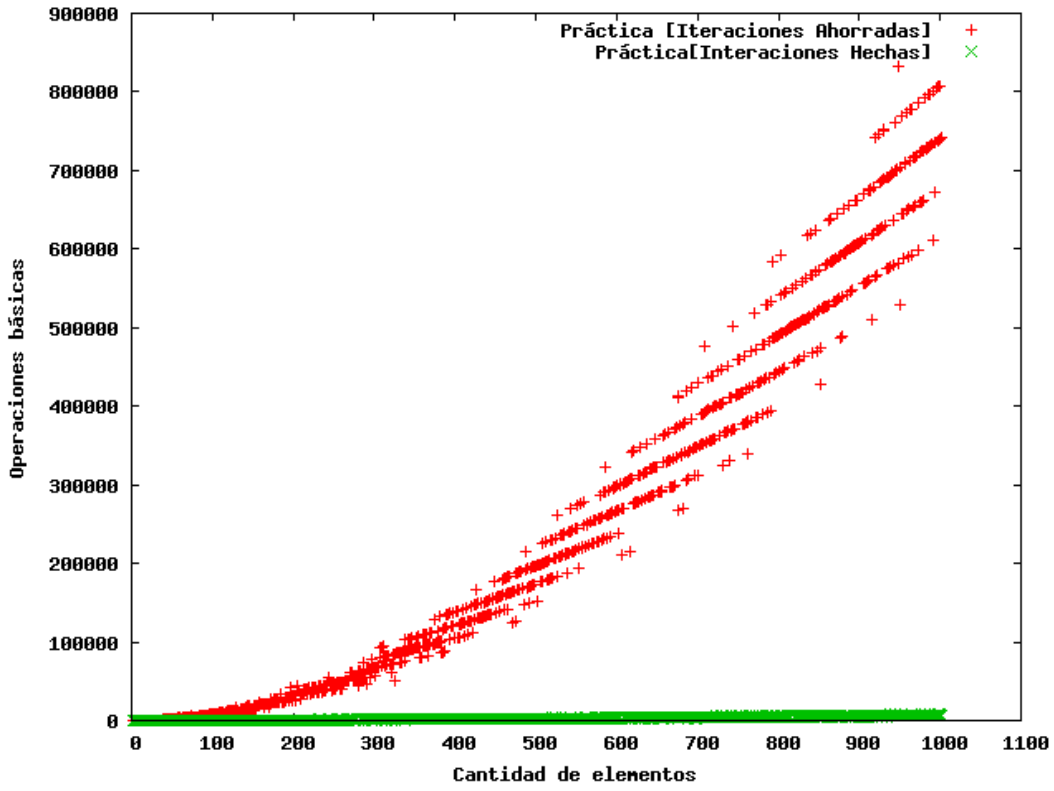

Figura 8: Cantidad de operaciones básicas ahorradas contra hechas

1.6. Discusión

De las figuras se pueden sacar las siguientes conclusiones:

En la primera se ve que la complejidad teórica es mayor a la práctica. Ésto se debe a implementaciones internas de Java, las cuales amortizan muchos de los algoritmos de listas, haciendo menores la cantidad de operaciones. Desde la segunda a la sexta, se ven las cantidades de operaciones desglosadas en cada algoritmo en particular. Observar que la del algoritmo principal es prácticamente lineal en funcion de la cantidad de elementos del conjunto. En el séptimo se ve la comparacion entre Quick sort y Selection sort, y se nota la clara diferencia entre ambos. Si bien los dos tienen igual complejidad en peor caso, en caso del Quick Sort es $n * log(n)$ en caso promedio. En el octavo se ve claramente los ahorros de iteraciones en BuscoMayor al aplicar el break en el else del if. Esto ahorra muchísimas iteraciones innecesarias.

2. Ejercicio 2

2.1. Introducción

El primer algoritmo pensado para resolver este problema fue el de fuerza bruta, el cual recorre los todos los nodos de un árbol de busqueda y calcula la altura cada vez que llega a una hoja.

Dicho árbol empesaría con un nodo con un cero y cumple con la propiedad de que cada nodo (excepto las hojas) tiene una cantidad de hijos equivalente a:

$$Cantidad\ Integrantes\ Familia - Valor\ Nodo - \#(enemistados\ con\ el)$$

Los valores que toman los hijos van desde $Valor\ Nodo + 1$ hasta $Cantidad\ Integrantes\ Familia$ sin tener en cuenta los valores enemistados con el nodo padre.

La cantidad de elementos, suponiendo que no hay enemistades en la familia, que tendria el árbol es $2^n - 1$, donde n es la cantidad de integrantes de la familia.

Luego de tener el algoritmo de fuerza bruta nos vimos con la necesidad de empezar a *podar*, debido a que se realizan muchas comparaciones innecesarias que aumentaban la complejidad.

La *poda* se realizo teniendo en cuenta el siguiente razonamiento: se van calculando alturas (cada una equivale a un tamaño de fiesta), si la máxima de éstas es mayor a la cantidad personas que me quedan para armar una fiesta nueva, no sigo y me quedo con dicha altura.

2.2. Detalles de implementación

Para guardar los datos de entrada se creo una clase llamada *Datos*, la cual contiene el valor de n, de m y una lista con los pares de enemitades.

El algoritmo que resuelve el problema, $TimeForParty$, consta de dos partes:

- La primera refiere a la primer parte del algoritmo, la que calcula cuantos familiares pueden invitarse "de una" ya que no están enemistados con nadie.

  Además en esta parte también se arman una lista con las personas que tienen alguna enemistad y una matriz, las cuales se usan en la segunda parte. Si la matriz tiene un $true$ en la posición $i\ j$ es porque i y j están enemistados entre si. En cambio, si hay un $false$, no.

- La segunda parte es la más costosa en complejidad algorítmica. Es el algoritmo recursivo($MasQueridos$) que recorre el árbol (y realiza la poda correspondiente) buscando la fiesta más grande que se puede hacer con las personas enemistadas. Este resultado es, luego, sumado al obtenido en la parte 1.

Algunas decisiones que se tomaron con respecto a la entrada son:

- n no puede ser menor que 2, si una persona sola conforma la familia entonces no tiene ningun problema para armar la fiesta ya que no puede estar enemistado con el mismo.

- m no puede ser cero, si no hay enemistades en la familia no necesita el algoritmo.

2.3. Pseudocódigo

TimeForParty($dato$**):** *La maxima cantidad de personas que se quieren* $\longrightarrow$

Require: $n > 2,\ m > 0$

```
1: invitados ← 0
2: for i = 1 ... dato.n − 1 do
3:     if EstaEnAlgunPar(i, dato.paresEnemistados)) then
4:         Agregar(listaProblematicos,i)
5:     else
6:         invitados ← invitados + 1
7:     end if
8: end for
9: for j = 0 ... dato.m do
10:    matriz[(paresEnemistados(j))(0)][(paresEnemistados(j))(1)] = true
11:    matriz[(paresEnemistados(j))(1)][(paresEnemistados(j))(0)] = true
12: end for
13: temp ← listaProblematicos.tamano()
14: Lista PosQueridos
15: Lista ParaAlturas
16: mayor ← 0
17: mayorAnt ← 0
18: for inti = 0 ... temp do
19:    if (temp − i) > mayorAnt then
20:        mayor = MasQueridos(matriz, listaProblematicos, PosQueridos, paraAlturas, i);
21:        if mayor > mayorAnt then
22:            mayorAnt = mayor
23:        end if
24:    else
25:        break
26:    end if
27: end for
28: invitados ← invitados + mayorAnt
29: return invitados
```

MasQueridos($matriz\ m, Lista\ lP, Lista\ PQ, Lista\ pA, nat\ i$**):** Calcula la maxima cantidad de familiares peliados que se quieren $\longrightarrow$

```
1: tam = lP.tamano()
2: Agregar(PQ, lP(n))
3: for k = n + 1 ... tam − 1 do
4:    if !TieneEnemigos(lP(k), PQ, m)&&(tam−(k)+PQ.tamano()) > MaximoElemento(pA)
      then
5:        ListaPQ2 ← PQ
6:        MasQueridos(m, lP, PQ2, pA, k)
7:    end if
```

```
 8: end for
 9: Agregar(pA, PQ.tamano())
10: PQ.vaciar()
11: ret ← MaximoElemento(pA)
12: return ret;
```

Aclaración sobre las funciones:

- *EstaEnAlgunPar*(*entero*, *listaDeParesEnemistados*): devuelve *true* en el caso en que el entero que entra como parametro este en alguna de las peleas contenida por la lista de enemistados.
- *TieneEnemigos*(*entero*, *lista*, *matriz*): devuelve *true* en el caso en que el entero este enemistado con algún elemento de la lista.
- *MaximoElemento*(*lista*): devuelve el máximo elemento de la lista.

2.4. Analisis de complejidad

Si la entrada tiene pocas peleas habrá pocas llamadas recursivas y poca cantidad de operaciones. Ahora si hay demasiadas peleas, por ejemplo si todos estan enemistados con todos, no habrá llamadas recursivas y por consiguiente la complejidad estará dada por la primera parte del algoritmo.El peor caso, para el algoritmo que resuelve el ejercicio, es cuando todos los elementos están peleados y pero no todos contra todos sino de forma disjunta.

Como el análisis se hará teniendo en cuenta el peor caso se va suponer que $m = n/2$.

El tamaño de entrada es $2 * m = 2 * n/2 = n$

La complejidad de la primera parte del algoritmo es $O(n^2)$ debido a que recorro n veces la lista de pares enemistados que tiene como tamaño $n/2$, más la complejidad de crear la matriz, que también es $O(n^2)$.

La complejidad de la segunda parte del algoritmo está dada por

$$O(cantidadderepeticiones) * O(MasQueridos)$$

Como la cantidad de repeticiones será menor o igual a n, la complejidad será menor o igual a

$$O(n) * O(MasQueridos)$$

El algoritmo de fuerza bruta tiene complejidad $O(n * 2^n)$ debido a que recorre todos los nodos del árbol y cada vez que llega a una hoja calcula la altura hasta ella. MasQueridos hace lo mismo a excepción que no recorre todo el árbol sino que usa la condición de *poda*. Igualmente dicha condición, si bien mejora un poco la complejidad para valores chicos de n, sigue siendo del mismo orden.

Finalmente la complejidad del algoritmo TimeForParty es $O(n^2 * 2^n)$.

2.5. Resultados

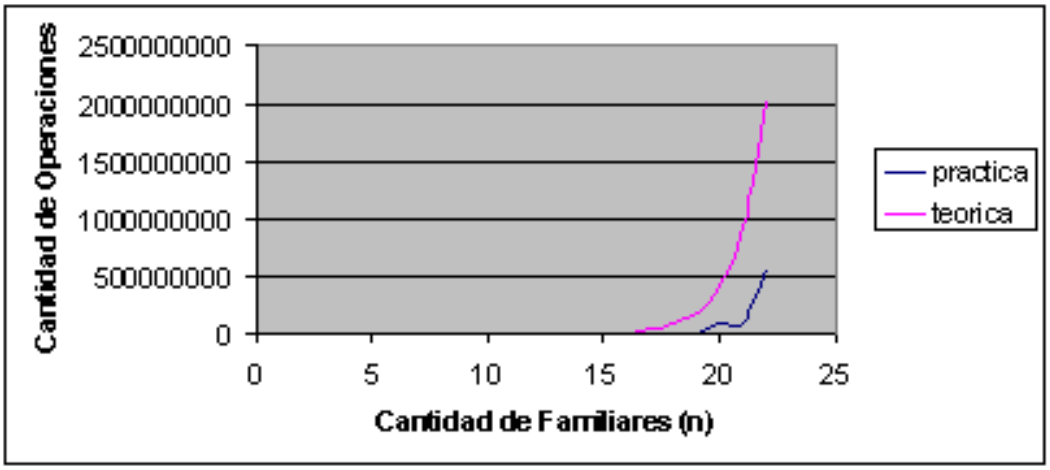

Figura 9: Cantidad de operaciones básicas del algoritmo completo

2.6. Conclusiones

La complejidad teórica es mayor que la práctica, esto se debe a que al aplicar O() perdemos algunas constantes que emparejarían más las lineas del gráfico y a que algunas implentaciones de Java (sobre todo las hechas sobres listas) tiene baja complejidad (por ejemplo en el acceso ordenado a una lista).

3. Ejercicio 3

3.1. Introducción

Al comenzar a pensar la solución de éste ejercicio, partimos de un algortimo por fuerza bruta e intentamos mejorarlo. Creímos que el punto clave en donde se podrían ahorrar operaciones era en la multiplicación de matrices. Luego de varios intentos e investigaciónes por internet, hallamos que existían algoritmos de multiplicación de matrices cuadradas más rápidos que el *standart* de orden cúbico sobre las dimensiones de la matriz. Por ejemplo, el algoritmo de Strassen[1] del orden de $O(k^{2,807})$ y el de Coppersmith-Winograd[2] del orden de $O(k^{2,376})$. Pero ninguno de ellos fue utilizado. La desición provino de notar que los algoritmos eran difíciles de implementar y solamente para obtener una pequeña disminución en la complejidad. Además, los cálculos de complejidad (especialmente en el modelo logarítmico) se hubiesen complejizado demasiado. Evidentemente, el gran salto entre la fuerza bruta y un algoritmo más óptimo no estaba allí.

Entonces se pensó en atacar el problema más desde arriba. Cada multiplicación entre matrices es una operación muy costosa, por lo que nos pareció interesante lograr disminuír el número de éstas al realizar la potenciación. La primer idea que surgió fue de suponer al exponente como potencia de 2, por ejemplo 4. Por fuerza bruta, el algoritmo hubiese hecho

$$A = A * A * A * A$$

costando tres multiplicaciones (complejidad lineal). Pero ésto se puede mejorar, ya que

$$A = (A^2)^2$$

con lo que la cantidad de multiplicaciones requeridas hubiesen sido una para multiplicar A por sí misma (A^2) y otra para multiplicar el resultado por sí mismo ($(A^2)^2$), disminuyendo en una tercera parte la cantidad de multiplicaciones. En efecto, la complejidad resultante (contando solamente multiplicaciones) sería logarítmica. Pero había un serio problema de fondo: el exponente no tenía porque ser potencia de dos.

Con ello en mente, buscamos información sobre como aplicar la misma idea a cualquier exponente. Y nos encontramos con el algortimo de potenciación binaria (*binary power*)[3]. Éste utiliza la misma idea que teníamos pero salva los casos que no son potencia de dos tomando la representación binaria del exponente. Así es como funciona:

- Suponer que quiero realizar A^n.
- Suponer que la representación binaria de n es $b_{k-1}...b_0$, donde cada b_i es un dígito binario.
- Claramente

$$n = \sum_{i=0}^{k-1} b_i 2^i \tag{1}$$

  o sea que para formar a n debo sumar aquellas potencias de 2 en donde su dígito binario de n es 1.

- De ésta forma es fácil construirse el algoritmo. La matriz resultado comienza siendo la identidad (considerar que equivale a A^0). A su vez tengo otra matriz con exponentes de A potencias de dos (comienza siendo $A^{2^0} = A$). Voy tomando los valores de b_0 en adelante y en donde halla un uno, multiplico la matriz resultado por la de las potencias (ya que la multiplicación entre las matrices repercute como una suma entre los exponentes). Al final de cada iteración elevo al cuadrado la matriz de las potencias de 2 para pasar a la siguiente. Repitiendo con cada b_i es como voy transformando el exponente del resultado en n. Al terminar con b_{k-1}, éste va a ser efectivamente n (y la justificación es la sumatoria expresada en (1)).

Éste algoritmo es del orden de $2\log_2(n)$ multiplicaciones en el peor caso, con lo que nos quedamos bastante satisfechos y fue el utilizado.

3.2. Detalles de implementación

Para facilitar la implementación lo primero que se hizo fue una clase Matriz, la cual representa una matriz cuadrada y posee solamente los métodos necesarios para resolver éste ejercicio (principalmente, Multiplicar y Potenciar).

Al algoritmo Potenciar se le hicieron pequeños cambios al implementarlo con el único fin de hacerlo más eficiente:

- La representación binaria de n no se calcula antes de comenzar a multiplicar matrices, sino que se realizan las dos operaciones a la vez. Se utilizó el algoritmo *standart* para obtenerla.
- Se agregó una guarda a la multiplicación de *pot*2 para evitar que se realize una multiplicación inútil cuando ya se alcanzó el resultado. Ésto suma $O(\log_2(n))$ operaciones básicas a la complejidad final del algoritmo, pero evita sumar el $O(k^3)$ de la complejidad de una multiplicación, por lo que en la práctica genera una notable optimización.

3.3. Pseudocódigos

Potenciar(A, n): Devuelve $A^n \longrightarrow \mathrm{O}(log_2(n)k^3)$

Require: $A \in \mathbb{N}^{k \times k}$

```
1: if n = 0 then
2:     return I ∈ N^{k×k}
3: end if
4: ret ← I
5: pot2 ← A
6: while n ≥ 1 do
7:     if 2 | n then
8:         n ← n / 2
9:     else
10:        n ← (n − 1) / 2
11:        ret ← Multiplicar(ret, pot2)
12:    end if
13:    pot2 ← Multiplicar(pot2, pot2)
14: end while
15: return ret
```

Multiplicar(A, B): Multiplica la matriz A por $B \longrightarrow \mathrm{O}(k^3)$

Require: $A, B \in \mathbb{N}^{k \times k}$

```
1: ret ∈ N^{k×k}
2: for i = 1 ... k do
3:     for j = 1 ... k do
4:         ret_ij ← Σ_{t=1}^{k} A_it B_tj
5:     end for
6: end for
7: return ret
```

3.4. Análisis de complejidad

3.4.1. Multiplicar

Modelo Uniforme

Fácilmente se puede ver en el algoritmo que para cada fila de A (las cuales son k) y cada columna de B (las cuales también son k) se debe recorrer cada elemento de ellas (los cuales, nuevamente, son k). Notar que siempre realiza lo mismo, sin importar de k o los valores en la matriz, por lo que no hay mejor ni peor caso.

Luego, la complejidad uniforme es $\mathrm{O}(k^3)$.

El tamaño de la entrada será igual a la suma de los tamaños de todos los elementos de cada matriz:

$$t = \sum_{i=1}^{k} \sum_{j=1}^{k} \log_2(A_{ij}) + \log_2(B_{ij})$$

Pero como, para acotar, se considera que todos los elementos de cada una son el mayor de ambas (llamado m), el tamaño se puede expresar como

$$t = \sum_{i=1}^{k} \sum_{j=1}^{k} \log_2(m) + \log_2(m) = 2k^2 \log_2(m) \in O(k^2 \log_2(m))$$

Por lo tanto, la complejidad uniforme es $\mathrm{O}(\sqrt{t/2}\ t/2) = \mathrm{O}(t^{3/2})$, con lo que es **superlineal** en función del tamaño de entrada.

Modelo Logarítmico

Para facilitar los cálculos, se considerará que todos los valores de la matriz serán el más grande de todos y el que, por consiguiente, más espacio ocupe. Ésto no afectará la validez del resultado ya que se estará calculando un orden, una cota superior. Supongo que m es éste valor. Luego, el espacio que ocupará su representación binaria será $\lceil \log_2(m+1) \rceil$. Como es del orden de $\mathrm{O}(\log_2(m))$, simplemente se considerará a éste como el tamaño de m. De ésta forma, realizar $m+m$ y $m*m$ se considerará que cuestan $\mathrm{O}(2\log_2(m)) = \mathrm{O}(\log_2(m))$ cada operación.

En Multiplicar, para calcular cada valor del resultado se deben multiplicar m con m tantas veces como la dimension de la matriz. Éstos productos deberán ser sumados entre sí. Primero $m^2 + m^2$, luego $2m^2 + m^2$, luego $3m^2 + m^2$, etc. Es decir que cada suma costará

$$O(\log_2(cm^2) + \log_2(m^2)) = O(2\log_2(m) + \log_2(c) + 2\log_2(m)) = O(\log_2(m))$$

Por lo tanto, para calcular un valor de la matriz resultado, se deberán calcular k productos cada uno con un costo de $\mathrm{O}(\log_2(m))$ y $k-1$ sumas cada una con el mismo costo. Ésto se deberá realizar para cada elemento de la matriz resultado, la cual contiene k^2 de ellos. Luego, la complejidad logarítmica de Multiplicar estará dada por

$$O((k\log_2(m) + (k-1)\log_2(m))k^2) = O(k^3 \log_2(m))$$

Se verifica fácilemnte que

$$t^2 \geq k^3 \log_2(m) \iff k^4 \log_2^2(m) \geq k^3 \log_2(m) \iff k \geq 1$$

Por lo tanto, la complejidad logarítmica se puede expresar como

$$O(t^2)$$

con lo que será **cuadrática** en función del tamaño de la entrada.

3.4.2. Potenciar

Modelo Uniforme

Dado que una multiplicación entre matrices es muy costosa, y que es la más costosa en éste algoritmo[1], el peor de los casos para Potenciar será cuando deba realizar la mayor cantidad de éstas. Como *pot*2 siempre será elevada al cuadrado, en cada iteración se realizará por lo menos una multiplicación. Pero la otra, correspondiente a $ret * pot2$, sólo cuando el n actual sea impar. Ésto, como mejor se explicó en la Introducción del presente, sucede si se encuentra un 1 en la representación binaria de n.

Por lo tanto, el peor n de entrada será aquel con todos unos en su representación binaria. Y ésto es equivalente a decir que

$$n = 2^r - 1$$

para algún $r \in \mathbb{N}$.

Ya que con un n así se realizarán 2 multiplicaciones por cada dígito binario suyo (excepto en la última iteración que realiza sólo una), y que la cantidad de éstos es $\log_2(n)$, la cantidad de multiplicaciones serán $\mathrm{O}(2 * \log_2(n) - 1)$. Pero, como cada multiplicación es $\mathrm{O}(k^3)$, la complejidad uniforme de Potenciar será

$$O(2 \log_2(n) k^3) = O(k^3 \log_2(n))$$

Notar que el mejor caso será aquel donde n tenga la mayor cantidad de ceros posible en su representación binaria. És decir, $n = 100\ldots00$. Allí, sólo realizaría $\log_2(n)$ multiplicaciones para *pot*2 y luego una más para hacer $ret * pot2$ (que en realidad sería $I * A^n$). Por lo tanto, el mejor caso será con n potencia de dos.

El tamaño de la entrada será igual a la suma del tamaño de todos los elementos de A mas el de n. Como se considera que todos los elementos de A son el máximo de ellos (llamado m), el tamaño de la entrada será

$$t = \log_2(n) + \sum_{i=1}^{k} \sum_{j=1}^{k} \log_2(m) = k^2 \log_2(m) + \log_2(n)$$

Tratando de acotar la complejidad, se puede ver primero que

$$t^2 = (k^2 \log_2(m) + \log_2(n))^2 =$$

[1] Notar que los costos de copiar A en *pot*2 e I en ret son del orden de $\mathrm{O}(k^2)$ cada una, pero como sólo se ejecutan una vez, y la complejidad de Multiplicar es mayor por ser $\mathrm{O}(k^3)$, se desprecian.

$$k^4 \log_2^2(m) + 2k^2 \log_2(m) \log_2(n) + \log_2^2(n) \geq k^2 \log_2(n)$$

y que

$$\sqrt{t} = \sqrt{k^2 \log_2(m) + \log_2(n)} \geq \sqrt{k^2} = k$$

Por lo tanto, la complejidad puede ser expresada como

$$O(t^2\sqrt{t}) = O(t^{5/2})$$

con lo que ésta sería **supercuadrática**[2] en función del tamaño de la entrada.

Modelo Logarítmico

Dado el enunciado, se considerará que $n = 2^r$ para algún $r \in \mathbb{N}$, además de las suposiciones hechas en el cálculo de Multiplicar.

Sea m_1 el mayor elemento de A. Como se considera que todos los elementos de la matriz son el mayor, todos los elementos de A serán m_1. Se define además m_i como el mayor elemento de A^i. De ésta forma

- $m_2 = \sum_{j=1}^{k} m_1 \; m_1 = k m_1^2$
- $m_3 = \sum_{j=1}^{k} m_2 \; m_2 = \sum_{j=1}^{k} k^2 m_1^4 = k^3 m_1^4$
- ...
- $m_i = \sum_{j=1}^{k} m_{i-1} \; m_{i-1} = k^{2^i - 1} m_1^{2^i}$

Dado que n es potencia de 2, mientras que sea mayor a 1 el algoritmo ejecutará dos divisiones enteras (costando cada una $\mathrm{O}(\log_2(n))$) y una multiplicación matricial para elevar *pot*2 al cuadrado. Notar que en la iteración i, *pot*2 será A^i, por lo que todos los elementos de *pot*2 serán m_i. De ésta forma, el costo de la multiplicación será del orden de $\mathrm{O}(k^3 \log_2(m_i))$. Ésto será ejecutado para cada $1 \leq i \leq \log_2(n)$. Por lo tanto, la complejidad de éstas multiplicaciones será

$$\sum_{i=1}^{\log_2(n)} k^3 \log_2(m_i) = k^3 \sum_{i=1}^{\log_2(n)} \log_2(k^{2^i-1} m_1^{2^i}) =$$

$$k^3 \left(\sum_{i=1}^{\log_2(n)} (2^i - 1) \log_2(k) + \sum_{i=1}^{\log_2(n)} 2^i \log_2(m_1) \right) =$$

$$k^3 \log_2(k) \left(\sum_{i=1}^{\log_2(n)} 2^i \; - \; \log_2(n) \right) + k^3 \log_2(m_1) \sum_{i=1}^{\log_2(n)} 2^i =$$

$$k^3 \log_2(k) \left(\sum_{i=0}^{\log_2(n)} 2^i \; - \; (\log_2(n) + 1) \right) + k^3 \log_2(m_1) \left(\sum_{i=0}^{\log_2(n)} 2^i \; - \; 1 \right) =$$

[2]Nombre generado por inducción en **superlineal**. Tranquilamente se podría afirmar que es **cúbica**.

$$k^3 \log_2(k) \left(\frac{2^{\log_2(n)+1} - 1}{2-1} - (\log_2(n) + 1) \right) + k^3 \log_2(m_1) \left(\frac{2^{\log_2(n)+1} - 1}{2-1} - 1 \right) =$$

$$k^3 \log_2(k) \left(2^{\log_2(n)+1} - 2 - \log_2(n) \right) + k^3 \log_2(m_1) \left(2^{\log_2(n)+1} - 2 \right) =$$

$$k^3 \log_2(k) 2^{\log_2(n)+1} - 2k^3 \log_2(k) - k^3 \log_2(k) \log_2(n) + k^3 \log_2(m_1) 2^{\log_2(n)+1} - 2k^3 \log_2(m_1) \leq$$

$$k^3 \log_2(k) 2^{\log_2(n)+1} + k^3 \log_2(m_1) 2^{\log_2(n)+1} =$$

$$k^3 2^{\log_2(n)+1} (\log_2(k) + \log_2(m_1)) \in$$

$$O\left(k^3 2^{\log_2(n)} (\log_2(k) + \log_2(m_1)) \right)$$

Si se agrega la complejidad de las demás operaciones, las $\log_2(n)$ iteraciones costarán

$$O\left(k^3 2^{\log_2(n)} (\log_2(k) + \log_2(m_1)) \right) + O\left(\log_2(n)(2 \log_2(n)) \right) =$$

$$O\left(k^3 2^{\log_2(n)} (\log_2(k) + \log_2(m_1)) + \log_2^2(n) \right) = O\left(k^3 2^{\log_2(n)} (\log_2(k) + \log_2(m_1)) \right)$$

(no altera el resultado ya que $2^{\log_2(n)} \geq \log_2(n)$).

Luego hará una última iteración. Aquí sucederá que $n = 1$, se agregará una multiplicación adicional entre *ret* (quien todavía será I) y *pot*2 (quien para ése entonces será A^n). Dado que A es una matriz de naturales, cada vez que se la eleva a alguna potencia natural todos sus valores mayores o iguales que los anteriores. Es por ésto que se verifica que el máximo elemento en *pot*2 será siempre mayor o igual que el máximo en *ret* (en efecto, sólo serán iguales si $A = I$). Por lo tanto, considero que todos los elementos de *ret* son m_n, y la complejidad de ésta multiplicación será

$$O(k^3 \log_2(m_n)) = O(k^3 \log_2(k^{2^n - 1} m_1^{2^n})) = O\left(k^3 \left((2^n - 1) \log_2(k) + 2^n \log_2(m_1) \right) \right) =$$

$$O\left(k^3 2^n \left(\log_2(k) + \log_2(m_1) \right) \right)$$

Notar que la complejidad de ésta sola multiplicación es de mayor orden que todas las anteriores juntas. Ésto se debe a que a medida que se eleva al cuadrado una matriz de naturales, sus valores aumentan rápidamente. Por lo que todos los valores de A^n ocupan un espacio mayor en orden que aquel ocupado por todos sus antecesores. Así es como las operaciones son mucho más costosas. Ésto podría tenerse en mente a la hora de realizar implementaciones del algoritmo para evitar éstas multiplicaciones ya que son innecesarias cuando n es potencia de 2 (ya que es más eficiente copiar el valor de *pot*2 a *ret* en lugar de hacer $I * A^n$)[3].

Juntando ésta última complejidad con la de las demás iteraciones se obtiene que la complejidad logarítmica de Potenciar es

$$O\left(k^3 2^{\log_2(n)} (\log_2(k) + \log_2(m_1)) \right) + O\left(k^3 2^n \left(\log_2(k) + \log_2(m_1) \right) \right) =$$

$$O\left(k^3 2^n \left(\log_2(k) + \log_2(m_1) \right) \right)$$

(ya que $2^n \geq 2^{\log_2(n)}$)

[3]El grupo decidió no implementarlas con el fin de mantener el algoritmo simple.

Se demostró en el análisis uniforme que el orden del tamaño de la entrada es $t = k^2 \log_2(m) + \log_2(n)$ (notar que $m_1 = m$).

Sabiendo que

$$2^t = 2^{k^2 \log_2(m) + \log_2(n)} = \left(2^{\log_2(m)}\right)^{k^2} 2^{\log_2(n)} = m^{k^2} n$$

y que

$$2^{2^t} = 2^{m^{k^2} n} = (2^n)^{m^{k^2}}$$

se puede ver facilmente que

- $k^3 \leq 2^{2^t}$ ya que en el segundo, k aparece como exponente y encima al cuadrado.
- $2^n \leq 2^{2^t}$ ya que en el segundo aparece 2^n pero con aún más exponentes.
- $\log_2(k) \leq 2^{2^t}$ trivial a partir del primer ítem por ser $\log_2(m) \leq k^3$.

Como la única variable que se encuentra en el exponente en $k^3 2^n \log_2(k)$ es n, todo ése producto nunca superará a 2^{2^t}. Sucede lo mismo con $k^3 2^n \log_2(m)$, ya que aquí aparece $\log_2(m)$ pero éste será menor porque m aparece en un exponente en 2^{2^t}. Luego

$$k^3 2^n \log_2(k) + k^3 2^n \log_2(m) \leq 2^{2^t} + 2^{2^t} = 2^{2^t+1} \in O(2^{2^t})$$

Por lo que la complejidad logarítmica se puede expresar como

$$O(2^{2^t})$$

con lo cual, ésta es **exponencial** en función del tamaño de la entrada.

3.4.3. Fuerza Bruta

El cambio con respecto a la fuerza bruta recae en Potenciar exclusivamente. Aquí, para calcular A^n se irá multiplicando el resultado actual con A hasta formar A^n. Más exactamente, se realizarán $n - 1$ multiplicaciones. Por lo tanto, la complejidad será $\mathrm{O}((n-1)k^3) = \mathrm{O}(nk^3)$.

3.5. Resultados

Los gráficos a continuación cuentan la cantidad de operaciones de los algoritmos en función de las variables de entrada. Se decidió así en lugar de hacerlos en función del tamaño de la entrada para que sean sencillos de analizar.

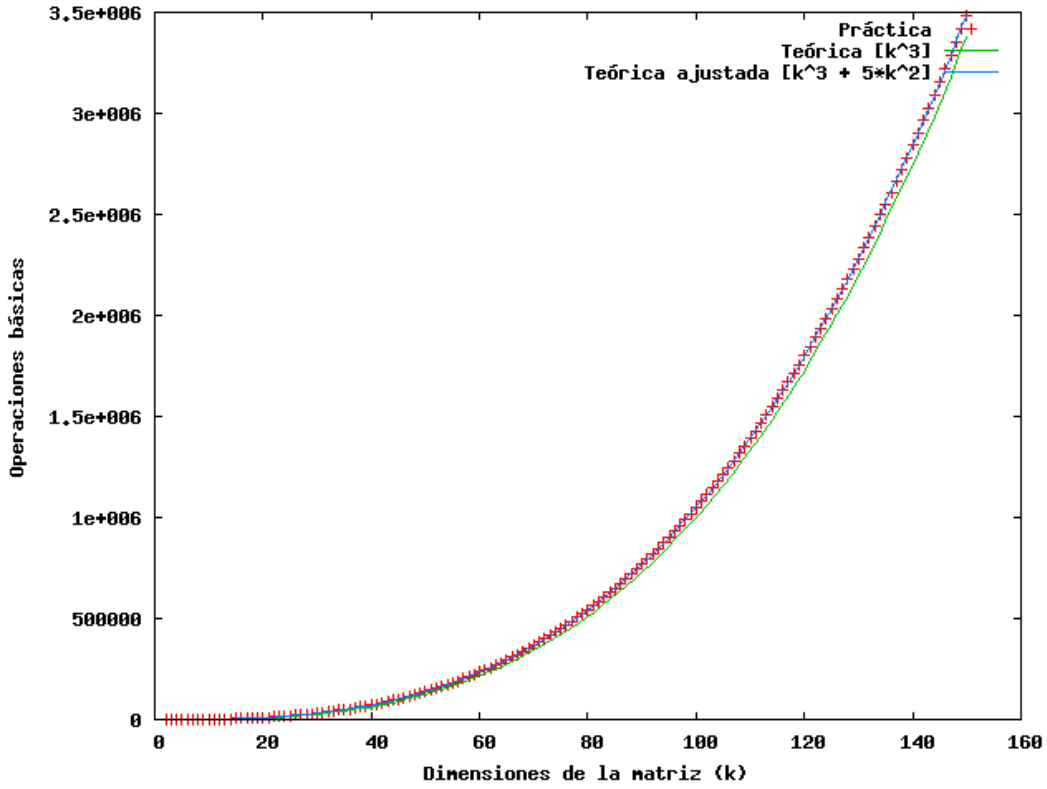

Figura 10: Cantidad de operaciones básicas de Multiplicar en función de las dimensiones de las matrices

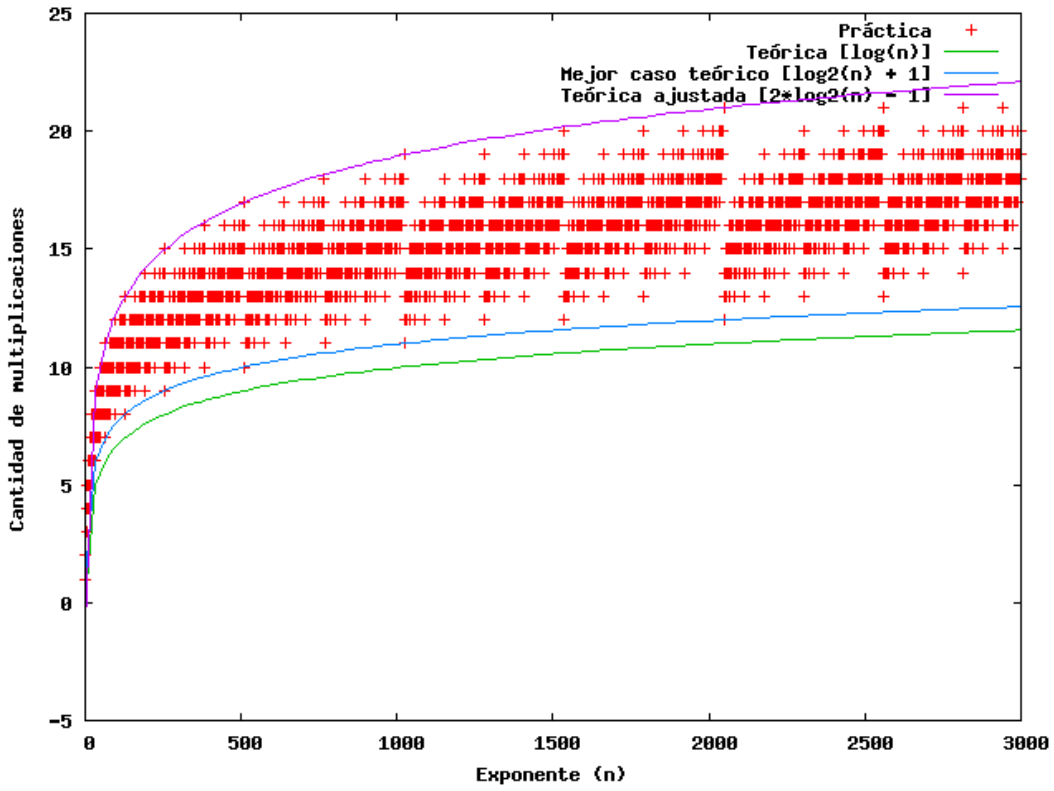

Figura 11: Cantidad de multiplicaciones realizadas en Potenciar en función del exponente

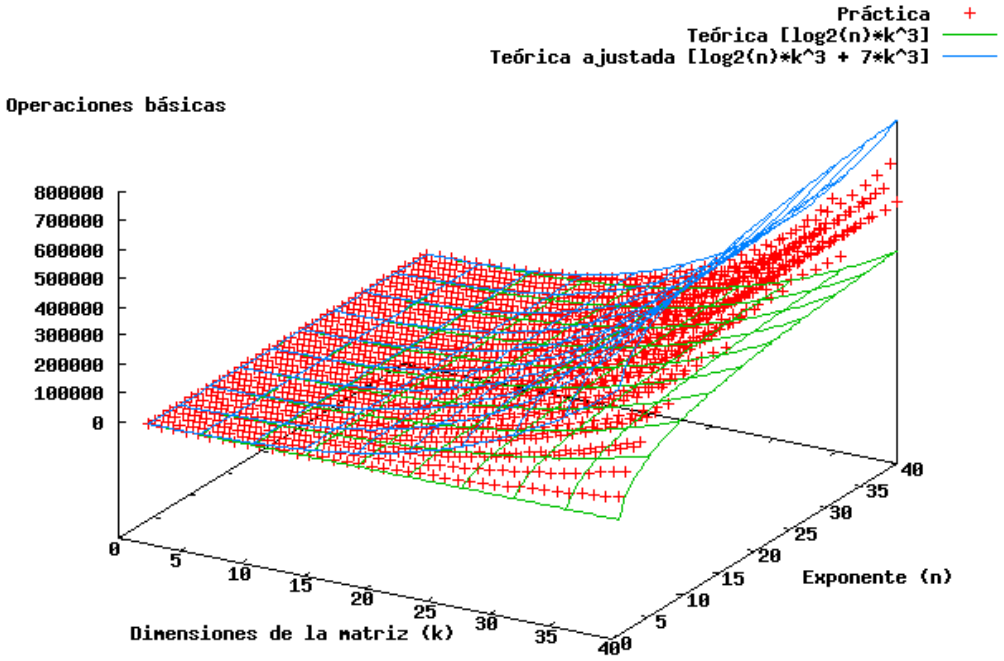

Figura 12: Cantidad de operaciones básicas de Potenciar en función del exponente y las dimensiones de las matrices

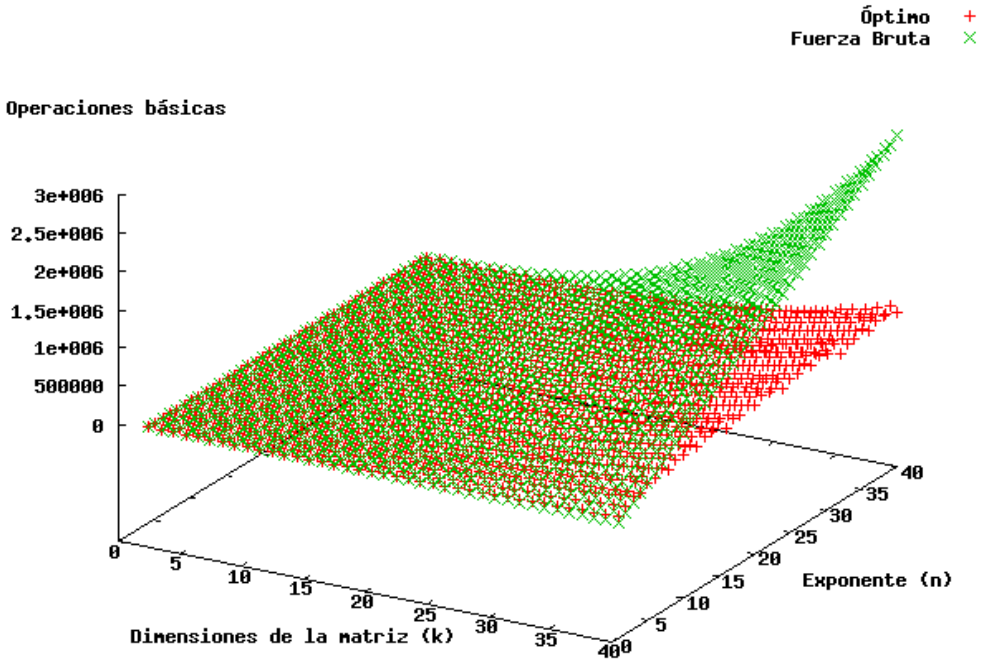

Figura 13: Cantidad de operaciones básicas de Potenciar mediante el algoritmo óptimo y el de fuerza bruta en función del exponente y las dimensiones de las matrices

3.6. Discusión

El primer gráfico muestra que la complejidad teórica y práctica son prácticamente la misma. Era de esperar un comportamiento así ya que el algoritmo es muy simple. Además, la curva práctica está bien definida por ser el algoritmo $O(k^3)$ en cualquier caso, no simplemente el "peor"(porque, en realidad, aquí no existe peor caso). Las constantes a la hora de sumar los valores son los responsables de ese $5k^2$ adicional en comparación con el caso teórico (siguen estando en el mismo orden por ser $k^2 < k^3$).

En el segundo se ve la cantidad de multiplicaciones matriciales que realiza Potenciar en función de n. Se verifica que en la práctica ésta cantidad es exactamente $2\log_2(n) - 1$ en el peor caso. Es decir, es exactamente como se calculó teóricamente. Notar que siempre el valor siguiente a un peor caso es de los mejores casos (es decir, cuando el exponente es potencia de 2) y que la cantidad de multiplicaciones en éstos casos es $\log_2(n) + 1$ como también se calculó teóricamente.

El tercer gráfico se apoya en los dos anteriores. És decir, como las complejidades teóricas anteriores fueron válidas, es de esperar que ésta también lo sea ya que, como se ve en el análisis de la complejidad, se basa en éstas dos. Y efectivamente, el orden de la complejidad teórica es buena cota de la complejidad práctica.

Por último, el cuarto gráfico es el análisis contra el algoritmo de fuerza bruta. Como era de esperar, al necesitar más multiplicaciones el segundo que el primero la cantidad total de operaciones básicas realizadas es notablemente mayor a medida que n y k aumentan.

4. Ejercicio 4

4.1. Introducción

Desde el principio, para resolver este problema se pensó en armar la factorización probando todos los naturales (comenzando por el 2, ya que el 1 no es primo ni compuesto). Si éste era primo, se verificaba si dividia al número de entrada y se agregaban tantas repeticiones como fueran necesarias. Y ello fue lo que terminamos haciendo.

Para verificar si un número era primo o no se utilizó el corolario de la criba de Eratóstenes, es decir, probar si algún otro primo menor a su raíz lo divide. De no ser así, es porque efectivamente lo era; caso contrario, no. Pero éste algoritmo era extremadamente ineficiente ya que debía efectuar llamadas recursivas para cada natural menor a la raíz de la entrada para verificar si éste era o no primo (y luego verificar si lo dividía). Por eso se optó por reutilizar mucha de ésta información. Ésto es, evitar preguntar si un número es primo mas de una vez. Para ello se agregó una lista ordenada como entrada en donde deverán estar todos los primos menores al valor que se está pasando. La ganancia fue del orden exponencial.

4.2. Detalles de implementación

En cierta manera, los algoritmos a implementar fueron relativamente sencillos. La mayor desición a tomar fue con respecto al tipo de lista utilizado.

La lista *ret* en Factorización tiene como propósito mantener valores en el orden que fueron agregados. Su acceso aleatorio no es relevante ya que en ningún momento se toman valores ya guardados. En cambio, es crítico que el insertado sea en tiempo constante ya que la complejidad final será afectada por la cantidad de operaciones que conlleva el insertado de cada factor. Es por ésto que se eligió la lista doblemente enlazada, la cual tiene un acceso lineal pero un insertado constante.

La otra lista utilizada fue la de *primos* en Factorización. El acceso aleatorio en éste caso tampoco es importante porque los únicos accesos realizados son lineales (en EsPrimo), con lo que acceder al siguiente elemento es siempre O(1). Pero el insertado sí que importa ya que si el valor a verificar efectivamente es primo, entonces hay que agregarlo. Por eso utilizamos también una lista doblemente enlazada.

4.3. Pseudocódigos

Factorizacion(n): Devuelve la factorización en primos de $n \longrightarrow O(n\sqrt{n})$

Require: $n \in \mathbb{N}$ tal que $n \geq 2$

```
1: p ← 2
2: while n > 1 do
3:     if EsPrimo(p, primos) then
4:         primos.AgregarAtras(p)
5:         while p | n do
6:             n ← n/p
7:             ret.AgregarAtras(p)
8:         end while
9:     end if
10:    p ← p + 1
11: end while
12: return ret
```

EsPrimo(n): Decide si n es o no primo $\longrightarrow O(\sqrt{n})$

Require: *primos* tiene todos los primos enteros menores a n ordenados crecientemente

```
1: r ← ⌊√n⌋;
2: for all p ≤ r in primos do
3:     if p | n then
4:         return false
5:     end if
6: end for
7: return true
```

4.4. Análisis de complejidad

4.4.1. EsPrimo

Modelo Uniforme

La idea de este algoritmo se basa en la Criba de Eratóstenes. Es decir, afirma que si para cierto número n no existe un primo p menor a $\sqrt{n}$ tal que $p \mid n$, entonces n es primo.

Fácilmente se ve que el peor caso es cuando n efectivamente es primo, ya que fue necesario calcular todos los restos por división entera con los primos menores a $\sqrt{n}$. La mayor cantidad posible de valores leídos es $\lfloor\sqrt{n}\rfloor$, y viene de suponer que todos los naturales menores a ese valor son primos.

Luego, la complejidad uniforme es $\mathrm{O}(\sqrt{n})$.

El tamaño de la entrada será el de n mas el de *primos*. El primero será $\log_2(n)$, mientras que el segundo se puede acotar considerando que todos los valores de 2 a $n-1$ son primos. Como el logaritmo es creciente, el mayor de éstos será $n-1$. Asique, considerando que todos los valores son iguales a éste, el tamaño de la lista quedaría como

$$\sum_{i=2}^{n-1} \log_2(n-1) = (n-2)\log_2(n-1) \in O(n\log_2(n))$$

Agregándo el tamaño de n quedaría

$$t = O(n\log_2(n) + \log_2(n)) = O(n\log_2(n))$$

Por lo que la complejidad uniforme se podría expresar como $\mathrm{O}(\sqrt{t})$, con lo que terminaría siendo **sublineal** en función de la entrada.

Modelo Logarítmico

Considero n primo por ser éste el peor caso. Calcular $\sqrt{n}$ es considerada que cuesta $\mathrm{O}(\log_2(n))$. Luego se deberá iterar para cada primo menor o igual que $\sqrt{n}$, por lo que considero que ciclará $\sqrt{n}$ veces. Para cada iteración, se deberán realizar una comparación y una división entera (en ralidad se calcula el resto, pero es prácticamente lo mismo), costando cada una de ellas $\mathrm{O}(\log_2(n))$.

Luego, la complejidad logarítmica es $\mathrm{O}(\sqrt{n}\log_2(n))$.

El tamaño de entrada es $t = O(n\log_2(n))$ y fácilmente se verifica que es mayor a la complejidad logarítmica. Asique ésta será $\mathrm{O}(t)$, con lo que es **lineal** en función del tamaño de la entrada.

4.4.2. Factorizacion

Modelo Uniforme

Éste algoritmo devuelve una lista de naturales, ordenados de menor a mayor, representando cada uno a un producto de la factorización de n.

Para calcularla, comienza por el menor primo (el 2) y verifica si lo es. De serlo, lo agrega a una lista (utilizada luego para probar si los siguientes valores son primos) y comienza a agregarlo a la lista de factores tantas veces como divida a n. Luego, incrementa el natural a probar y repite el procedimiento hasta que se hallan obtenido todos los divisores primos.

El peor de los casos es cuando n es primo, ya que se deberá llamar a EsPrimo n veces (el único divisor que se encontrará es n). De ser un número compuesto, se llamará a lo sumo n/t veces a ésta función, siendo t el menor primo en la factorización. Además, el bucle interno que verifica cuantas veces divide un primo a n nunca se ejecutará en total (o sea, sumando para todos los primos divisibles) más de $\log_p n$ veces, siendo p el máximo primo en la factorización.

Por lo tanto, como EsPrimo se ejecuta a lo sumo n veces, y en cada llamado cuesta $O(\sqrt{n})$ operaciones, entonces la complejidad de Factorizacion es $O(n\sqrt{n})$.

El tamaño de la entrada será simplemente $t = \log_2(n)$. Notar que $2^{\log_2(n)} = 2^t = n$, por lo que la complejidad uniforme de Factorizacion se puede expresar como

$$O(2^t\sqrt{2^t}) = O(2^{3t/2})$$

con lo que sería **exponencial** en función del tamaño de la entrada.

Modelo Logarítmico

Se supone nuevamente que n es primo por ser el peor caso. Entonces, se deberá iterar $n-1$ veces realizando en cada ciclo:

- Una comparación contra 1 $\rightarrow$ $O(\log_2(n))$.
- Una llamada a EsPrimo $\rightarrow$ $O(\sqrt{n}\log_2(n))$.
- Una suma a p, el cual nunca es mayor a n $\rightarrow$ $O(\log_2(n))$.
- Cuando $p = n$, en el bucle interno se realizan dos divisiones enteras y dos AgregarAtras $\rightarrow$ $O(4\log_2(n)) = O(\log_2(n))$.

Por lo que cada iteración costará la suma de todos los órdenes: $O(\sqrt{n}\log_2(n))$.

Luego, la complejidad de Factorizacion es $O(n\sqrt{n}\log_2(n))$.

De forma equivalente al modelo uniforme, siendo $t = \log_2(n)$ el tamaño de la entrada, la complejidad logarítmica se puede expresar como

$$O(2^t\sqrt{2^t}t) = O(2^{3t/2}t)$$

con lo que ésta sería **exponencial** en función del tamaño de la entrada.

4.4.3. Fuerza Bruta

Las diferencias entre el algoritmo de fuerza bruta y el implementado son principalmente dos en EsPrimo:

- El algoritmo verificará si algún primo menor a n lo divide, en lugar de verificar hasta $\sqrt{n}$.
- No posee una lista de primos menores a n, por lo que deberá verificar para cada natural p desde 2 hasta n si es o no primo, donde a su vez deberá verificar también si algún primo de 2 a $p-1$ lo divide, etc.

Así es como la cantidad de llamadas recursivas será expresada en función de la cantidad de llamadas de los valores anteriores (suponiendo que todos los números de 2 a n son primos). i.e.

- $T(2) = 0$
- $T(3) = T(2) + 1 = 0 + 1 = 1$
- $T(4) = T(2) + T(3) + 2 = 0 + 1 + 2 = 3$
- $T(5) = T(2) + T(3) + T(4) + 3 = 0 + 1 + 3 + 3 = 7$
- $T(6) = T(2) + T(3) + T(4) + T(5) + 4 = 0 + 1 + 3 + 7 + 4 = 15$
- $T(7) = T(2) + T(3) + T(4) + T(5) + T(6) + 5 = 0 + 1 + 3 + 7 + 15 + 5 = 31$
- ...
- $T(n) = \sum_{i=2}^{n-1} T(i) + (n-2)$

El término general de la recursión es

$$T(n) = \sum_{i=2}^{n-1} 2^{i-2} = \sum_{j=0}^{n-3} 2^j = \frac{2^{n-2} - 1}{2 - 1} = 2^{n-2} - 1$$

Demostración por inducción sobre n:

Caso base ($n = 2$):

$$T(2) = 2^{2-2} - 1 = 0$$

Caso inductivo (se cumple para $T(n-1) \implies$ se cumple para $T(n)$):

$$T(n) = \sum_{i=2}^{n-1} T(i) + (n-2) = \sum_{i=2}^{n-2} T(i) + T(n-1) + (n-3+1) =$$

$$\sum_{i=2}^{n-2} T(i) + (n-3) + T(n-1) + 1 = T(n-1) + T(n-1) + 1 = (HI)$$

$$2(2^{n-3} - 1) + 1 = 2^{n-2} - 1$$

□

Por lo tanto la complejidad de EsPrimo por fuerza bruta es (en el peor caso)

$$O(2^{n-2} - 1) = O(2^n)$$

con lo que de Factorizar por fuerza bruta es del orden deO($n2^n$).

4.5. Resultados

Los gráficos a continuación cuentan la cantidad de operaciones de los algoritmos en función de las variables de entrada. Se decidió así en lugar de hacerlos en función del tamaño de la entrada para que sean sencillos de apreciar.

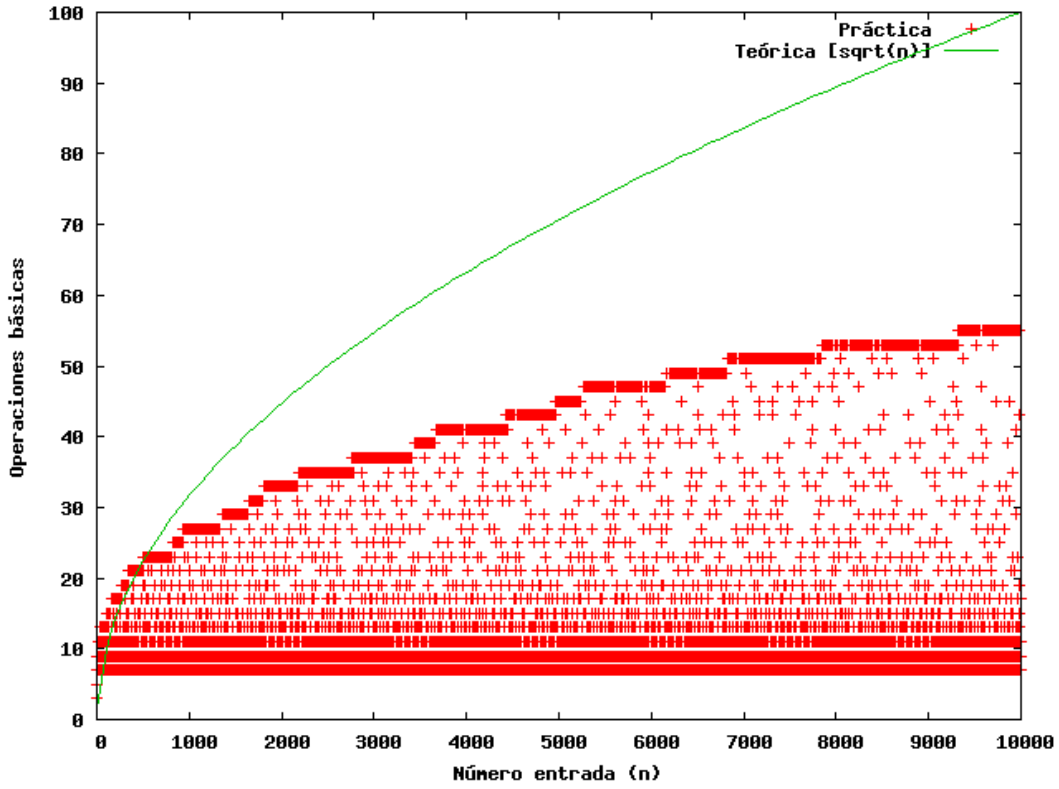

Figura 14: Cantidad de operaciones básicas de EsPrimo en función del número de entrada

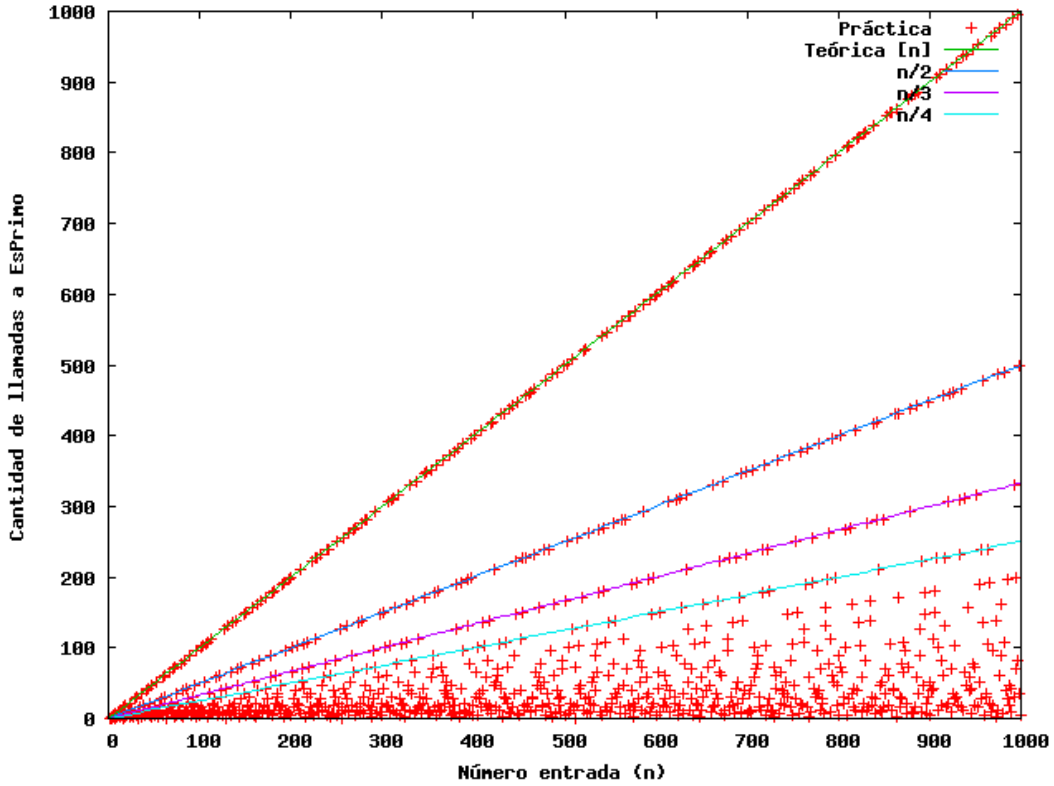

Figura 15: Cantidad de llamadas a la función EsPrimo desde Factorizacion en función del número a factorizar

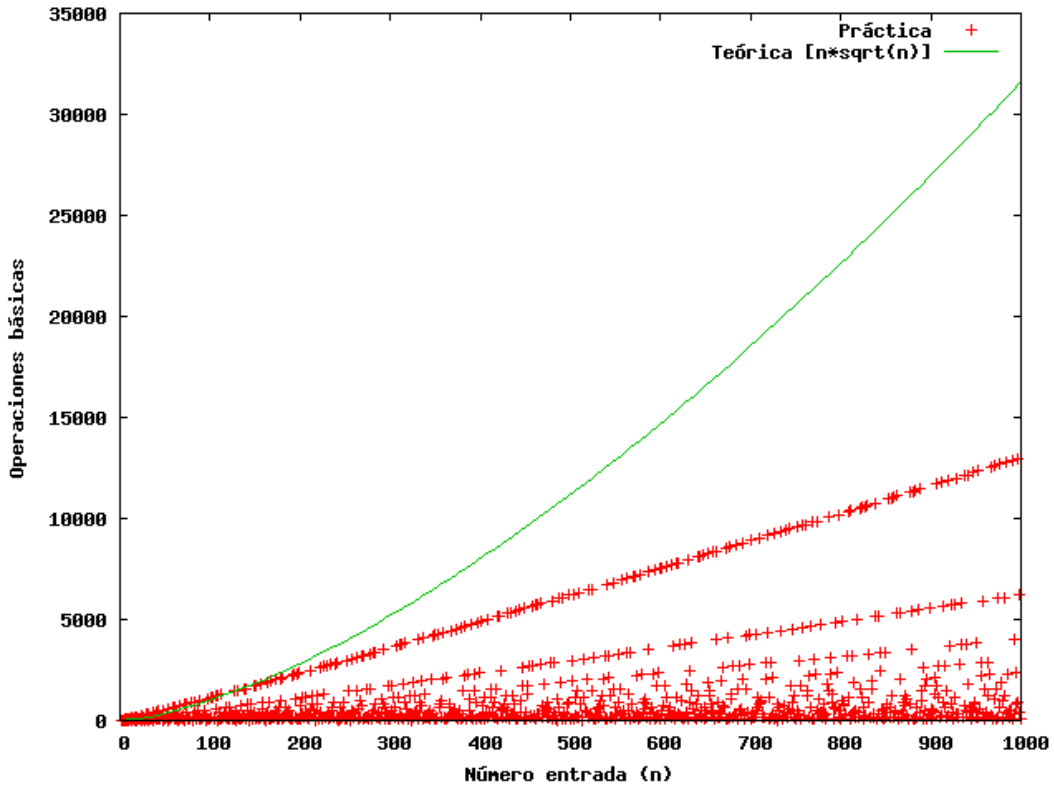

Figura 16: Cantidad de operaciones básicas de Factorizacion en función del número a factorizar

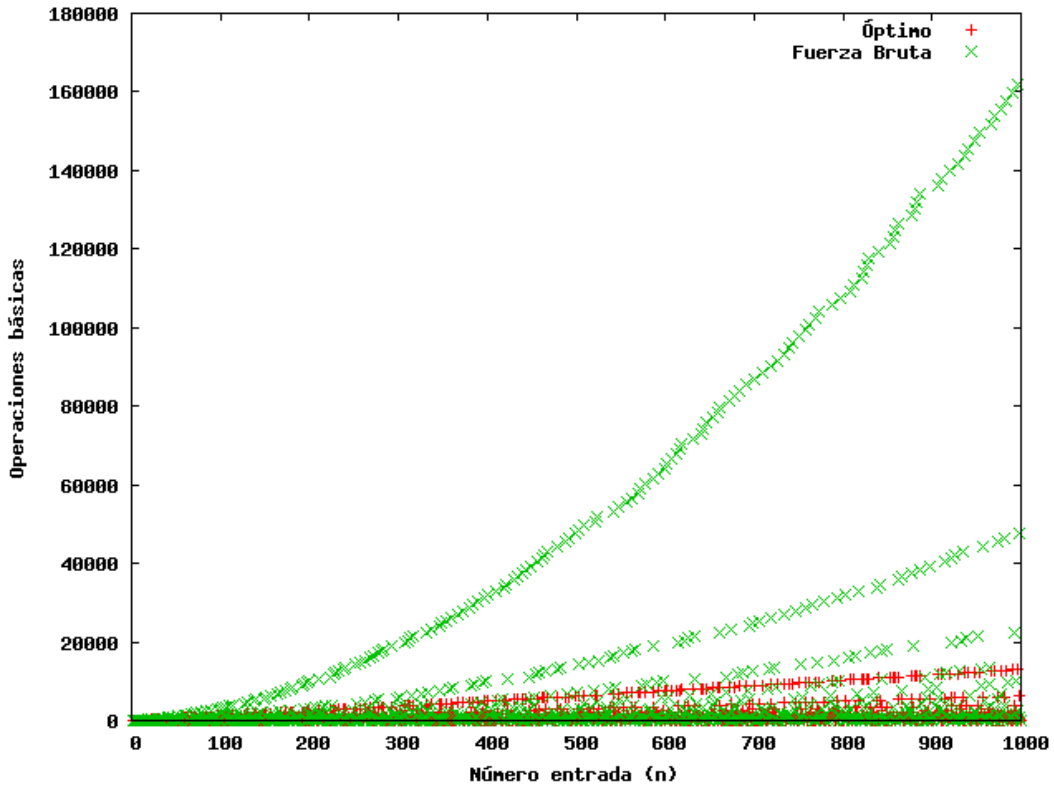

Figura 17: Cantidad de operaciones básicas de Factorizacion mediante el algoritmo óptimo y el de fuerza bruta en función del número a factorizar

4.6. Discusión

En el primer gráfico se puede ver que la cota de $O(\sqrt{n})$ es válida, ya que para cierto valor inicial ésta función siempre es mayor que la cantidad de operaciones reales. Pero puede parecer exagerada. Ésto es perfectamente esperable debido a que en el cálculo de la complejidad se consideró como peor caso de EsPrimo aquel donde **todos** los valores menores a $\sqrt{n}$ eran primos, y por supuesto ésto es imposible (excepto, obviamente, con valores pequeños). De existir una mejor forma para acotar la cantidad de primos menores a $\sqrt{n}$, entonces la complejidad teórica podría ser mejor ajustada.

En el segundo se notan patrones peculiares en la distribución de los valores: ciertas rectas de diferentes pendientes. Después de analizar algunos puntos de cada una se descubrió que:

- La mayor son la cantidad de llamados que se hicieron a EsPrimo para valores que efectivamente eran primos. Es más, los tiempos de **todos** los primos calleron aquí. Dado que la distribución es idéntica a la funcion n, nos demuestra que el análisis teórico fue correcto: si el valor n a factorizar es primo entonces la cantidad de valores consultados por EsPrimo será del orden de n.
- La recta que le sigue a la anterior tiene la mitad de pendiente que la anterior, es decir, sigue la función $n/2$. Los valores para los cuales se hacen esta cantidad de llamadas a EsPrimo es cuando en su factorización aparecen sólo dos productos, y uno de ellos es el 2. Ésto tiene mucho sentido debido a que, al encontrar el primer primo fácilmente (es el primero) el algoritmo deberá correr hasta encontrar al primo restante. Al no ser éste compuesto, la cantidad de llamados a EsPrimo que le restan por hacer será del orden de éste valor. Por lo tanto, la cantidad de llamadas será del orden de $n/2$, lo cual justifica la distribución adoptada.
- La siguiente recta tiene pendiente $n/3$, y similarmente al caso anterior, aquí recaen aquellos valores que en su factorización está el 3 y otro primo.
- La siguiente es $n/4$ y los valores tienen una factorización con el 2 dos veces y luego otro primo.
- etc...

El tercer gráfico se basa en los dos anteriores. El patrón de rectas se sigue notando y se debe a la cantidad de llamados a EsPrimo. La cota teórica es válida pero, debido a que la cota en EsPrimo era elevada y la primera se basa en la segunda, aquí también termina siéndolo.

Por último, en el cuarto gráfico se puede ver que nuesto algoritmo es mucho más eficiente en comparación con el de fuerza bruta. Y era de esperar ya que nuestro algoritmo es polinomial y el de fuerza bruta, exponencial.

5. Referencias

Referencias

[1] Algoritmo de Strassen para la multiplicación de matrices cuadradas:
http://en.wikipedia.org/wiki/Strassen_algorithm

[2] Algoritmo de Coppersmith-Winograd para la multiplicación de matrices cuadradas:
http://en.wikipedia.org/wiki/Coppersmith %E2 %80 %93Winograd_algorithm

[3] Golub & Van Loan, *Matrix Computations*; sección 11.2.5, página 569.