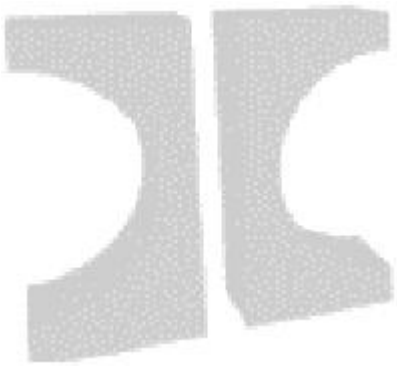


Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación
Algoritmos y estructuras de datos III

Trabajo Práctico no. 1

Integrante	LU	Correo electrónico
González, Emiliano	426/06	xjesse_jamesx@hotmail.com
González, Sergio	481/06	gonzalezsergio2003@yahoo.com.ar
Martínez, Federico	17/06	federicoemartinez@gmail.com
Sainz-Trápaga, Gonzalo	454/06	gonzalo@sainztrapaga.com.ar

Resumen

En el siguiente trabajo se presentan las soluciones a los ejercicios propuestos como primer trabajo práctico de la materia Algoritmos y Estructuras de Datos III. En el mismo se presenta el planteo de cada problema, el desarrollo de un algoritmo para resolverlo, así como también el cálculo de la complejidad del mismo. Por otro lado, se realiza una experimentación para observar el comportamiento de los algoritmos, con el objetivo de contrastar los resultados experimentales con el análisis teórico.

Índice

Parte I
Ejercicio 1

1. Enunciado

Dado un número natural n mayor que 1, encontrar el número primo p que aparece con mayor potencia en la factorización de n. En caso de haber más de un número primo con la mayor potencia, encontrar el mayor de ellos.

2. Desarrollo

La primera idea para resolver el ejercicio fue obtener los primos menores que el número a factorizar (en adelante n) y luego obtener la potencia con la que cada uno lo divide, quedándonos con el de mayor potencia o con el mayor de todos los de máxima potencia. Sin embargo esta solución era costosa, ya que requería obtener primeramente todos aquellos primos que sean menores a n.

Un segundo acercamiento nos permitió salvar esta dificultad, de manera que no fue necesario obtener los primos menores a n explícitamente. El proceso consiste en partir de 2, probar manualmente si 2, 3 y 5 dividen a n y a partir de aquí ciclar generando números de la forma $6 * k + 1$ o $6 * k + 5$ con $k \geq 1$ (se demostró que todos los números primos mayores que 5 tienen esta forma en la demostración 1).

Cuando hallamos algún número primo que divide a n, se lo divide y se guarda en n el cociente de dicha división. Mientras el primo siga dividiendo a n, vamos guardando la potencia con que lo divide. Cuando deja de dividir a n, verificamos si es necesario actualizar dicha potencia, y luego construimos un nuevo candidato.

Utilizando que si un número es compuesto entonces algún primo menor que su raíz cuadrada lo divide (ver demostración 3) no necesitamos ciclar hasta n sino únicamente hasta su raíz cuadrada, o equivalentemente hasta que el primo actual al cuadrado sea menor que n (recordemos que n se actualiza con cada división).

Usamos la segunda opción (elevar el primo al cuadrado en lugar de calcular la raíz cuadrada) ya que en el modelo logarítmico el costo de la raíz cuadrada es mayor que el de la multiplicación (ver Cálculo de Complejidad).

Una desventaja del método propuesto es que hace divisiones por números que no son primos, pero igualmente el costo de estas divisiones es menor que el de ver si el número es primo cada vez.Es importante notar que si un número no es primo, este no puede dividir a n (ver demostración 2). Si no se pudiera asegurar esto, el algoritmo podría fallar.

3. Demostraciones

3.1. Teorema 1

Enunciado:

Sea $p \in \mathbb{Z}$, primo , $p > 5$, entonces $\exists$ k $\in \mathbb{Z} \geq 1$ tal que $p = 6 * k + 1$ o $p = 6 * k + 5$.

Demostración:

Lo demostraremos por absurdo.

Supongamos que $\exists\ p \in \mathbb{Z}$, primo, tal que $p > 5$ y $p \not\equiv 1 \mod 6$ y $p \not\equiv 5 \mod 6$, luego $p = 6 * k + j$ con $j \in 0, 2, 3, 4$

si $j = 0$

$6 * k \equiv 0 \mod 6$, absurdo pues p es primo

si $j = 2$

$6 * k + 2 \equiv 0 \mod 2$, absurdo pues p es primo

si $j = 3$

$6 * k + 3 \equiv 0 \mod 3$, nuevamente absurdo

si $j = 4$

$6 * k + 4 \equiv 0 \mod 2$, también llegamos a un absurdo.

Ergo, si p es primo y $p > 5$, entonces $p = 6 * k + 1$ o $p = 6 * k + 5$.

3.2. Teorema 2

Enunciado:

Sea $k \in \mathbb{Z}$ compuesto, $k > 1$, y sea $n \in \mathbb{Z}$ para todo $p \in \mathbb{Z}$ primo, $p <$ k, $(n : p) = 1$, entonces $n \not\equiv 0 \mod k$

Demostración:

Dado que k es compuesto existen $q_1, \ldots, q_j$ con q_i primo tal que $q_1 * \ldots * q_j = k$

Supongamos que $n \equiv 0 \mod k$,

Entonces como los q_i son primos, vale que n $\equiv 0 \mod q_1 \wedge ... \wedge n \equiv 0 \mod q_j$

Pero sabemos que $q_i < k$ y que por lo tanto $(n : q_i) = 1$

Llegamos entonces a un absurdo que provino de suponer que $n \equiv 0 \mod k$

Luego $n \not\equiv 0 \mod k$, que era lo que queríamos probar.

3.3. Teorema 3

Enunciado:

$k \in \mathbb{Z}$, $k > 1$; k es compuesto $\longleftrightarrow \exists p$ primo tal que $p \leq \sqrt{k}$ y $k \equiv 0 \mod p$

Demostración:

Analizamos por separado las dos implicaciones:

$\leftarrow$) trivial

$\rightarrow$) Como k es compuesto se puede factorizar como $p_1 * ... * p_n$
supongamos que $p_i > \sqrt{k} \forall i \in 1...n$

Entonces

$k = p_1 * ... * p_n > (\sqrt{k})^2 * T$, con $T \geq 1$

$k = p_1 * ... * p_n > \sqrt{k}^2 * T$ pero $k * T > k$

Absurdo, que provino de suponer que $p_i > \sqrt{k}$ $\forall i \in 1...n$.

4. Pseudocódigo

Algoritmo 1 Halla *mejorPrimo* y *mejorPotencia*

```
1:  mejorPrimo←1
2:  mejorPotencia←0
3:  primoActual←generar_candidato()
4:  potenciaActual←0
5:  limite←n
6:  Mientras n≠1 | primoActual*primoActual≤limite hacer
7:    Si primoActual | n entonces
8:      potenciaActual++
9:      n←n/primoActual
10:   Si no
11:     Si potenciaActual≥mejorPotencia entonces
12:       mejorPrimo←primoActual
13:       mejorPotencia←potenciaActual
14:     Fin si
15:     potenciaActual←0
16:     primoActual←generar_candidato()
17:     limite←n
18:   Fin si
19: Fin mientras
20: Si primoActual>l entonces
21:   primoActual←n
22:   potenciaActual←1
23: Fin si
24: Si potenciaActual≥mejorPotencia entonces
25:   mejorPrimo←primoActual
26:   mejorPotencia←potenciaActual
27: Fin si
```

En el pseudocódigo presentado se utiliza una funcion auxiliar:

- *generar_candidato*() es un procedimiento tipo *factory* que conserva un estado interno y genera candidatos a números primos como se describió en el Teorema 1

5. Cálculo de complejidad

En este ejercicio usamos el modelo logarítmico de complejidad ya que la entrada del problema es únicamente un número entero y el grueso de las operaciones involucradas son únicamente sentencias de control de flujo y operaciones aritméticas sobre enteros. En función de esto no sería lógico despreciar el aumento del costo de operar sobre dichos números a medida que la entrada crece.

A partir del pseudocódigo se puede identificar rápidamente el bucle principal donde se realizan la mayoría de las operaciones. Veamos primero el código fuera del bucle.

- *generar_candidato*() tiene un costo $O(log^2 n)$ ya que para casos grandes solo efectúa un chequeo de una variable booleana y una multiplicación de un entero acotado por n.

El resto de las operaciones realizadas antes y después del bucle tienen una complejidad de a lo sumo $O(log\ n)$, y por lo tanto son despreciables respecto del orden predominante que es hasta el momento $O(log^2 n)$ si no tenemos en cuenta al ciclo principal.

Observemos ahora el comportamiento de dicho bucle. Se demostrará a continuación que este bucle se ejecuta a lo sumo $O(\sqrt{n})$ veces, pero por el momento veamos la complejidad del código contenido en el mismo. Si observamos línea por línea el pseudocódigo puede verse que el cuerpo del ciclo es una sucesión de operaciones aritméticas, donde la más costosa es la multiplicación, cuya complejidad es $O(log^2 n)$.

Resta entonces demostrar que el ciclo itera a lo sumo $O(\sqrt{n})$ veces. Si analizamos la guarda del ciclo vemos que se trata de una conjunción booleana. Con que cualquiera de las dos formulaciones booleanas sea falsa, el ciclo termina y el algoritmo se finaliza en tiempo constante. Para que esto ocurra, alcanza con que $n = 1$ o $primoActual^2 > limite$ o equivalentemente $primoActual > \sqrt{limite}$. Represento por $nombre_i$ el valor de la variable *nombre* en la iteración i.

Al entrar al ciclo hay dos casos posibles: o bien $primoActual \mid n$, o no lo hace. Llamemos a estas dos posibilidades caso 1 y caso 2 respectivamente, y analicemos por separado.

- Caso 1: $primoActual \mid n$
  En este caso se divide a n por *primoActual* mientras sea posible. Como $primoActual \geq 2$, resulta sencillamente que la cantidad de veces que puede ejecutarse este bloque es $O(log_2 n)$. Necesariamente después de esa cantidad de iteraciones tendremos $n = 1$ y el ciclo termina.
- Caso 2: $primoActual \nmid n$
  En este otro caso se cumple que $\forall n > N$, $primoActual_{i+1} \geq primoActual_i + 1$. Esto se deduce por construcción del procedimiento *generar_candidatos*() que produce una sucesión de la forma:

  2, 3, 5, 6 * 1 + 1, 6 * 1 + 5, 6 * 2 + 1, 6 * 2 + 5, 6 * 3 + 1, 6 * 3 + 5 ...

  Si además vemos que en este caso $primoActual_{i+1}$ es el término de la sucesión que está inmediatamente a continuación de $primoActual_i$. Por consiguiente *primoActual* crece de forma tal que supera el valor de $\sqrt{limite}$ en $O(\sqrt{limite})$ iteraciones (puesto que $primoActual > 0$ en todo momento, $|primoActual - \sqrt{limite}| < \sqrt{limite}$). Además resulta claro que $limite \leq n$ ya que su valor inicial es n y luego decrece progresivamente.

Finalmente vemos que el caso 1 resulta en a lo sumo $O(log_2 n)$ iteraciones, mientras que el caso 2 resulta en $O(\sqrt{n})$. Por lo tanto la totalidad de iteraciones es:

$$O(log_2 n + \sqrt{n}) = O(\sqrt{n})$$

Y el orden de complejidad del ciclo teniendo en cuenta el costo de cada iteración es por lo tanto:

$$O(log^2(n)\sqrt{n})$$

Siendo n el valor del entero que el algoritmo toma como parámetro. Por otra parte sabemos que el tamaño en memoria de un entero arbitrario n es $t = log_2 n$, por lo tanto, el tamaño de la entrada no es n, sino t:

$$2^t = n \Rightarrow \sqrt{n}\ log^2 n = 2^{t/2}\ t^2$$

Finalmente pudimos demostrar que $T(t) \in O(2^{t/2}\ t^2)$.

Cuando un número es primo, ocurre que el ciclo itera exactamente $\sqrt{n}$ veces, mientras que si el número es compuesto la cantidad de iteraciones es menor, ya que si $n = p_1^{k_1} * m$ donde p_1 es primo y es el menor primo que divide a n, tenemos que el ciclo va a iterar aproximadamente p_1 veces hasta hallar a p_1, luego k_1 veces y por ende como máximo $\sqrt{m} - p_1$ veces.

En particular si el número n es de la forma p^k con p primo, el ciclo itera hasta p y luego itera k veces, y $k = log_p n$ por lo que la cantidad de iteraciones va a ser logarítmica. Si lo pensamos en función del tamaño de la entrada, si n es primo se producen $O(2^{t/2})$ iteraciones, mientras que si el número es una potencia de un primo, el número de iteraciones es $O(t)$.

6. Análisis Experimental

6.1. Experiencias realizadas

Para el análisis del algoritmo decidimos medir tanto operaciones como tiempo.

Primero medimos dichas variables para los números entre 2 y 100000 para observar un patrón de comportamiento. A partir de esta experiencia, decidimos realizar otras dos, donde separamos números primos en una de ellas, y en la otra tomamos las potencias de un número primo (en particular de 7). Esto lo hicimos por considerar que el peor caso del algoritmo es precisamente cuando el número es primo, mientras que el mejor caso es cuando el número es potencia de un primo.

Por otro lado realizamos experiencias similares pero teniendo en cuenta ya no el número, sino la cantidad de bits de su representación binaria, es decir teniendo en cuenta el tamaño de la entrada. Para estas experiencias tomamos distintos números pero medimos la cantidad de operaciones y el tiempo en función de $\lfloor log_2(n) \rfloor + 1$.

Para medir los tiempos se utilizó una clase basada en $QueryPerformanceCounter$ de Win32. Para contar operaciones, el procedimiento se basó en declarar una variable global e ir incrementándola en base a las operaciones que se realizaban línea a línea. Este procedimiento se aplicó en los tres ejercicios.

6.2. Gráficos

6.2.1. Gráficos en funcion de n

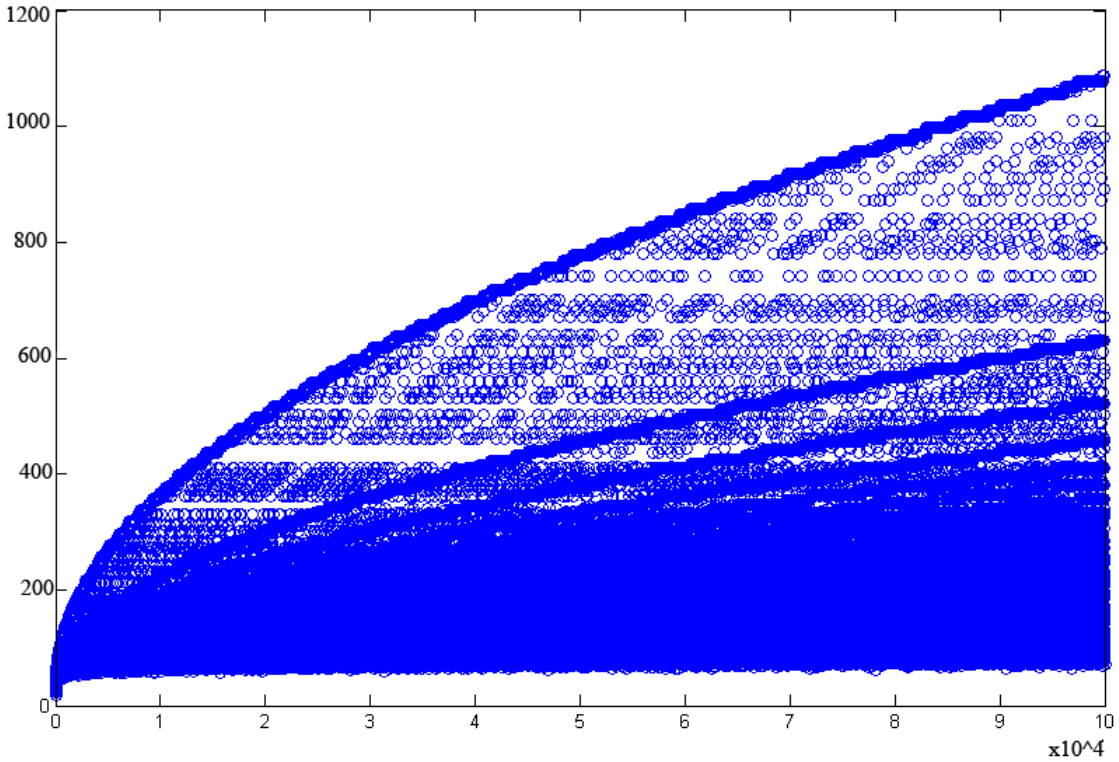


Figura 1: Cantidad de operaciones en función de n

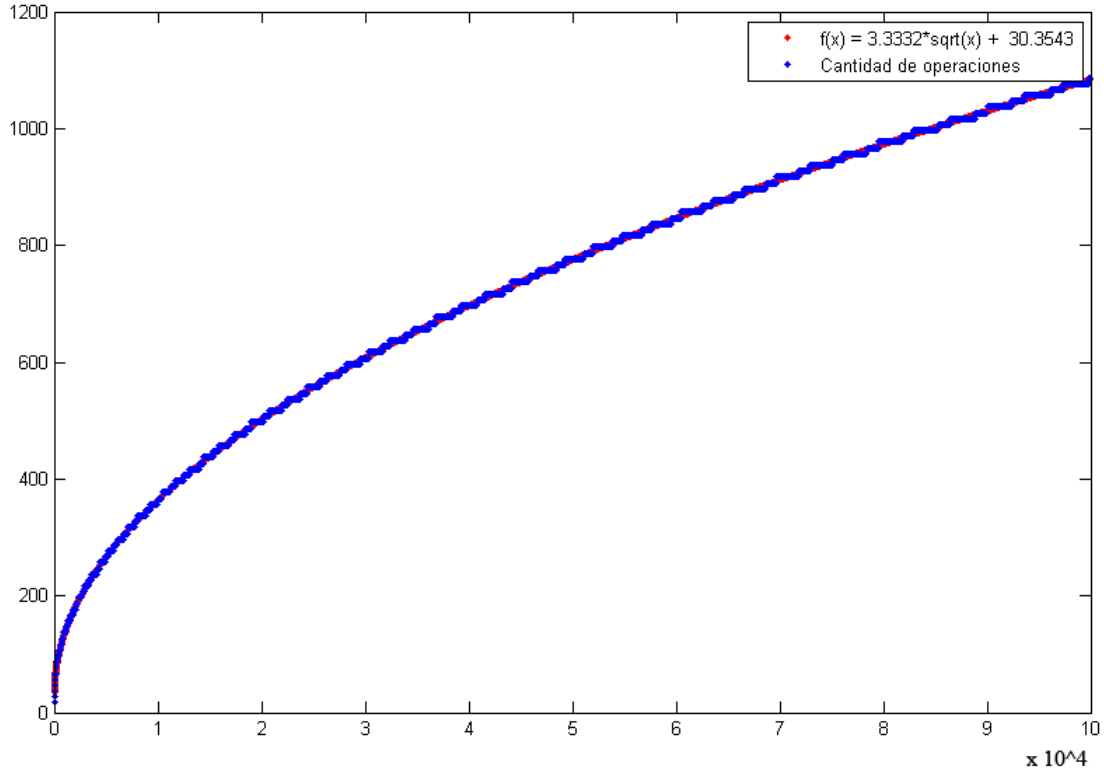


Figura 2: Cantidad de operaciones en función del n para n primo

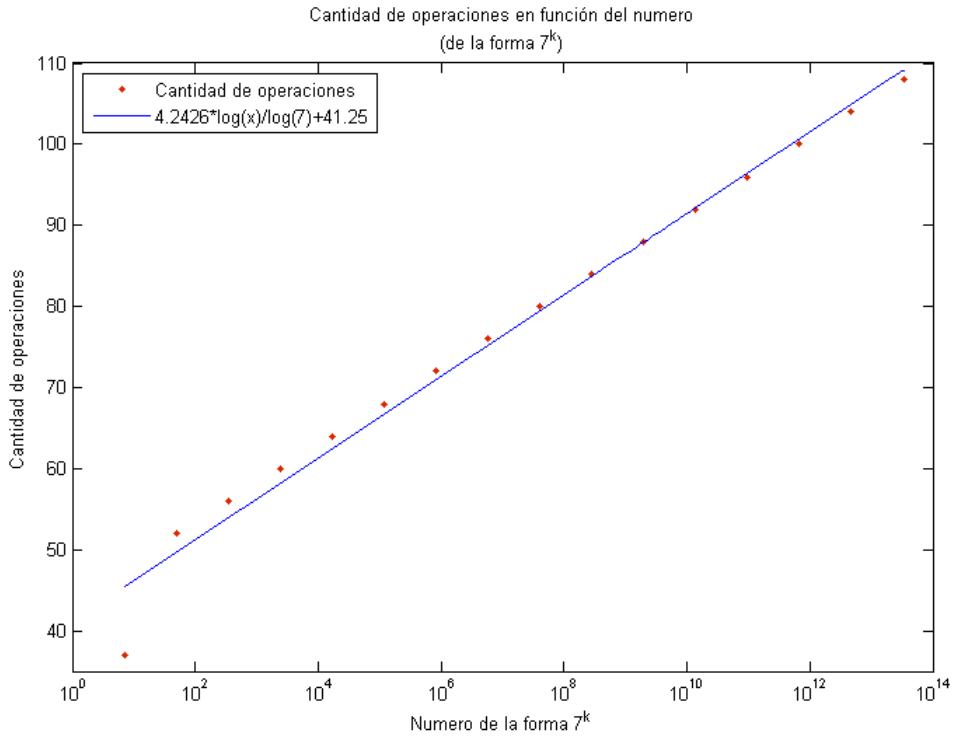


Figura 3: Cantidad de operaciones en función de n para n de la forma 7^k

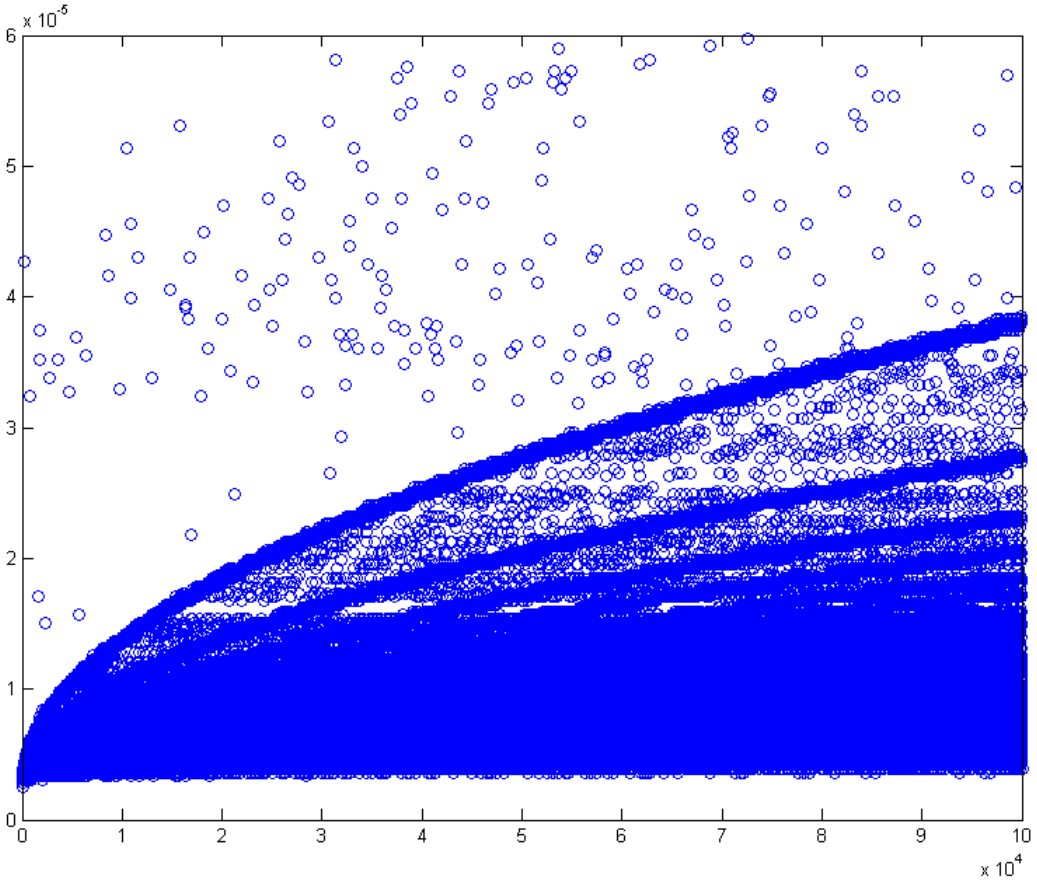


Figura 4: Tiempo en función de n

6.2.2. Gráficos en funcion del tamaño de la entrada

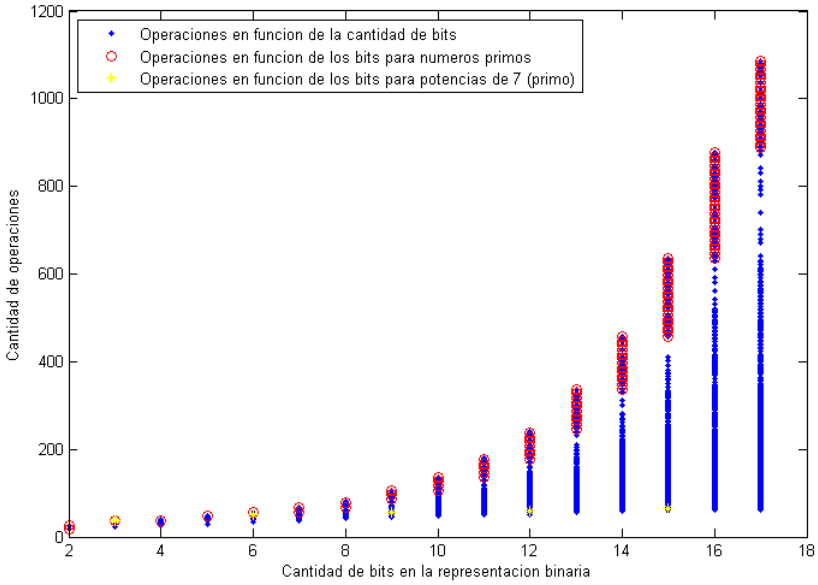


Figura 5: Cantidad de operaciones en función del tamaño de la entrada

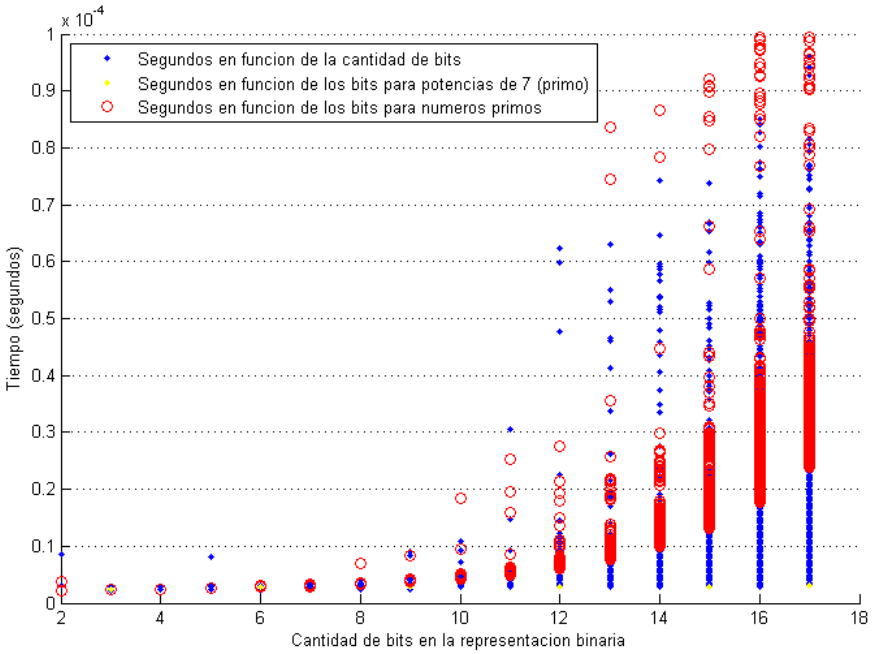


Figura 6: Tiempo en función del tamaño de la entrada

7. Discusión

En los gráficos pudimos observar lo que el análisis teórico nos anunció.

En la figura 1, se ve claramente como existen diversos patrones de comportamiento. Así el techo es un función del tipo raíz cuadrada, mientras que debajo se encuentran otras funciones de orden logarítmico. Esta situación se ve mas claro en los gráficos donde sólo están los números primos. En ambos casos pudimos, mediante cuadrados mínimos, encontrar una función que se asemeje al comportamiento de las muestras, lo cual refuerza nuestra idea de mejor y peor caso.

El gráfico en función del tiempo muestra un comportamiento muy similar al gráfico de cantidad de operaciones, pero presenta *outliers* que se pueden explicar por el hecho de que tomar tiempos esta sujeto a un error de medición producto del uso del sistema de pruebas por parte de otros procesos.

A la hora de analizar los gráficos en función del tamaño de la entrada esperamos ver g'raficos con forma exponencial y fue exactamente eso lo que obtuvimos. En el caso de cantidad de operaciones, se mantuvo lo observado en los otros gráficos, es decir que en general los primos requieren más operaciones que el resto de los números, mientras que las potencias de 7 requieren muchas menos.

Finalmente la figura 6 mantiene la tendencia, y nuevamente presenta *outliers* propios de los factores externos que intervienen en la medición de tiempos.

A modo de conclusión podemos afirmar que la experimentación empírica validó nuestros resultados teóricos.

Parte II
Ejercicio 2

1. Enunciado

La empresa Musimundo cuenta con una serie de sucursales repartidas por el país. Recientemente ha decidido cerrar su sucursal de Iruya y llevarse toda la mercadería al depósito central. Debido al difícil acceso ha dispuesto sólo un camión para llevar toda la mercadería. Sin embargo, es posible que no todo el material pueda ser llevado en un sólo viaje del camión débido a las restricciones de carga, por lo que la parte que no entre en el camión será vendida a valores despreciables el día del cierre.

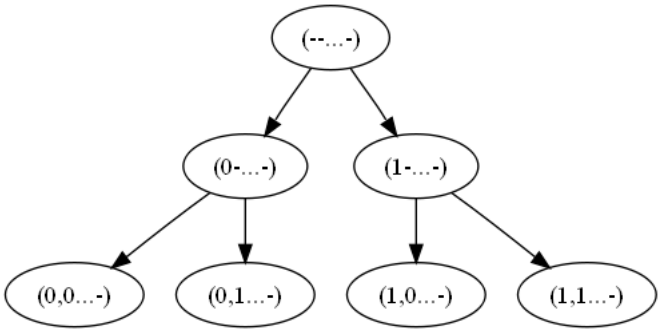
Dada la capacidad de carga máxima P del camión y una lista de los productos del local conteniendo el valor v_i y peso p_i de cada uno, encontrar la lista de productos de mayor valor que sea posible llevar en el camión sin que el peso total de la lista supere la carga máxima. El valor de una lista de productos se calcula como la suma de los valores de los productos involucrados. En caso de haber varias listas con el mismo valor máximo, encuentre cualquiera de ellas. Realice un algoritmo para resolver el problema usando la técnica de backtracking.

2. Desarrollo

Dado que el problema debía ser resuelto con backtracking, buscamos la forma de aplicar dicha técnica algorítmica para su resolución. Básicamente la idea es formar todos los subconjuntos de cosas para encontrar aquel que maximice el valor total, sin exceder la capacidad de carga del camión.

A medida que se va armando una solución, son descartadas aquellas cuyo peso excede la capacidad del camión, y de esta manera se va podando el árbol dejando solo posibles soluciones (por eso es backtracking y no fuerza bruta).

De esta manera, se construye el siguiente árbol de recursión:



En la raíz del árbol todavia no se determino para ningun elemento, si se lleva o no. Luego, en cada nodo se va construyendo una $n-upla$ donde un 0 en la posición i indica que el elemento i no es parte de esta solución, mientras que un 1 indica que el elemento está incluído en la solución. Las hojas de este árbol son las soluciones posibles. Como cada elemento puede estar o no estar, se tienen dos posibilidades para cada uno (llevarlo o no). Esto implica que la cantidad posible de soluciones si se tienen n elementos es 2^n.

El algoritmo implementado toma como parámetros el camión (cosas, cuantas cosas y capacidad) y dos soluciones posibles (una en donde está guardada la mejor encontrada hasta el momento y otra que guarda la que se va construyendo). Además se lleva un índice que indica sobre que objeto se está decidiendo si conveniente o no llevarlo y el valor óptimo que puede lograr una solución (se explica a continuación).

Básicamente una llamada a la función consiste en verificar si se agrega o no a la solución el elemento apuntado por *indice*. Si es así (es decir que al agregarlo, el peso total no excede el peso máximo), se agrega a la solución candidata y se continúa con la siguiente llamada recursiva. Luego se realiza otra llamada recursiva que consiste en no agregar el elemento apuntado por índice. Cuando se encuentra una solución mejor a la actualmente óptima, ésta es reemplazada. Repitiendo este proceso se pueden observar todas las hojas del árbol, para quedarse con la mejor.

Ademas de la restricción propia del problema, buscamos definir otras que nos permitieran efectuar mejores podas. Asi una primera idea fue ordenar el arreglo en orden creciente de pesos. De esta forma, si durante la construcción de una solución, llegamos a que el elemento k no se puede agregar, sabemos que ningun elemento entre $k+1$ y n se va a poder agregar, porque su peso es mayor que el de k, y por ende mayor que la capacidad disponible.

Otra idea que tuvimos consistió en preguntar durante la contrucción de una solución si tiene sentido que el algoritmo siga bajando por la rama que se deriva de este punto. Esto lo hacemos de la siguiente manera: si considero la suma de los valores de los elementos que me quedan por visitar y este valor sumado al del candidato actual no me da un valor más alto que el de la mejor solucíon hasta el momento, no tiene sentido que baje porque no voy a lograr una solución con valor mayor. En un principio, para hacer esto se nos ocurrió hacer la suma en cada llamada. Sin embargo este cálculo era costoso, y por lo tanto buscamos alguna alternativa mejor. La idea entonces fue antes de empezar a trabajar con el arreglo de cosas, sumar el valor de todos los elementos. Este número es el máximo valor que puedo lograr con un conjunto de cosas. Cada vez que hago una llamada calculo si el valor de mi candidato actual más el máximo me sirve para mejorar mi solución. Por otro lado, cuando hago una llamada sin incluir a un elemento, resto al máximo posible el valor de dicho elemento.

Veamos un ejemplo: Tengo un conjunto de cosas tal que sus valores son: 1,2,3,4. El máximo valor que podría llevar es 10. Cuando estoy armando una solución que excluye al valor 1, el máximo es 9. Si ademas quito al 2, el máximo es 7, etc. Consideramos que si bien estas podas no nos mejorarían el orden de complejidad, si nos permitirían en general observar un mejor desempeño. Es importante notar que por como tenemos que devolver la solución es necesario mapear de alguna manera los índices de los elementos luego de ordenarse con los del arreglo original. Esto agrega un *overhead* a la poda que debe tenerse en cuenta.

Para implementar el algoritmo se definieron los tipos *Cosa*, *Camion* y *SolucionPosible* con la finalidad de aportar más claridad al mismo.

3. Pseudocódigo

SolucionPosible: tupla$< cantCosas, guardo : [bool], valor, costo >$
Camion: tupla$< cantCosas, capacidad, cosas : [Cosa] >$
Cosa: tupla$< costo, valor >$
cosas $= \{a_1, ..., a_{cant}\}$

Algoritmo 2 Halla la solución óptima al problema del camion

```
1: s ← SolucionPosible(cant cosas)
2: ordenar arreglo de cosas {mediante merge sort}
3: valorMaximo ← ∑_{t=0}^{cantcosas} (cosa_i)_valor
4: camionAux(s,c,0,mejorSol,valorMaximo)
```

Algoritmo 3 camionAux: Halla la solución óptima *mejorSol* al problema del camion

```
1: Si probé con las cant cosas & valor(candActual) > valor(mejorSol) entonces
2:    mejorSol ← candActual
3:    terminar
4: Fin si
5: Si el valor del actual + valorMaximo ≤ el valor de la mejor solución hasta el momento entonces
6:    terminar
7: Si no
8:    Si no probé las cant cosas entonces
9:       Si no me paso del peso máximo agregando a_i a candActual entonces
10:         agregar(a_i, candActual)
11:         camionAux(candActual, cosas, capacidad, i+1, cant, mejorSol)
12:         sacar(a_i, candActual)
13:      Si no
14:         { como los demas pesan mas, no puedo agregar a mas nadie}
15:         Si valor(candActual) > valor(mejorSol) entonces
16:            mejorSol ← candActual
17:         Fin si
18:         terminar
19:      Fin si
20:      camionAux(candActual, cosas, capacidad, i+1, cant, mejorSol)
21:   Fin si
22: Fin si
```

Por una cuestión de claridad (y para no desviar la atención del algoritmo que realmente resuelve el problema), se excluyó del pseudocódigo la traducción entre los índices originales, y los índices en que quedan los elementos luego de ordenarse (lo cual necesitamos para devolver la solución en el formato pedido). Este proceso consiste en recorrer un diccionario (sobre arreglo) donde está mapeado para cada elemento la posición donde quedó luego de ordenarse.

4. Cálculo de complejidad

Para este ejercicio decidimos usar el modelo uniforme, ya que consideramos que lo que hace al núcleo del problema es la cantidad de cosas a llevar. Por esa razón no nos parece desacertado considerar que el peso y el valor de las cosas estan acotados. Asimismo consideraremos que el tamaño de la entrada es la cantidad de cosas entre las cuales elegir (es decir la cantidad de items de la sucursal), en adelante N.

Antes de llamar a la función que hace backtracking, el algoritmo ordena el arreglo con *mergesort* en $O(N\ log(N))$ y luego suma todos los valores del mismo en $O(N)$. Al finalizar el algoritmo, realizamos una traducción entre los indices originales de las cosas y su posición luego de ordenar que tiene un costo lineal, es decir $O(N)$. Veremos a continuación que como el orden de la función principal es mayor que éstos, la complejidad total del algoritmo no se ve afectada.

Observemos que dado un N, si llamamos n a la cantidad de elementos que quedan por procesar (es decir por decidir que hacer con ellos) se observa que $T(0) = N + 1$, pues hay que preguntar si encontré una mejor solución y hay que copiar la solución actual. Si no quedan más elementos que mirar y tengo que copiar la nueva solución. Esta copia tiene como costo la cantidad de elementos entre los cuales hay que elegir.

Además, si se observa el pseudocódigo, se puede ver que para un n dado a lo sumo se hacen dos llamadas recursivas con $n - 1$ elementos a procesar. A este costo se le agrega una constante que llamaremos k, por lo tanto podemos deducir las siguientes ecuaciones de recurrencia:

$$T(0) = N + 1$$
$$T(n) = 2 * T(n - 1) + k$$

donde k viene de las operaciones que se realizan dentro de cada llamada, que son todas $O(1)$.

En algun caso, podría ocurrir que por accion de la poda, T(n) = N, ya que si la solución que se está armando no puede incorporar ningún objeto más (encontró que el elemento del índice actual es mayor al peso, y como están ordenados todos los siguientes pesan más) pero tiene un valor mayor que el de la mejor hasta el momento, hay que actualizar. Sin embargo, veremos a continuación que en el caso donde esto no ocurre (el descripto por las ecuaciones) el orden es mayor que N.

Proponemos que:

$$T(n) = N * 2^n + k * (2^n - 1) + 2^n$$

Lo podemos demostrar por inducción:

Para $n = 0$:

$$T(0) = N + 1 = N * 2^0 + k * (2^0 - 1) + 2^0$$

supongamos que vale para n, veamos que vale para $n + 1$:

$$T(n + 1) = 2T(n) + k$$

usando la hipótesis inductiva:

$$T(n + 1) = 2(N * 2^n + k * (2^n - 1) + 2^n) + k$$

$$T(n+1) = N * 2^{n+1} + k * (2^{n+1} - 2) + 2^{n+1} + k$$

$$T(n+1) = N * 2^{n+1} + k * (2^{n+1} - 1) + 2^{n+1}$$

Que era lo que queríamos ver.

Ahora bien como $T(n) = N * 2^n + k * (2^n - 1) + 2^n$ y además sabemos que n es del orden de N, tenemos que $T(n) \in O(N * 2^N)$.

Luego el algoritmo tiene un orden exponencial en función del tamaño de la entrada, aún pese a las podas; y esto es de esperarse ya que las podas funcionan únicamente en algunos casos, mientras que en otros no ayudan en nada. Otra forma de ver el orden es considerar el funcionamiento de algoritmo: la idea es revisar cada solución posible, y sabemos que estas son 2^N. En el peor de los casos, tenemos que recorrerlas todas y además actualizar la mejor solución cada vez. Como esto último se realiza en a $O(N)$, el orden final del algoritmo resulta ser $O(2^N * N)$.

Si el algoritmo solo desecha un camino cuando no puede agregar a un elemento porque se excede del peso, el peor caso se da cuando puede poner a todos los elementos, ya que llegará a cada hoja. Observando las podas que implementamos, uno esperaría que en general el algoritmo se comporte bien, incluso en el caso antes mencionado. Ahora supongamos que tenemos un conjunto $a_1, a_2, ...a_N$ donde el peso y el valor de $a_i = 1 \ \forall i \in 1...N-1$ y el peso y valor de $a_N = N$ y que la capacidad del camión es N.

En este caso ninguna de nuestras podas va a funcionar, puesto que la poda por pesos baja siempre hasta el anteúltimo nivel, salvo en el caso donde no agregué a ningún a_i con $i < N$. Tampoco funciona ver cual es el máximo que puedo armar porque hasta que no llegue al caso donde solo pongo a a_N el valor máximo que puedo armar siempre es mayor que N, pero las soluciones que arme siempre valen menos que N. Por ende en este caso se recorren prácticamente todas las ramas.

5. Análisis experimental

5.1. Experiencias realizadas

Para probar el comportamiento del algoritmo medimos tiempos y cantidad de operaciones en función del tamaño de la entrada, es decir de la cantidad de posibles elementos a llevar. A modo de ver si nuestras podas representaban una mejora en la práctica contrastamos los resultados entre el algoritmo sin podas, y el algoritmo con podas.

Para hacer las pruebas generamos casos donde la cantidad de elementos era creciente y la composicíon del conjunto de cosas era aleatoria. El peso que el camión puede cargar se fijó en 200 y los valores y pesos de las cosas siguen una distribución uniforme (0,100). Ademas se analizó el comportamiento del algoritmo en aquellas situaciones que consideramos como peores casos. En los gráficos, se denomina algoritmo sin poda a la versión que no ordena el arreglo, ni observa la suma potencial del camino.

5.2. Gráficos

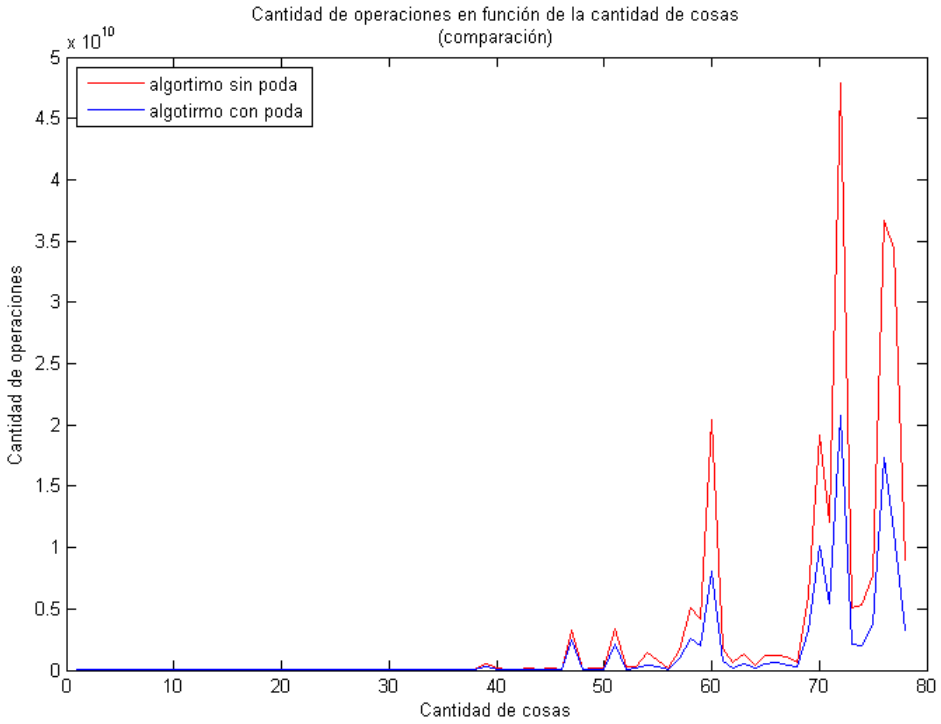


Figura 7: Cantidad de operaciones en función de la cantidad de cosas (peso y valor aleatorios con distribución uniforme)

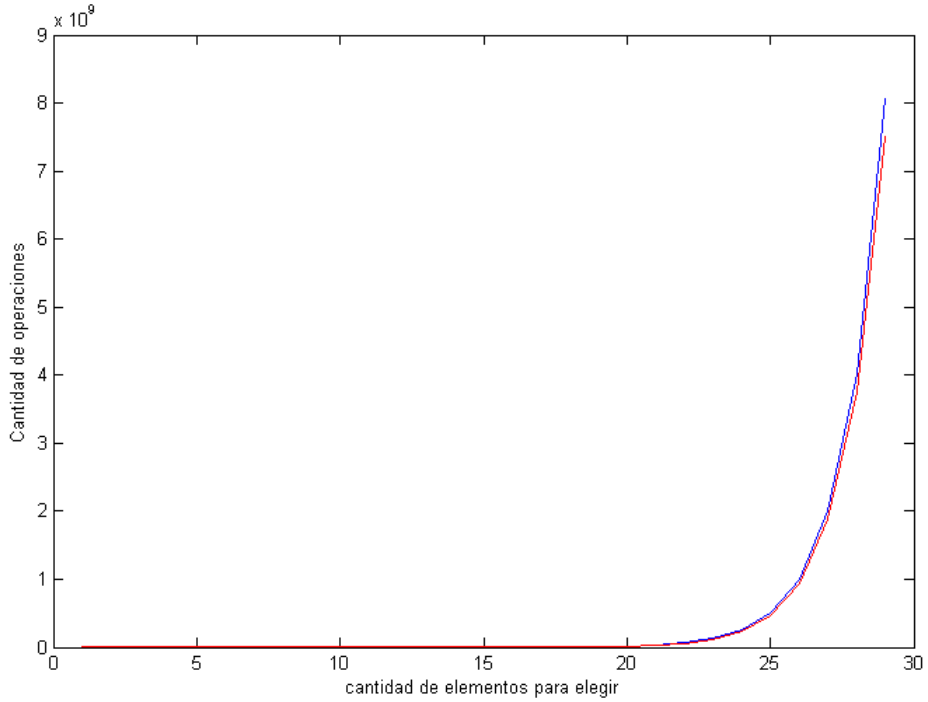


Figura 8: Cantidad de operaciones en función de la cantidad de elementos, peor caso para la poda

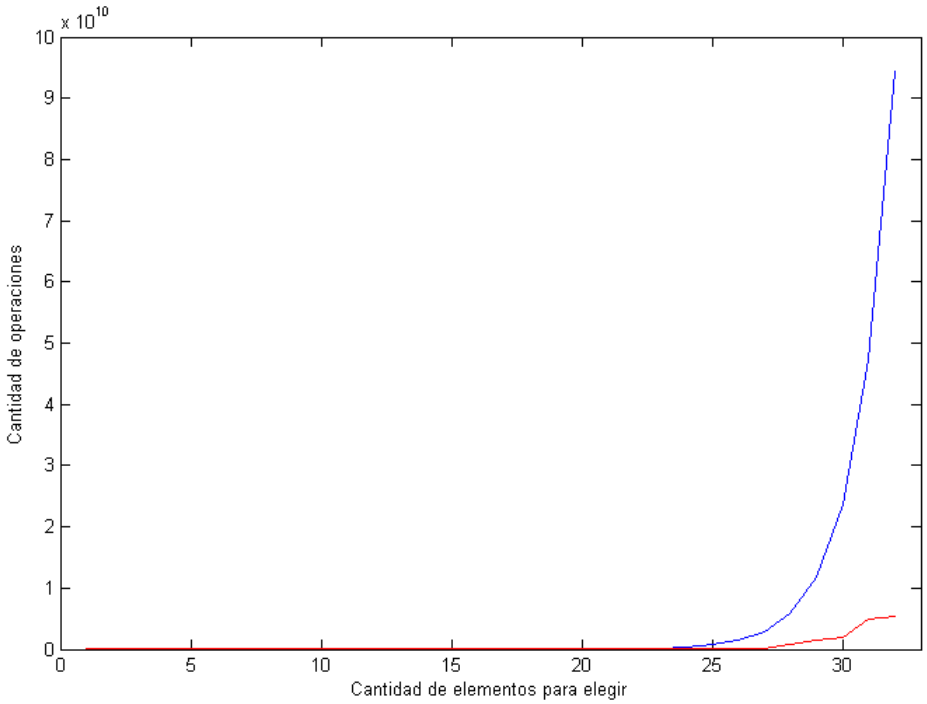


Figura 9: Operaciones en función de la cantidad de cosas, peor caso sin podas

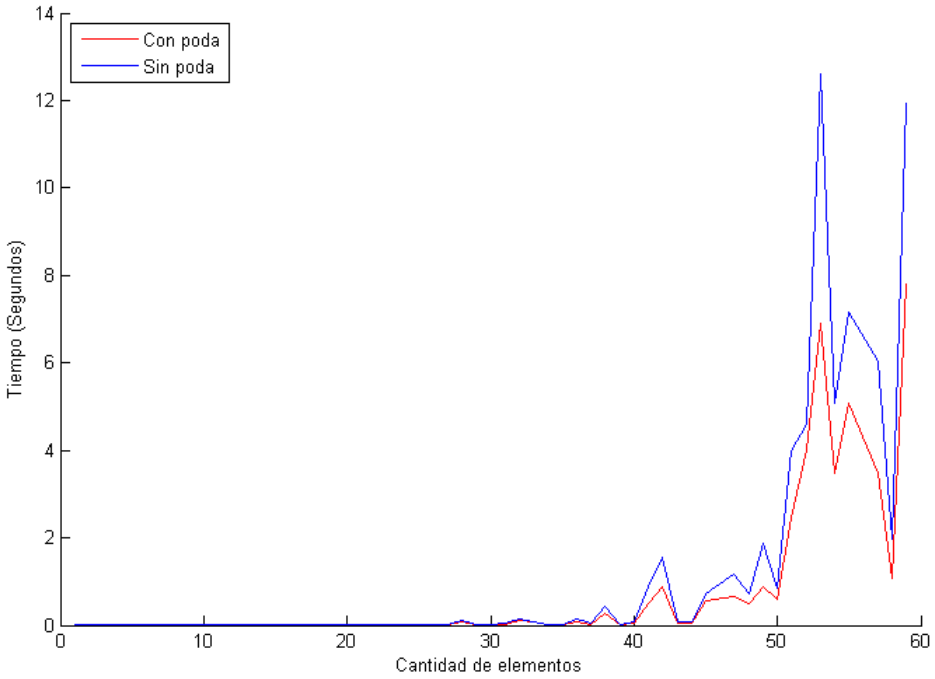


Figura 10: Tiempo en función de la cantidad de cosas, casos aleatorios

6. Discusión

En los gráficos pudimos observar el comportamiento exponencial del algoritmo. En las corridas de casos generales, se observó un mejor comportamiento del algoritmo que implementaba las diversas podas, lo cual es esperable, puesto que uno tendería a creer que algunas ramas se van a podar necesariamente si la configuración de las cosas es aleatoria.

El algoritmo con podas se comportó mucho mejor que el algoritmo sin podas en el peor caso de este último: esto se explica porque en este caso la primer mejor solución que se arma es la que es óptima, y por eso no baja por ninguna rama, ya que ninguna tiene el potencial de superarla.

En el peor caso de nuestro algoritmo con podas sí se observa que el comportamiento es peor que el del algoritmo común y esto se explica si tenemos en cuenta que en este caso no se hacen podas efectivas y sin embargo tenemos el *overhead* de ordenar el arreglo y hacer la traducción de los índices para poder reconstruir los valores que se almacenan en el archivo de salida.

El gráfico de tiempos muestra la misma tendencia que el de cantidad de operaciones, y por tanto no merece un análisis adicional.

A modo de conclusión podemos decir que se validaron las hipótesis planteadas durante el análisis teórico.

Parte III
Ejercicio 3

1. Enunciado

Dado un arreglo de n elementos en el que hay un elemento que aparece más de la mitad de las veces, encontrar la moda, es decir, el valor que aparece más veces.

2. Desarrollo

La primera idea para abordar este problema fue la de ordenar el arreglo y contar la cantidad de apariciones guardando el elemento con más apariciones. Esta primera idea no tuvo mucha aceptación ya que se observó que el orden requerido para ordenar dicho arreglo era $O(n * log\ n)$ y pensamos que debíamos buscar un orden mejor.

Una mejora a este procedimiento fue notar que la moda en un arreglo con las características dadas por el problema coincide con el elemento $n/2$ del arreglo ordenado. Por lo tanto, si se ordena el arreglo, en vez de contar apariciones basta con tomar el elemento $n/2$ para obtener la moda. Sin embargo, esta mejora también requiere que el arreglo sea ordenado, y por lo tanto, al igual que la idea anterior, quedó descartado por tener un orden de complejidad demasiado alto.

Investigando sobre el tema se encontró el algoritmo de Loyd-Pratt-Rivest-Tarjan, que permite encontrar la mediana (mediana en el sentido de elemento del medio, que si el arreglo tiene una cantidad impar de elementos coincide con la mediana estadística, si no, no es exactamente el mismo concepto; pero lo usamos para abreviar) de un arreglo en orden lineal sin necesidad de ordenarlo.

El algoritmo se basa en el uso de un pivote para partir el arreglo y quedarse solo con la parte donde se puede encontrar la mediana. Es necesario para tener orden lineal poder encontrar un buen pivote en un orden a lo sumo lineal. El algoritmo resuelve esto partiendo el arreglo principal en arreglos de 5 elementos, tomando la mediana de éstos y luego tomando una seudomediana a partir de dicha "mediana". Se puede demostrar que este procedimiento permite encontrar la mediana en un orden lineal. Esta solución se descartó ya que su implementación era bastante complicada, y la demostración del orden de complejidad también lo era.

Se buscó entonces otra forma de obtener un orden lineal aprovechando que la frecuencia de la moda era mayor a la mitad. La idea que tuvimos consiste básicamente en recorrer el arreglo sirviéndose de dos índices. Si se encuentran dos elementos iguales, se incrementa uno de los índices (en adelante j) y se deja inmóvil al otro (en adelante i) en la posición donde se encuentra. Si se encuentran dos elementos diferentes, estos son tachados (para esto se utiliza una marca sobre un arreglo de posiciones booleanas).

Luego se avanza i hasta que llegue a una posición sin tachar y se hace lo propio con j hasta llegar a una posición sin tachar pero que también sea mayor que i. El ciclo termina cuando j excede el límite del arreglo. Se devuelve entonces el valor donde quedó parado el índice i. En cada paso el algoritmo "tacha" dos elementos que no son la moda, o uno que es la moda y otro que no. De esta forma solo terminan sobreviviendo algunos elementos de la moda. Al terminar el ciclo, los elementos sin tachar desde i hasta el final del arreglo son iguales. Entonces podrían o bien ser moda o ser elementos distintos a la moda. Sin embargo, esto último no puede ocurrir ya que la frecuencia de la moda es como mínimo $n/2 + 1$, entonces que eso ocurra implicaría que taché por lo menos $2 * (n/2 + 1)$ elementos pero $2 * (n/2 + 1) > n$.

El siguiente es un ejemplo de la aplicación del algoritmo:

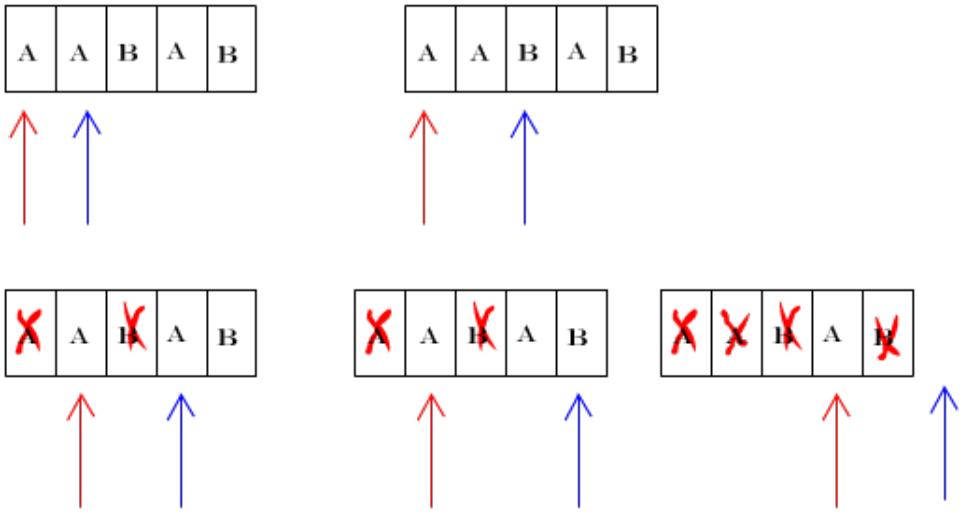


Figura 11: Ejemplo de la aplicación del algoritmo

En el ejemplo, al principio como estamos parados en dos elementos iguales, solo se avanza el índice de adelante. Como ahora son diferentes, se tachan y se avanzan los índices. Nuevamente son iguales por lo que sólo se avanza el de adelante. Se tacha nuevamente un par de elementos, y al avanzar el índice de adelante se termina el arreglo. La moda es entonces el elemento donde está parado el índice de atrás. Este algoritmo fue finalmente el que se adoptó como solución ya que nos garantizaba el orden lineal buscado y su implementación era simple.

3. Pseudocódigo

Algoritmo 4 Halla la moda *moda* del arreglo a

```
1: indiceDeAtras ← 0
2: indiceDeAdelante ← 0
3: tachados: [bool] {los elementos de tachados inicializan en false}
4: Mientras indiceDeAdelante < tamanio(a) hacer
5:    Si a_indiceDeAtras ≠ a_indiceDeAdelante entonces
6:       tachados ← tachar ambos elementos //donde: tachar i es asignar con true en el elemento iesimo de tachados
7:       Mientras (indiceDeAtras < tamanio(a) & (indiceDeAtras no fue tachado) hacer
8:          indiceDeAtras ← indiceDeAtras + 1
9:       Fin mientras
10:      Mientras indiceDeAdelante < tamanio(a) & (indiceDeAdelante no fue tachado | indiceDeAdelante ≤ indiceDeAtras)
         hacer
11:         indiceDeAdelante ← indiceDeAdelante + 1
12:      Fin mientras
13:   Si no
14:      indiceDeAdelante ← indiceDeAdelante + 1
15:   Fin si
16: Fin mientras
17: moda← a_indiceDeAtras
```

4. Cálculo de complejidad

Para hacer el cálculo de complejidad utilizamos el modelo uniforme pues consideramos, al igual que con el ejercicio anterior, que lo fundamental a la hora de estudiar el orden pasa por la cantidad de elementos a estudiar, y no por los valores de los mismos. Por ende es razonable asumir que el costo de operar con los números propiamente dichos es constante.

Considerando esto, afirmamos que si el arreglo tiene n elementos, el tamaño de la entrada es n.
Pensando en el comportamiento que tiene el algoritmo, vemos que siempre recorremos el arreglo hacia adelante y nunca volvemos hacia atrás. Es por esta razón que si bien hay ciclos anidados (dentro del de la línea 4 tenemos al de la línea 7 y el de la línea 10) el algoritmo es $O(n)$.

El ciclo de la línea 7 usa indiceDeAtras para recorrer el arreglo, ahora la guarda del ciclo es:

indiceDeAtras < tamanio(a) & (indiceDeAtras no fue tachado)

Entonces podemos ver que este ciclo, dado que i no se reinicia, solo recorre el arreglo a lo sumo una vez en todos los ciclos del de la línea 4. Es decir, si el ciclo principal hace k iteraciones y llamamos $iterAtras_t$ al numero de iteraciones del ciclo de la línea 7 en la iteración t del ciclo principal, vale que:

$$\sum_{t=1}^{k} iterAtras_t \leq n$$

ya que en el momento en que i se hace n el ciclo interno no itera más.

La situación del ciclo de la línea 10 es similar, en este caso la guarda es:

indiceDeAdelante < tamanio(a) & (indiceDeAdelante no fue tachado || indiceDeAdelante ≤ indiceDeAtras)

Es decir que el número de iteraciones está en principio acotado por n. Por otro lado la variable *indiceDeAdelante*, que no se reinicia, es la que establece la condición de salida del ciclo principal. Si, análogamente a como hicimos antes, el ciclo principal hace k iteraciones y llamamos $iterAdel_t$ al número de iteraciones del ciclo de la línea 10, se puede ver que:

$$\sum_{t=1}^{k} iterAdel_t \leq n$$

porque en el momento en que el índice de adelante llega a n, ninguno de los ciclos itera mas.
Y en particular si definimos:

$beta(b)$ = if b then 1 else 0

Y definimos $indiceAtras_t$, $indiceAdelante_t$ a los valores de dichos índices en la iteración t, tenemos que:

$$n = \sum_{t=0}^{k} beta(a[indiceDeAtras_t] == a[indiceDeAdelante_t]) + iterAdel_t$$

ya que o indiceDeAdelante se incrementa en una unidad porque los elementos eran iguales, o itera el ciclo interno.

Viendo esto es de notar que tanto indiceDeAtras como indiceDeAdelante solo recorren el arreglo una vez.

Por lo tanto el algoritmo tiene un orden de complejidad lineal, es decir $O(n)$.

Observando el pseudocódigo, tambien notamos como en la medida que aumenta el número de apariciones de la moda (y por lo tanto la guarda del if de la línea 5 se haga falsa en más oportunidades) más rápido debería andar el algoritmo, ya que sólo se incrementa $indiceDeAdelante$. En vista de ésto, si los elementos del arreglo son todos iguales, el algoritmo hace solo una pasada, mientras que en el peor de los casos, hace dos pasadas completas.

5. Análisis Experimental

5.1. Experiencias realizadas

Nuevamente para este algoritmo se decidió medir tanto la cantidad de operaciones como el tiempo en función de la cantidad de elementos del arreglo, con la intención de confirmar nuestro analisis teórico. Para ello se generaron arreglos de largo creciente con elementos al azar (distribución uniforme) respetando que la frecuencia de la moda sea de por lo menos $n/2 + 1$.

Por otro lado para observar la influencia de la frecuencia de la moda en el comportamiento del algoritmo se hicieron corridas de prueba para un n fijo aumentando la frecuencia y midiendo la cantidad de operaciones y el tiempo insumido.

Además, en algunos casos fue posible utilizar la técnica de cuadrados mínimos para dar una función que aproxime el comportamiento observado experimentalmente. En dichos casos, la ecuación de $f(x)$ se encuentra en el gráfico.

5.2. Gráficos

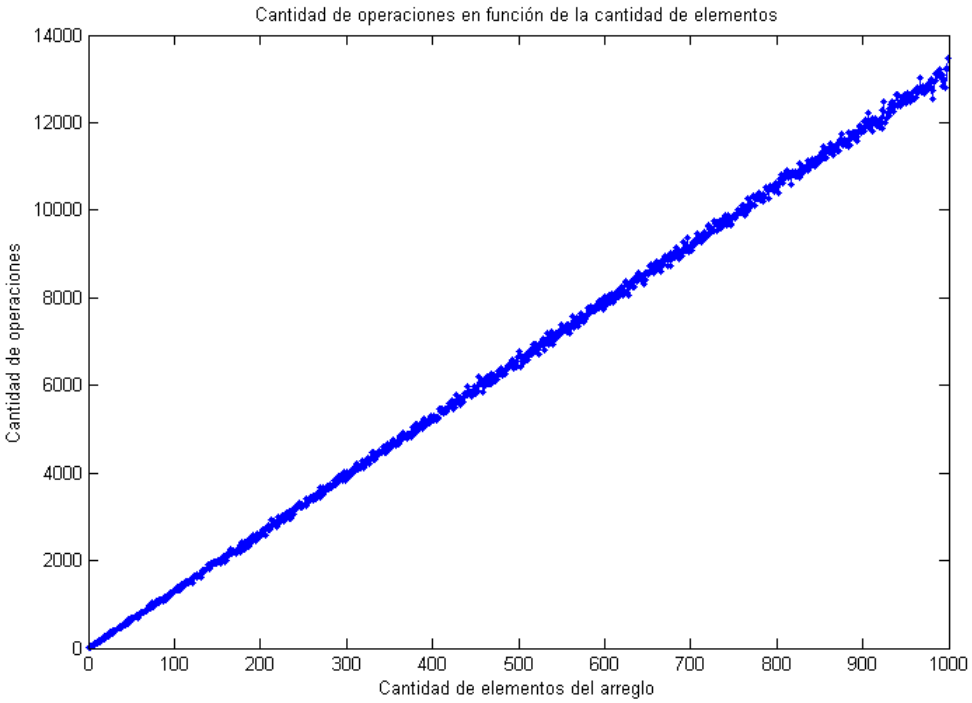


Figura 12: Cantidad de operaciones en función del tamaño del arreglo

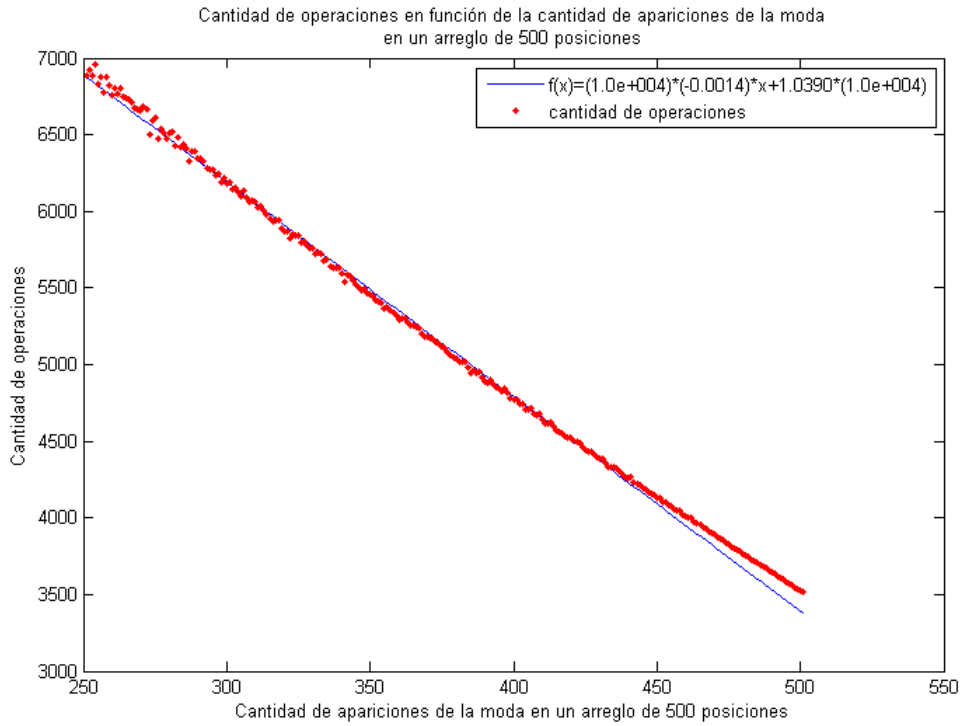


Figura 13: Cantidad de operaciones en función de la frecuencia ($n = 500$)

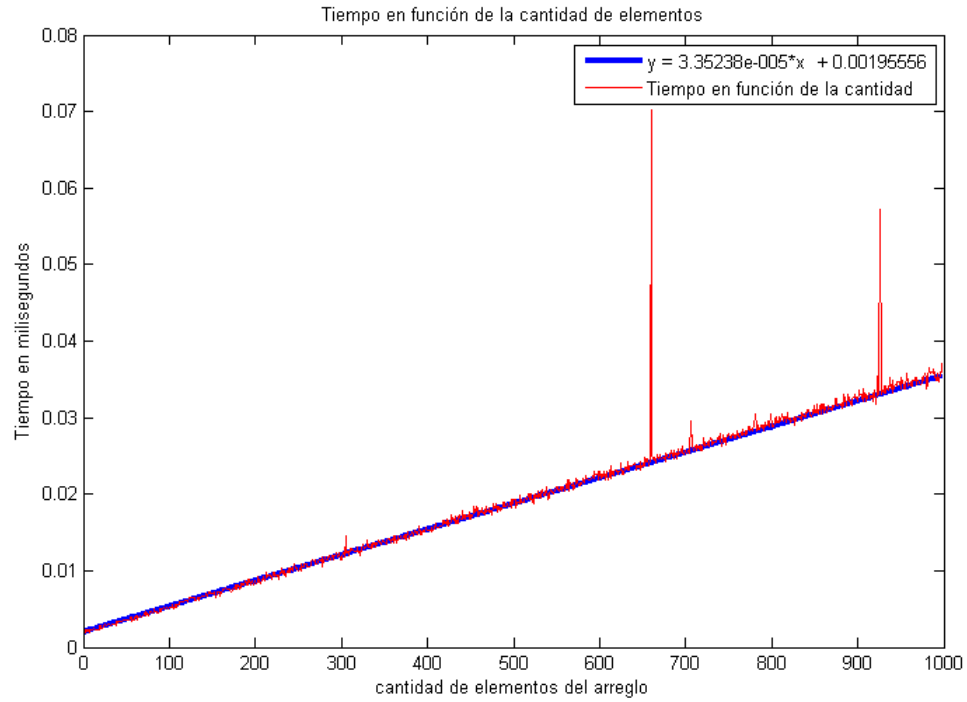


Figura 14: Tiempo (milisegundos) en función del tamaño del arreglo

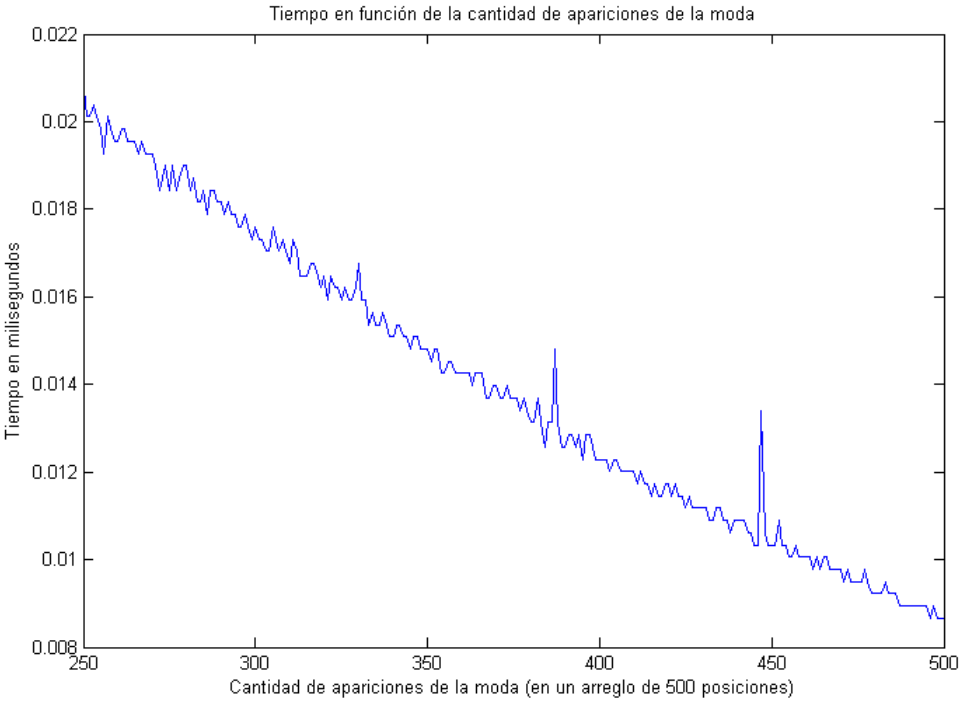


Figura 15: Tiempo (milisegundos) en función de la frecuencia ($n = 500$)

6. Discusión

En el gráfico de tiempo se observó que al aumentar el tamaño del arreglo, el tiempo aumenta de forma lineal. Lo mismo ocurre con la cantidad de operaciones . Esta situación se corresponde con el análisis teórico que se realizó.

Por otro lado en los gráficos en función de la frecuencia de la moda, el tiempo y las operaciones parecieran decrecer linealmente. Esto evidencia que el mejor caso se produce cuando todos los elementos del arreglo resultan ser la moda. Esto se explica porque si todos los elementos son iguales a la moda, el if de la línea 5 del pseudocódigo siempre tiene una condición de entrada falsa, por lo que el ciclo recorre el arreglo sin hacer otro tipo de operación.

Para finalizar, podemos decir que el comportamiento que obtuvimos coincidió con el esperado.

Parte IV
Conclusión

Consideramos que la realización del trabajo nos resultó provechosa en la medida que nos permitió contrastar el análisis teórico de la complejidad de los algoritmos con su comportamiento a nivel empírico.

Por otro lado, nos gustaría comentar posibles extensiones al trabajo, a saber, mejorar la selección de candidatos en el algoritmo de factorización, desarrollar nuevas podas para el ejercicio 2 y en el 3 lograr no utilizar un arreglo auxiliar.

Finalmente y a modo de conclusión, nos gustaría decir que pudimos observar como el comportamiento de los algoritmos se correspondió con lo que dedujimos durante el análisis.