

# Algoritmos y Estructura de Datos III

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

## Trabajo Práctico 1

### Grupo 1

Integrante	LU	Correo electrónico
Hernandez, Nicolas	122/13	nicoh22@hotmail.com
Kapobel, Rodrigo	695/12	rok_35@live.com.ar
Rey, Esteban	657/10	estebanlucianorey@gmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

### Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

# Contents

<b>1</b>	<b>Informe de correcciones</b>	<b>3</b>
1.1	Descripción de correcciones . . . . .	3
<b>2</b>	<b>Ejercicio 1</b>	<b>3</b>
2.1	Descripción de problema . . . . .	3
2.2	Explicación de resolución del problema . . . . .	4
2.3	Algoritmos . . . . .	5
2.3.1	Estructura interna para chequear elecciones y estados ocurridos . . . . .	7
2.4	Análisis de complejidades . . . . .	7
2.5	Demostración de correctitud . . . . .	8
2.6	Experimentos y conclusiones . . . . .	9
2.6.1	2.5 . . . . .	9
2.6.2	2.5 . . . . .	10
<b>3</b>	<b>Ejercicio 2</b>	<b>15</b>
3.1	Descripción de problema . . . . .	15
3.2	Explicación de resolución del problema . . . . .	15
3.3	Algoritmos . . . . .	17
3.4	Análisis de complejidades . . . . .	17
3.5	Demostración de correctitud . . . . .	18
3.6	Experimentos y conclusiones . . . . .	19
3.6.1	2.5 . . . . .	19
3.6.2	2.5 . . . . .	20
<b>4</b>	<b>Ejercicio 3</b>	<b>28</b>
4.1	Descripción de problema . . . . .	28
4.2	Explicación de resolución del problema . . . . .	28
4.3	Algoritmos . . . . .	31
4.4	Análisis de complejidades . . . . .	32
4.5	Demostración de correctitud . . . . .	33
4.6	Experimentos y conclusiones . . . . .	35
4.6.1	2.5 . . . . .	35
4.6.2	2.5 . . . . .	35
<b>5</b>	<b>Aclaraciones</b>	<b>42</b>
5.1	Aclaraciones para correr las implementaciones . . . . .	42

# 1 Informe de correcciones

## 1.1 Descripción de correcciones

### Ejercicio 1

- Se rehizo el algoritmo por realizar incorrectamente una poda la cual nos anulaba resultados óptimos.
- Se realizo un nuevo análisis de complejidad con la respectiva demostración de correctitud del algoritmo
- Se rehicieron las experimentaciones para este nuevo algoritmo

### Ejercicio 2

- Se modifico una sección del algoritmo para obtener una mejor performance.
- Se realizo un nuevo análisis de complejidad con la respectiva demostración de correctitud del algoritmo
- Se rehicieron las experimentaciones para el algoritmo modificado

### Ejercicio 3

- Se modifico una sección del algoritmo para obtener una mejor performance.
- Se realizo una nueva demostración de correctitud del algoritmo
- Se rehicieron las experimentaciones para el algoritmo modificado

# 2 Ejercicio 1

## 2.1 Descripción de problema

En este punto y en los restantes contaremos con un personaje llamado Indiana Jones, el cuál buscará resolver la pregunta más importante de la computación, es  $P=NP?$ .

Focalizandonos en este ejercicio, Indiana, irá en busca de una civilización antigua con su grupo de arqueologos, además, una tribu local, los ayudará a encontrar dicha civilización, donde se encontrará con una dificultad, la cual será cruzar un puente donde el mismo no se encuentra en las mejores condiciones.

Para cruzar dicho puente Indiana y el grupo cuenta con una única linterna y además, dicha tribu suele ser conocida por su canibalismo, por lo tanto al cruzar dicho puente no podrán quedar más canibales que arqueologos.

Nuestra intención será ayudarlo a cruzar de una forma eficiente donde cruce de la forma más rápido todo el grupo sin perder integrantes en el intento.

Por lo tanto, en este ejercicio nuestra entrada serán la cantidad de arqueologos y canibales y sus respectivas velocidades.

## 2.2 Explicación de resolución del problema

Para solucionar este problema, se deben ver la combinación de viajes entre lados del puente (lado A, origen y lado B, destino) que nos permiten pasar a todos los integrantes del grupo, del lado A al B en el menor tiempo posible. Siendo esta búsqueda una tarea exponencial, buscamos la forma de poder disminuir los casos evaluados, acotandonos solo a los relevantes para la consigna, aplicando de este modo la búsqueda del mínimo a travez de backtracking.

Dadas las restricciones del problema, se decidieron aplicar las siguientes podas sobre el árbol de desiciones:

- Las combinaciones evaluadas son, en el caso de enviar 2 personas, a **duplas sin repeticiones**.
- Los viajes, tanto de paso de A a B como de B a A que generan un 'desbalance', son obviados; es decir en el caso que en algún lado hay más canibales que arqueologos.
- Se toman las decisiones que no repitan estados ya alcanzados dentro de una misma rama para **no generar ciclos**
- Las secuencias de viajes que tomen más tiempo que una previamente calculada se descartan.

## 2.3 Algoritmos

**función** *EJ1()*

    crear cola desiciones para guardar elecciones

    eleccion es un par valido posible con información de las personas (canibal y/o arqueologo)

    se crean booleanos sePudoDeshacerEnvio y sePudoDeshaceRetorno con valor verdadero

$O(1)$

    se crea un entero minimo con valor -1

$O(1)$

**Mientras** *sePudoDeshacerEnvio*  $\wedge$  *sePudoDeshaceRetorno* **hacer**

**si** *pasaronTodos(escenario)*  $\wedge$  (*tiempo(escenario)* < *minimo*) **entonces**

            guarda:  $O(1)$

                para *minimo* asignar *tiempo(escenario)*

$O(1)$

**fin si**

**si** *tieneLampara(escenario)* **entonces**

            guarda:  $O(1)$

**si** (*tiempo(escenario)* < *minimo*) **entonces**

                    guarda:  $O(1)$

                        eleccion asignar *eleccionEnvioPosible(escenario)*

$O(\binom{n}{2} \times n)$

                        aplicarEleccionEnvio(*eleccion*, *escenario*)

$O(n)$

**de lo contrario**

*sePudoDeshacerEnvio* asignar *deshacerUltimaElecciónEnvio(escenario)*  $O(n)$

**fin si**

**fin si**

**de lo contrario**

            para *eleccion* asignar *eleccionRetornoPosible(escenario)*

$O(\binom{n}{2} \times n)$

**si** (*tiempo(escenario)* < *minimo*) **entonces**

                    guarda:  $O(1)$

                        aplicarRetornoEnvio(*eleccion*, *escenario*)

$O(n)$

**de lo contrario**

*sePudoRetornar* asignar *deshacerUltimoRetorno(escenario)*

$O(n)$

**fin si**

**fin si**

**fin si**

**fin ciclo**

**fin función**

**función** *aplicarElecciónEnvio(elección, escenario)*

    encolar en decisiones la elección tomada

$O(1)$

    actualizar estado del sistema en base a la decisión tomada

$O(1)$

    sumar a *escenario.tiempo* *tiempo(eleccion)*

$O(1)$

    guardar estado isla A y estado isla B como efectuado

$O(n)$

    marcar *escenario.lampara* falso

$O(1)$

**fin función**

**Complejidad total:**  $O(n)$

**función** *aplicarRetornoEnvio*(*eleccion*, *escenario*)

- encolar en decisiones la elección tomada O(1)
- actualizar estado del sistema en base a la decisión tomada O(1)
- sumar a escenario.tiempo tiempo(*eleccion*) O(1)
- guardar estado isla A y estado isla B como efectuado O(n)
- marcar escenario.lampara verdadero O(1)

**fin función**

**Complejidad total:**  $O(n)$

**función** *deshacerUltimaEleccionEnvio*(*escenario*)

- si** *hay decisiones para desencolar entonces*
  - desencolar de decisiones una elección O(1)
  - actualizar estado del sistema en base a la decisión desencolada O(1)
  - restar a escenario.tiempo tiempo(*eleccion*) O(1)
  - borrar estado isla A y estado isla B como efectuado O(n)
  - marcar escenario.lampara verdadero O(1)
  - devolver verdadero O(1)
- fin si**
- de lo contrario**
  - devolver falso O(1)
- fin si**

**fin función**

**Complejidad total:**  $O(n)$

**función** *deshacerUltimoRetorno*(*escenario*)

- si** *hay decisiones para desencolar entonces*
  - desencolar de decisiones una elección O(1)
  - actualizar estado del sistema en base a la decisión desencolada O(1)
  - restar a escenario.tiempo tiempo(*eleccion*) O(1)
  - borrar estado isla A y estado isla B como efectuado O(n)
  - marcar escenario.lampara falso O(1)
  - devolver verdadero O(1)
- fin si**
- de lo contrario**
  - devolver falso O(1)
- fin si**

**fin función**

**Complejidad total:**  $O(n)$

**función** *eleccionEnvioPosible*(*escenario*)

- Mientras** *Hay elecciones disponibles hacer*
  - si** *ambas personas estan en isla A  $\wedge$  no ocurrió elección  $\wedge$  cantidades balanceadas entonces*
    - Ciclo:**  $O(\binom{n}{2})$
    - Guarda:**  $O(n)$
    - devolver elección O(1)
  - fin si**
- fin ciclo**

**fin función**

**Complejidad total:**  $O(\binom{n}{2}) \times n$

```

función eleccionRetornoPosible(escenario)
  Mientras Hay elecciones disponibles hacer
    Ciclo:  $O(\binom{n}{2})$ 
    si ambas personas estan en isla B  $\wedge$  no ocurrió elección  $\wedge$  cantidades balanceadas
    entonces
      Guarda: O(n)
      devolver elección
      O(1)
    fin si
  fin ciclo
fin función
  Guarda: O(n)
  O(1)
Complejidad total:  $O(\binom{n}{2}) \times n$ 

```

### 2.3.1 Estructura interna para chequear elecciones y estados ocurridos

Siendo que la disposición de las personas en las dos islas, pueden representarse por un arreglo en donde cada elemento representa la presencia de una persona en la isla A con un booleano (Esta o no esta) entonces, si guardamos dichas secuencias en una estructura de trie, podemos chequear la ocurrencia de un estado en  $O(n)$ , al igual que el insertado de un estado nuevo.

## 2.4 Análisis de complejidades

Para analizar la complejidad temporal consideraremos un árbol de opciones que el algoritmo recorre. Cada caso completo evaluado sea solución o no, está caracterizado por una hoja, y la rama que va desde la raíz a cada una de ellas es la secuencia de pasos necesarios para alcanzar cada caso.

Al ver las distintas podas efectuadas en el algoritmo podemos ver que, la referida al tiempo no caracteriza la complejidad del mismo: puede darse el caso en que la mejor solución se encuentre en la primer rama evaluada, con lo cual todas las demás serán analizadas de forma acotada gracias a la poda; o bien puede darse el caso en que el orden de las ramas evaluadas sea de mayor a menor en tiempo de solución encontrada, con lo que no se aplica la poda en ningún momento.

Siendo  $n$  la cantidad total de arqueólogos y canibales, la poda referida a la cantidad de opciones a tener en cuenta por viaje a través del puente, de las  $n \times n$  combinaciones existentes, solo toma  $\binom{n}{2}$  tuplas de personas si enviamos dos personas o  $n$  posibilidades si enviamos una; es decir  $O(\binom{n}{2} + n) \subseteq O(\binom{n}{2})$  posibilidades para cada cruce.

Un estado en cada isla representa la combinación de canibales y arqueólogos presente.

Si caracterizamos a cada nodo del árbol como un estado de cada isla, entonces para hacer cada transición debemos realizar  $O(\binom{n}{2} * n)$  evaluaciones de posibles caminos. La multiplicación por  $n$  se debe a que en cada posible combinación debemos chequear si se producen ciclos, lo que se resuelve en  $O(n)$  consultando un trie.

Bajo este conteo de casos tendremos que en el primer nivel del árbol se tienen  $O(\binom{n}{2} * n)$  estados, en el segundo  $O((\binom{n}{2} * n)^2)$ , en el tercero  $O(((\binom{n}{2} * n)^3))$  y así por cada nivel que haya en él.

La cantidad de niveles presentes en el árbol está dada por la cantidad de estados que hay en cada rama, o sea en cada solución. Como en cada solución no se repiten los estados por la poda efectuada, entonces se pueden contabilizar los casos de la siguiente forma:

Dado un estado, la cantidad de arqueólogos de cada lado debe ser mayor o igual a la cantidad total de canibales, y que las cantidades de ambos en un lado están en función a las del otro: llamamos  $a$  a la cantidad de arqueólogos y  $c$  a la de canibales del lado A. Siendo  $\sum_{i=1}^a \binom{a}{i}$  la cantidad de combinaciones

posibles de arqueologos que se pueden tener, y dado que siempre hay menor o igual cantidad de canibales, entonces para la cantidad  $i - esima$  de arqueologos se tienen  $\sum_{j=1}^i \binom{c}{j}$  combinaciones de canibales. Resumiendo se tiene la siguiente cantidad de estados válidos:

$$\sum_{i=1}^a \binom{a}{i} \left[ \sum_{j=1}^{\min(i,c)} \binom{c}{j} \right] = k$$

Con lo cual, en adición a lo anteriormente calculado, la cantidad de operaciones totales será:

$$\sum_{i=1}^k \left( \binom{n}{2} * n \right)^i$$

Para acotar este valor utilizaremos que el número total de subconjuntos combinatorios  $\sum_{i=1}^m \binom{m}{i}$  es  $2^m$ :

$$\sum_{i=1}^a \binom{a}{i} \left[ \sum_{j=1}^{\min(i,c)} \binom{c}{j} \right] \leq \sum_{i=1}^a \binom{a}{i} \left[ \sum_{j=1}^c \binom{c}{j} \right] \leq 2^a * 2^c = 2^{a+c} = 2^n$$

Considerando que  $\binom{n}{2} \leq n^2$ , la complejidad queda acotada de la siguiente forma:

$$\sum_{i=1}^k \left( \binom{n}{2} * n \right)^i \leq \sum_{i=1}^{2^n} n^{3i} = n^3 + n^6 + n^{12} + \dots + n^{3*2^n} \in O(n^{2^n})$$

## 2.5 Demostración de correctitud

Como dentro de todas las posibles soluciones que podría encontrar un algoritmo de fuerza bruta, existen varias que no son válidas, al aplicar las podas descriptas anteriormente nos quedaremos con todas aquellas que tienen sentido dentro de nuestro problema:

### Combinaciones posibles

En cada viaje a travez del puente, a priori se puede mandar cualquier persona, o combinación de 2 personas. Para enumerar estos casos u opciones, podemos plantear una matriz de  $N \times N$  para simbolizar en cada casilla cada combinación posible. Es inmediato ver que las casillas de un lado y otro de la diagonal de la matriz repiten casos y otras en la diagonal de la misma, que repite la persona. Como también existe la posibilidad de enviar de a 1 persona por vez, entonces la diagonal de la matriz representará estas elecciones, por otro lado, nos quedaremos con solo 1 mitad de la matriz para no tener en cuenta los casos repetidos. Bajo esta representación obtenemos todas las combinaciones validas de personas que pueden llegar a viajar en cada cruce.

Como las personas solo pueden encontrarse en 1 isla en todo momento, entonces de todas las opciones se consideran aquellas que posean a todos los integrantes en el lado de partida.

### Desbalance

No puede haber más caníbales que arqueologos en ningúun lado en ningúun momento, la Elección tomada deberá mantener este invariante al enviar para el otro lado las personas elegidas.

## Ciclos

En el caso en que se envíe la opción  $(i; j)$  al lado A, bajo las restricciones anteriores nada impide que en el paso siguiente se decida mandar de regreso al mismo par, generandonos un ciclo. Esta elección claramente carece de sentido ya que nos regresa a un estado previo y con más tiempo acumulado. Podemos generalizar el ejemplo anterior a decir que no tiene sentido, dentro de una rama de solución, repetir un estado previo. Para evitarlo se debe guardar los estados previos por los cuales se pasó y desestimar cualquier opción que nos retorne a ellos.

## Tiempo

De haber encontrado una solución, no tendrá sentido, al estar buscando una nueva, seguir por una rama si se supera el tiempo de la misma, ya que de alcanzar una solución, la misma no será óptima. Con lo cual, se desestima la búsqueda por una rama si la misma supera en tiempo alguna solución ya encontrada. De esta forma, al finalizar el algoritmo y haber encontrado solución, podemos recuperar el tiempo logrado por la solución más rápida. Esta poda ahorra la necesidad de una vez finalizado el algoritmo, tener que hallar la solución de menor tiempo, dentro de todas las soluciones válidas encontradas.

## 2.6 Experimentos y conclusiones

### 2.6.1 Test

Para verificar el correcto funcionamiento de nuestro algoritmo , elaboramos diversos tests, los cuales serán enunciados a continuación.

#### **Caso 1: Todos los arqueologos y canibales presentan la misma velocidad**

Este caso se da cuando  $V_i = W_j \forall (i,j) \leftarrow [0..6]$

#### **Caso 2: No hay canibales**

Esta versión se da cuando  $M = 0$ .

#### **Caso 3: Todos los arqueologos y canibales presentan velocidades distintas**

Este caso se da cuando  $V_i \neq W_j \forall (i,j) \leftarrow [0..6]$

#### **Caso 4: Hay un canibal cada dos arqueologos y viceversa**

Este tipo de caso se cumple cuando  $N = 2 * M$  o  $M = 2 * N$

#### **Caso 5: Hay más canibales que arqueologos**

Este tipo de caso se cumple cuando  $N < M$ , también denominado sin solución.

## 2.6.2 Performance De Algoritmo y Gráfico

Acorde a lo solicitado, mostraremos distintos tipos de familias de casos para nuestro algoritmo, y además, daremos el tiempo estimado según la complejidad del algoritmo calculada anteriormente.

Luego de realizar varios chequeos de nuestro algoritmo, se pudieron elaborar una serie de casos puntuales:

- Hay más canibales que arqueologos, es decir, no hay solución.
- No hay canibales
- Todos los canibales y arqueologos presentan velocidades iguales
- Todos los canibales y arqueologos presentan velocidades distintas
- Hay más arqueologos que canibales con velocidades dispares

Dado estos estilos de familias y, como las combinaciones de arqueologos y canibales no se pueden respetar en todos los casos, desarrollamos un gráfico para cada familia con la función resultante de lo que demora nuestro algoritmo en obtener una solución posible para cada tipo de entrada.

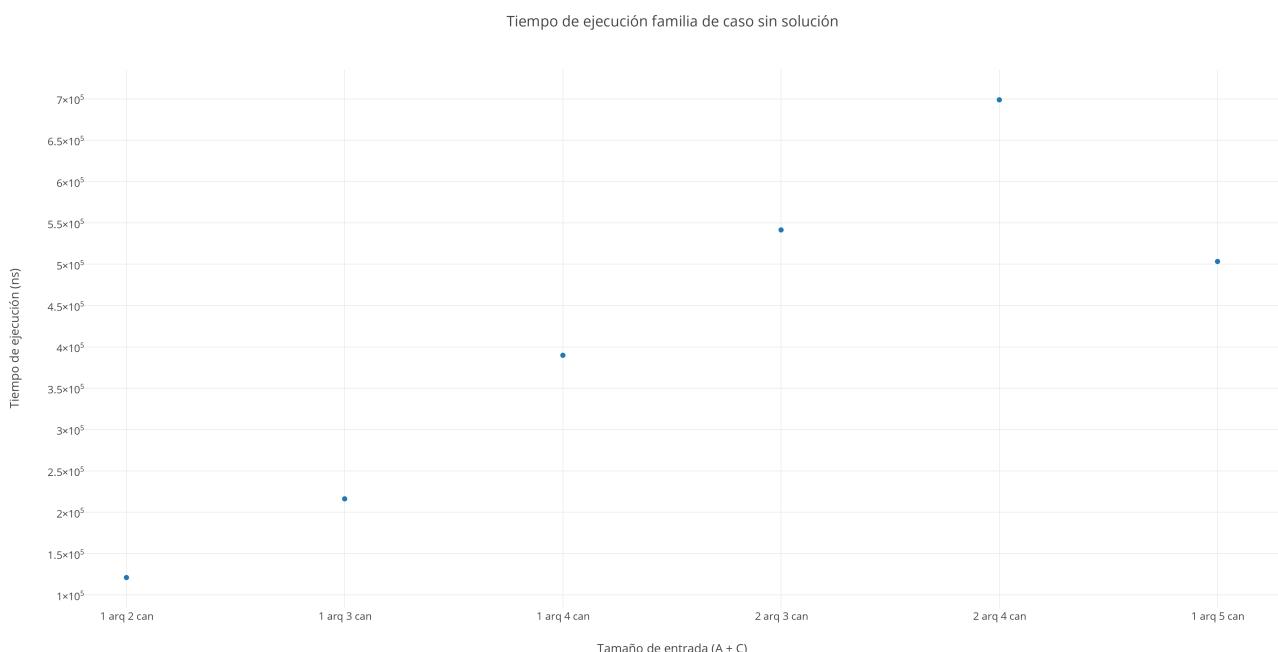


Gráfico 1.1 - Sin Solucion

Tiempo de ejecucion de familia de caso sin canibales

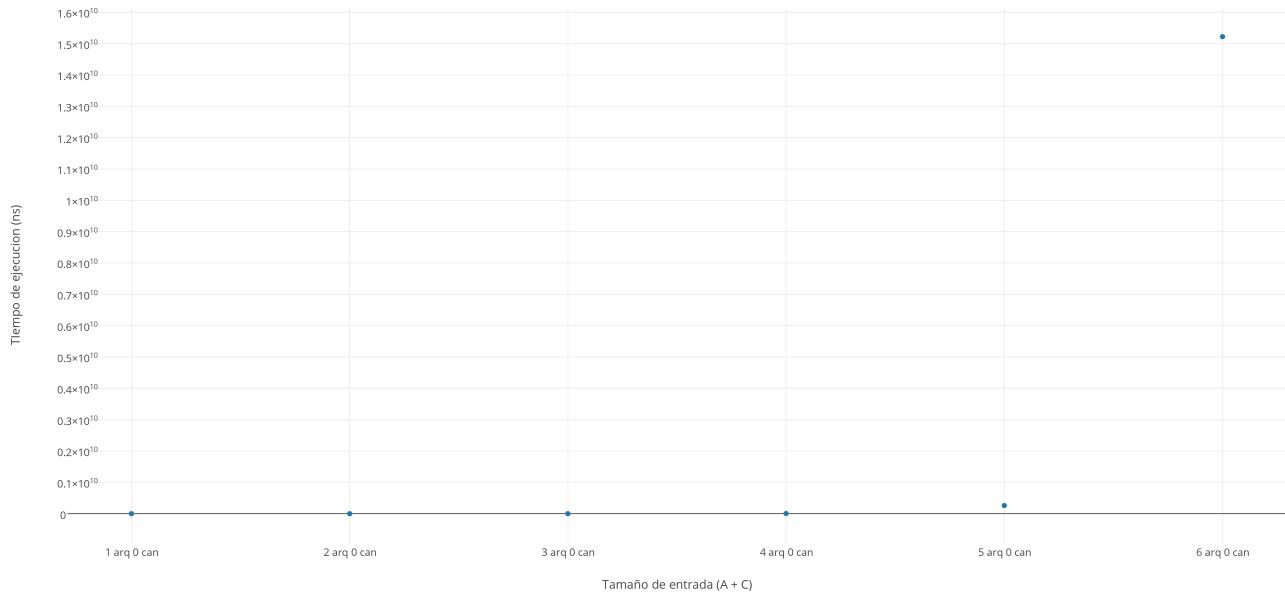


Gráfico 1.2 - Sin Canibales

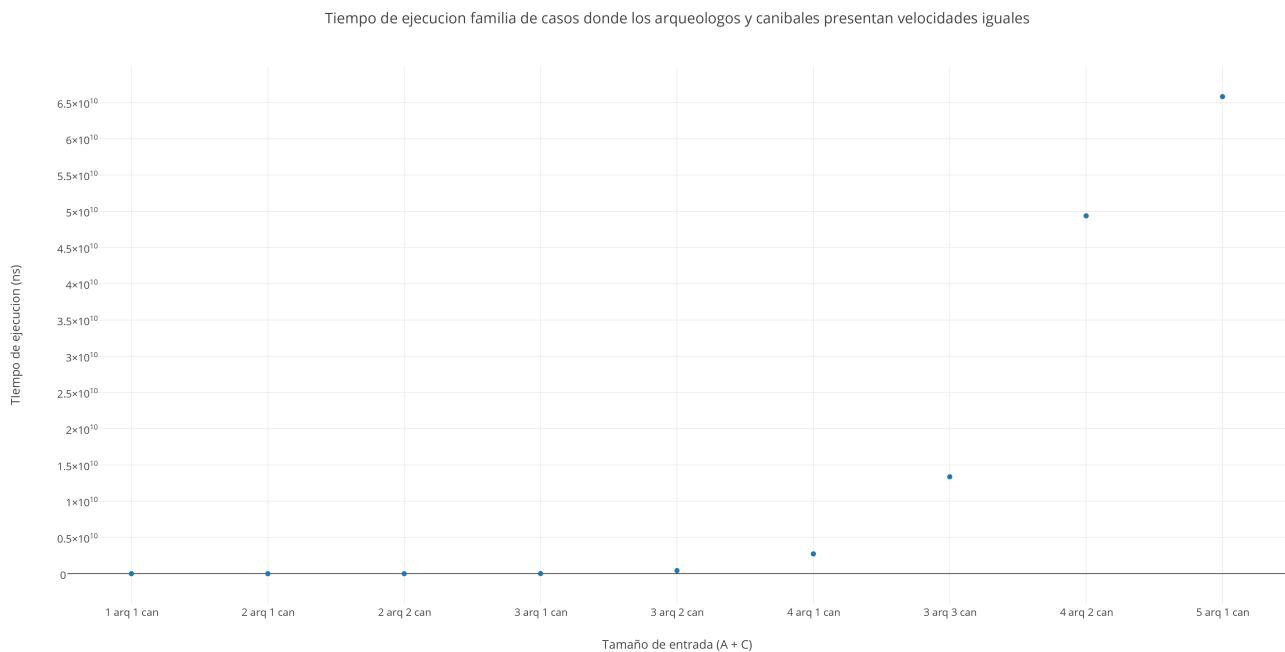


Gráfico 1.3 - Velocidades Iguales Para Todos

Tiempo de ejecucion familia de casos donde los arqueologos y canibales presentan velocidades distintas

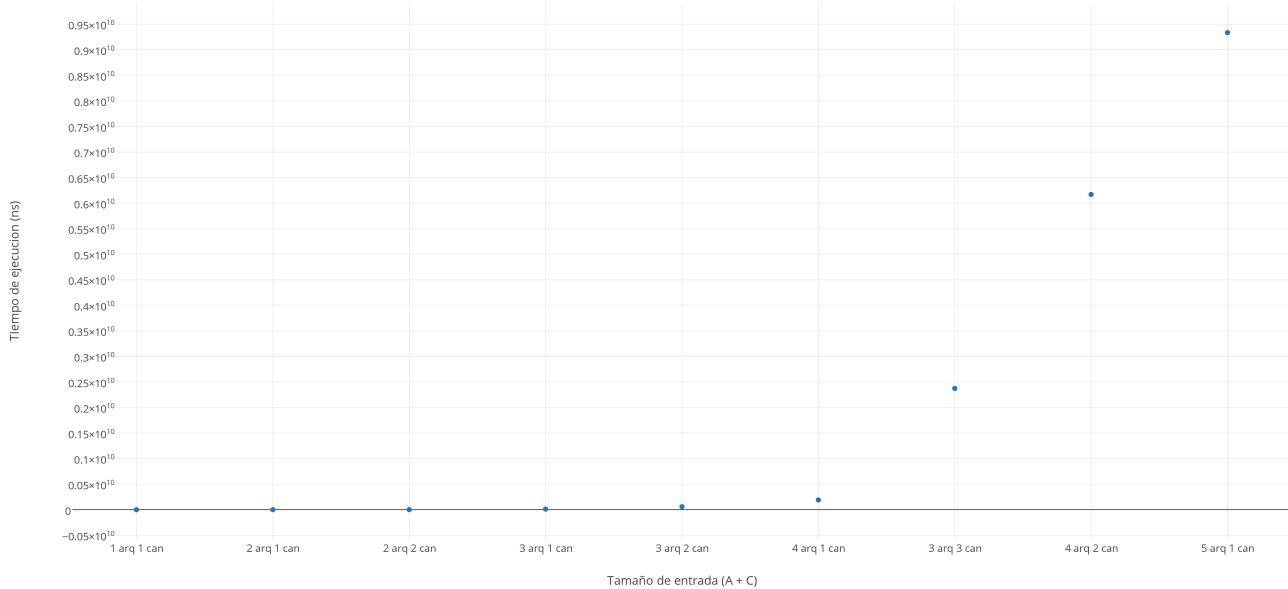


Gráfico 1.4 - Velocidades Distintas Para Todos

Tiempo de ejecucion familia de caso donde hay mas arqueologos que canibales con velocidades random

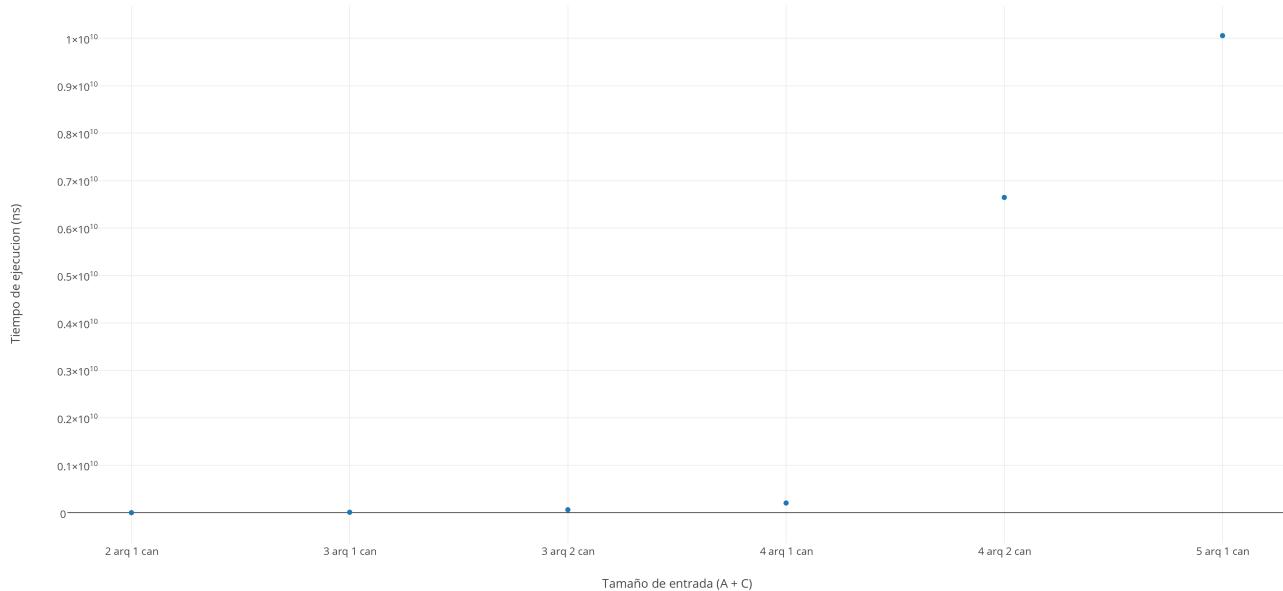


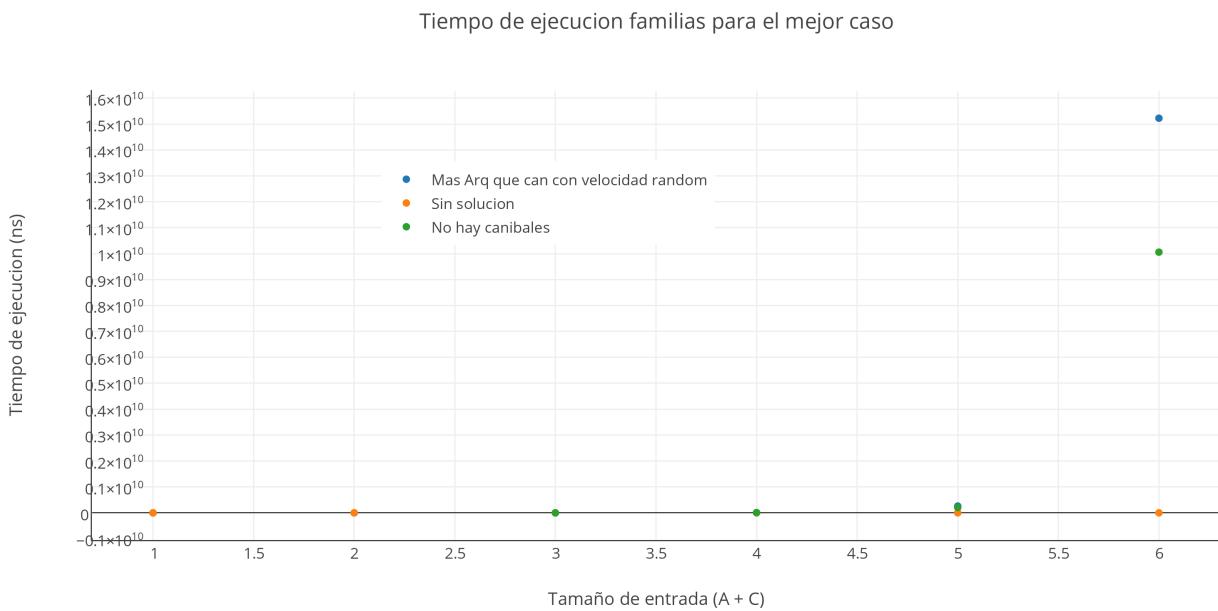
Gráfico 1.5 - Velocidades Random Para Todos

Luego de realizar dichas mediciones pudimos observar que todas las funciones representativas a cada test son crecientes cuando el  $N = A+C$  (con A la cantidad de arqueologos y C la cantidad de canibales) aumenta, exceptuando el caso sin solución (Ver gráfico 1.1) el cual tiene un máximo con 2 arqueologos y 4 canibales y luego decrece. Esto se da ya que cuando nuestro algoritmo va chequeando las ramas y cantidades de elementos, verifica con mayor rapidez que hay más cantidad de canibales que arqueologos ya que la diferencia entre canibal - arqueólogo es mayor. Además, corroboramos que para los casos en los cuales no hay solución o no hay canibales se obtiene una mejor performance,

que para aquellos en los cuales hay solución siempre y/o hay canibales siempre, como son los casos de velocidades distintas o iguales.

Es por esto que desarrollamos dos gráficos comparativos para visualizar cual será nuestro mejor y peor caso.

En el primero trabajaremos con los que se obtiene una mejor performance. Cabe aclarar que debido a que las entradas para un caso no son compatibles en otros (por ejemplo, en el caso sin solución se utiliza la entrada 1 arqueólogo y 3 canibales y en el sin canibales dicho caso no es permitido), trabajamos con el valor de N final, el cual se obtiene de sumar la cantidad de arqueólogos con la cantidad de canibales.



*Gráfico 1.6 - Comparativo Mejor Caso*

Se puede observar en el gráfico, tres funciones las cuales representan el tiempo de ejecución de las familias que resultan mas beneficiadas al trabajar con nuestro algoritmo:

- Hay más canibales que arqueólogos, es decir, no hay solución.
- No hay canibales
- Hay más arqueólogos que canibales con velocidades dispares

Como se observa en el gráfico, la función representativa de la familia número 1, presenta una mejor performance en relación a las otras. Esto se debe a que nuestro algoritmo va chequeando y probando todos los posibles pares que viajen de un lado al otro, y, como inicialmente chequea que hay más canibales que arqueólogos imposibilitando un par posible, el algoritmo corta sin probar los posibles pares, finalizando su ejecución.

Luego de haber chequeado dichas instancias, pudimos llegar a la conclusión que la familia de casos que presenta una mejor performance es en el cual **Hay más canibales que arqueólogos, es decir, no hay solución**

Luego, para verificar el peor caso, desarrollamos otro gráfico de mediciones que contendrá los peores casos posibles. Como dichos casos presentan las mismas entradas válidas tanto en uno como

en otro, fue posible graficarlos juntos mostrando el tiempo medido para cada entrada.

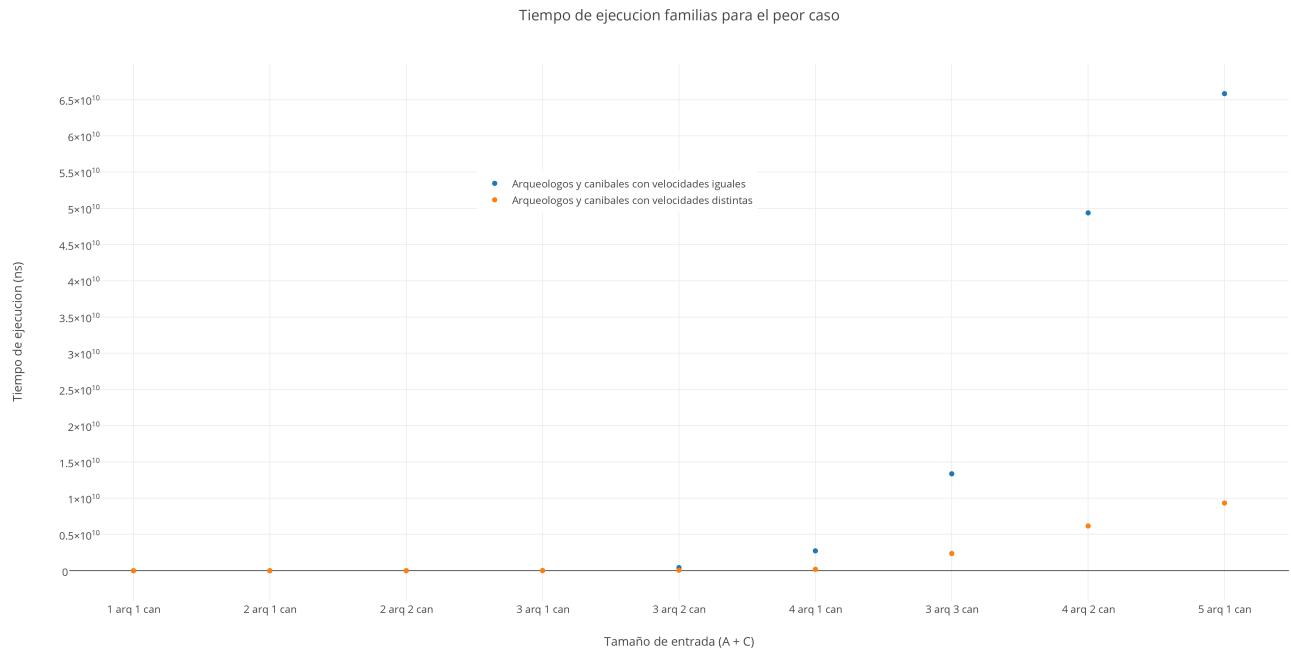


Gráfico 1.6 - Comparativo Peor Caso

Se puede observar en el gráfico, dos funciones las cuales representan el tiempo de ejecución de las familias que resultan menos beneficiadas:

- Todos los canibales y arqueologos presentan velocidades iguales
- Todos los canibales y arqueologos presentan velocidades distintas

Es posible observar en el gráfico, que la función representativa de la familia número 1, presenta una peor performance en relación a la otra. Esto se debe a que nuestro algoritmo va chequeando y probando todos las permutaciones posibles de pares que viajen de un lado al otro, y, además como el mismo tiene realizado una poda la cual consiste en anular las ramas que tarden más tiempo, dicha familia finalizará más tarde su ejecución en relación a la otra, ya que al tener a todas las personas con las mismas velocidades dicha poda no podrá ser utilizada, generando el peor caso posible.

Por lo tanto, el peor caso será cuando **Todos los canibales y arqueologos presentan velocidades iguales, y dicha combinación de arqueólogo - canibal tenga una solución posible**

Como se trabaja con pocos casos y la complejidad teórica es muy grande, no será posible graficarla ya que el tiempo insumido es muy elevado, y dado que  $N$  es pequeño (1 a 6), la muestra no es significativa. De querer trabajar con  $N > 6$ , la capacidad de cómputo de los dispositivos físicos disponibles es inferior a la necesaria, haciendo impracticable su análisis.

### 3 Ejercicio 2

#### 3.1 Descripción de problema

Como habíamos enunciado en el punto anterior, Indiana Jones buscaba encontrar la respuesta a la pregunta es  $P = NP?$ .

Luego de cruzar el puente de estructura dudosa, Indiana y el equipo llegan a una fortaleza antigua, pero, se encuentran con un nuevo inconveniente, una puerta por la cual deben pasar para seguir el camino se encuentra cerrada con llave.

Dicha llave se encuentra en una balanza de dos platillos, donde la llave se encuentra en el platillo de la izquierda mientras que el otro está vacío.

Para poder quitar la llave y nivelar dicha balanza, tendremos unas pesas con valores en potencias de 3.

Nuestra objetivo en este punto consistirá en ayudarlos, mediante dichas pesas, a reestablecer la balanza al equilibrio anterior a haber sacado la llave.

#### 3.2 Explicación de resolución del problema

Una aclaración previa que será de utilidad: como se dio como precondición que la llave puede llegar a tomar un valor límite de  $10^{15}$ , trabajaremos con variables del tipo long long las cuales nos permitirán alcanzarlo.

Para solucionar el problema de poder quitar la llave y dejar equilibrada la balanza, realizamos un algoritmo que explicaremos a continuación:

1) Como las pesas, presentan un peso potencia de 3, creamos un arreglo de sumas parciales *sumasParciales* donde iremos guardando las sumas parciales de las potencias desde  $3^0$  hasta un  $3^i$  en el paso  $i - \text{esimo}$ . La última suma parcial (que llamaremos paso  $n - \text{esima}$ ) será  $\geq P$ .

Cada valor en el arreglo *sumasParciales* representa un intervalo:

$$[0, \sum_{j=0}^i (3^j)] \quad (1)$$

Que es el número más grande representable con las potencias de 3 que realizan la sumatoria indicada en el límite superior del intervalo. Veremos luego que un número  $x \in \mathbb{Z}$  que esté contenido en un intervalo como el anterior puede formarse con una combinación de sumas y restas de las potencia de 3 que conforman el límite del intervalo (no necesariamente todas ellas).

2) Para que nuestro algoritmo pueda encontrar que pesas con valor potencia de 3 utilizar para equilibrar la balanza, empezaremos con  $P$  (el valor de nuestra llave), buscando el intervalo que lo contiene.

Este intervalo, será por defecto el más grande, dado que así armamos nuestro arreglo *sumasParciales*. Al valor  $P$  le restaremos la pesa más grande posible que será en este caso  $3^n$ . Para obtener este valor simplemente debemos hacer una resta entre la posición  $n - \text{esima}$  y la anterior del arreglo

*sumasParciales.*

Luego sabemos que el resto  $x$  es un valor entero. Es decir, puede ser mayor, menor o igual a cero.

Si  $x$  es cero, quiere decir que  $P$  era un valor potencia de 3. Por lo cual ya hemos finalizado y colocamos la pesa encontrada ( $3^n$ ) en el balanzin donde se encontraba la llave  $P$ .

Si el valor  $x$  es menor que cero, sabemos que en los siguientes pasos, tendremos que cambiar de plato hasta que el valor vuelva a ser mayor que cero o cero (en cuyo caso habremos finalizado).

Si el valor  $x$  es mayor que cero, no cambiamos de plato y continuamos agregando pesas hasta llegar a cero.

Los siguientes pasos serán iguales al primero, siempre buscando el intervalo donde se encuentra cada resto sucesivo en el arreglo *sumasParciales* recorriendo cada una de las posiciones del arreglo.

Cuando  $X$  sea cero, tendremos nuestras pesas en orden decreciente, con lo cual solo es necesario invertir este orden para cumplir con el formato de salida deseado.

### 3.3 Algoritmos

```

función Balancear()
    entero sumaParcial ← 0                               O(1)
    arreglo sumasParciales                                O(1)
    Mientras sumaParcial ≤ P hacer                      ciclo: O(log(P))
        sumasParciales ∪ sumaParcial                      O(1)
        para sumaParcial asignar sumaParcial + 3i      O(1)
        incrementar i                                     O(1)
    fin ciclo
    entero indice ← size(sumasParciales)-1             O(1)
    entero equilibrioActual ← P                         O(1)
    entero potenciaActual ← 0                           O(1)
    Mientras /equilibrioActual/ > 0 hacer           ciclo: O(log(P))
        si sumasParciales[indice-1] < /equilibrioActual/ ≤ sumasParciales[indice] entonces
            guarda: O(1)
            para potenciaActual asignar sumasParciales[indice] - sumasParciales[indice-1] O(1)
            si equilibrioActual < 0 entonces
                para equilibrioActual asignar potenciaActual + equilibrioActual
                platoDerecho ∪ potenciaActual
            de lo contrario
                para equilibrioActual asignar equilibrioActual - potenciaActual
                platoIzquierdo ∪ potenciaActual
            fin si
        fin si
        indice←
    fin ciclo
    devolver size(platoDerecho)                         O(1)
    devolver size(platoIzquierdo)                        O(1)
    devolver invertir(platoDerecho)                      O(1)
    devolver invertir(platoIzquierdo)                    O(1)
fin función                                         total: O(log(P))

```

### 3.4 Análisis de complejidades

Nuestro algoritmo como mencionamos anteriormente presenta dos ciclos predominantes de los cuales el segundo ciclo es el que realiza la búsqueda de las pesas.

El primero ciclo consta en armar un arreglo de sumas de potencias de tres *sumasParciales*, donde la  $i - \text{esima}$  posición es la suma desde  $3^0$  hasta  $3^i$  y la  $n - \text{esima}$  será la suma desde  $3^0$  hasta  $3^n$  tal que es igual a  $P$  (el valor de nuestra llave) o en su defecto el inmediato mayor.

Podemos notar que para representar un número en base tres necesitamos  $\lceil \log_3(P) \rceil$  divisiones con lo que obtendremos un total de  $\lceil \log_3(P) \rceil$  potencias de tres. En nuestro algoritmo, puede ser necesario tomar la potencia inmediatamente mayor a  $P$  por lo cual tendríamos  $\lceil \log_3(P) \rceil + 1$  potencias de tres en el peor caso, aunque en términos de complejidad sigue siendo orden logarítmico.

Queremos ver entonces que  $\lceil \log_3(P) \rceil$  es una buena cota para la cantidad de iteraciones que realizamos para obtener el arreglo *sumasParciales* y en consecuencia, para realizar el ciclo que encuentra las pesas adecuadas. Es decir, que seguro será suficiente sumar a lo sumo  $\lceil \log_3(P) \rceil$

potencias de tres para obtener una suma que sea mayor que  $P$ . Por lo tanto:

$$P \leq \sum_{i=0}^{\lceil \log_3(P) \rceil} (3^i) \quad (2)$$

Tomando el último elemento de la suma tenemos que:

$$3^{\lceil \log_3(P) \rceil} \geq 3^{\log_3(P)} = P \quad (3)$$

Por lo tanto se puede ver que el último término de la sumatoria ya es más grande que  $P$ . Por lo tanto la suma total lo será. En consecuencia, a lo sumo será necesario sumar  $\lceil \log_3(P) \rceil$  potencias de tres.

Además como se sabe, en términos de complejidad, los logaritmos en cualquier base son iguales al logaritmo en base diez, dado que la única diferencia es una constante multiplicativa. Por lo tanto

$$O(\lceil \log_3(P) \rceil) \subseteq O(\lceil \log(P) \rceil) \quad (4)$$

Dado que el ciclo donde se obtienen las pesas se basa en recorrer todas las posiciones del arreglo *sumasParciales* y además las operaciones realizadas son en orden constante, el orden de complejidad total será el mencionado. Además, esta clase de complejidad está incluida en la clase de complejidad  $\sqrt{P}$ . Por lo tanto se cumple con la complejidad pedida en el enunciado del problema.

### 3.5 Demostración de correctitud

En nuestro algoritmo como hemos mencionado anteriormente en la explicación del mismo, la etapa más importante es, a la hora de obtener las pesas que equilibran la balanza, donde se realiza un ciclo que va disminuyendo el valor *equilibrioActual* (inicialmente  $P$ ) hasta llegar a 0 recorriendo el arreglo de *sumasParciales* de sumas de potencias de tres.

Una vez alcanzado ese valor se puede dar por finalizado el algoritmo, dado que hemos encontrado una manera de sumar y restar potencias de tres tales que como resultado se obtiene  $P$ .

Por lo tanto para probar que el algoritmo es correcto deberemos probar primero que el algoritmo termina, es decir, que *equilibrioActual* alcanza el valor cero luego de iterar el arreglo *sumasParciales* una sola vez y luego que utilizamos siempre potencias de tres diferentes.

Queremos probar entonces que nuestro algoritmo encuentra las pesas necesarias para todo  $P \in \mathbb{Z}$  utilizando a lo sumo  $k$  potencias de tres diferentes, con  $k$  el primer valor tal que la suma de las primeras  $k$  potencias es mayor o igual a  $P$ . Es decir:

$$(\forall P \in \mathbb{Z}), |P| \leq \sum_{i=0}^k (3^i) \text{ con } k \geq 0 \quad (5)$$

Para  $k = 0$ ,  $P$  tiene que ser igual a uno en modulo. Por lo cual, luego de realizar la resta el algoritmo ha finalizado.

Para el paso inductivo  $k = n > 0$  sabemos que nuestro algoritmo encuentra la respuesta a cualquier  $P$  con a lo sumo  $n$  potencias de tres diferentes tal que  $n$  es el primer valor que cumple que la suma de las primeras  $n$  potencias es mayor o igual a  $P$ .

Sea  $P'$  tal que  $n + 1$  es el primer valor que cumple que la suma de las primeras  $n + 1$  potencias es mayor o igual a  $P'$ , entonces:

$$\sum_{i=0}^n (3^i) < |P'| \leq \sum_{i=0}^{n+1} (3^i) \quad (6)$$

El algoritmo realiza la siguiente operación:

$$P'' = \begin{cases} 3^{n+1} + P' & \text{con } P' < 0 \\ P' - 3^{n+1} & \text{con } P' > 0 \end{cases} \quad (7)$$

Veamos que en cualquier caso, el nuevo valor  $P''$  es menor o igual a  $\sum_{i=0}^n (3^i)$ . En el caso  $P' < 0$ :

$$3^{n+1} - |P'| \leq \sum_{i=0}^n (3^i) \iff |P'| \geq -\sum_{i=0}^n (3^i) + 3^{n+1} \quad (8)$$

Lo cual es cierto ya que:

$$-\sum_{i=0}^n (3^i) + 3^{n+1} > \sum_{i=0}^n (3^i) \iff 3^{n+1} > 2 * \sum_{i=0}^n (3^i) \iff \quad (9)$$

$$3^{n+1} > 2 * \frac{1 - 3^{n+1}}{1 - 3} \iff 3^{n+1} > -1 + 3^{n+1} \iff 0 > -1 \quad (10)$$

Además veamos que  $-\sum_{i=0}^n (3^i) + 3^{n+1} > |P'|$  no puede suceder dado que el módulo del intervalo es:

$$-\sum_{i=0}^n (3^i) + 3^{n+1} - \sum_{i=0}^n (3^i) = 3^{n+1} - 2 * \sum_{i=0}^n (3^i) = 3^{n+1} - (-1 + 3^{n+1}) = 1 \quad (11)$$

Y  $P$  es entero, por lo cual nunca tomará valores decimales.

El caso con  $P' > 0$  se puede ver directamente pasando  $3^{n+1}$  sumando.

$$P' - 3^{n+1} \leq \sum_{i=0}^n (3^i) \iff P' \leq \sum_{i=0}^n (3^i) + 3^{n+1} \iff P' \leq \sum_{i=0}^{n+1} (3^i) \quad (12)$$

Por lo cual, el valor de  $P''$  se encuentra en un intervalo más pequeño que  $P'$ . Por hipótesis inductiva, podemos generar a  $P'$  con a lo sumo  $n$  potencias de tres diferentes, y como la pesa más grande será  $3^n$ , se garantiza que no se repita  $3^{n+1}$ . Por lo tanto, hemos probado inductivamente que vale (5). Es decir, podemos generar cualquier numero entero, con a lo sumo  $n$  potencias de tres diferentes, con  $n$  el primero que cumple la propiedad enunciada en (5). Como en a lo sumo  $k$  iteraciones se llega al valor cero, el algoritmo finaliza y se obtiene la solución al problema.

## 3.6 Experimentos y conclusiones

### 3.6.1 Test

Luego de realizar la implementación de nuestro algoritmo, desarrollamos tests, para corroborar que nuestro algoritmo es el indicado.

A continuación enunciaremos varios de nuestros tests:

### Caso 1: El valor de entrada $P$ es de la forma $3^i$ para un $i \leftarrow [0, N]$

Este caso se cumple cuando se recibe un  $P$  el cual al realizar nuestro primer ciclo que chequea cual es la potencia igual o mayor, termina siendo igual y de esta forma solo se itera una única vez el segundo y tercer ciclo.

### Caso 2: El valor de entrada $P$ es de la forma

$$\sum_{i=1}^n 3^i = P$$

Veremos más adelante que este caso será el peor a resolver ya que se iterará la totalidad completa de elementos de nuestros arrays.

### Caso 3: El valor de entrada $P$ es múltiplo de 3

Este caso se cumple cuando se recibe un  $PMOD3 = 0$ .

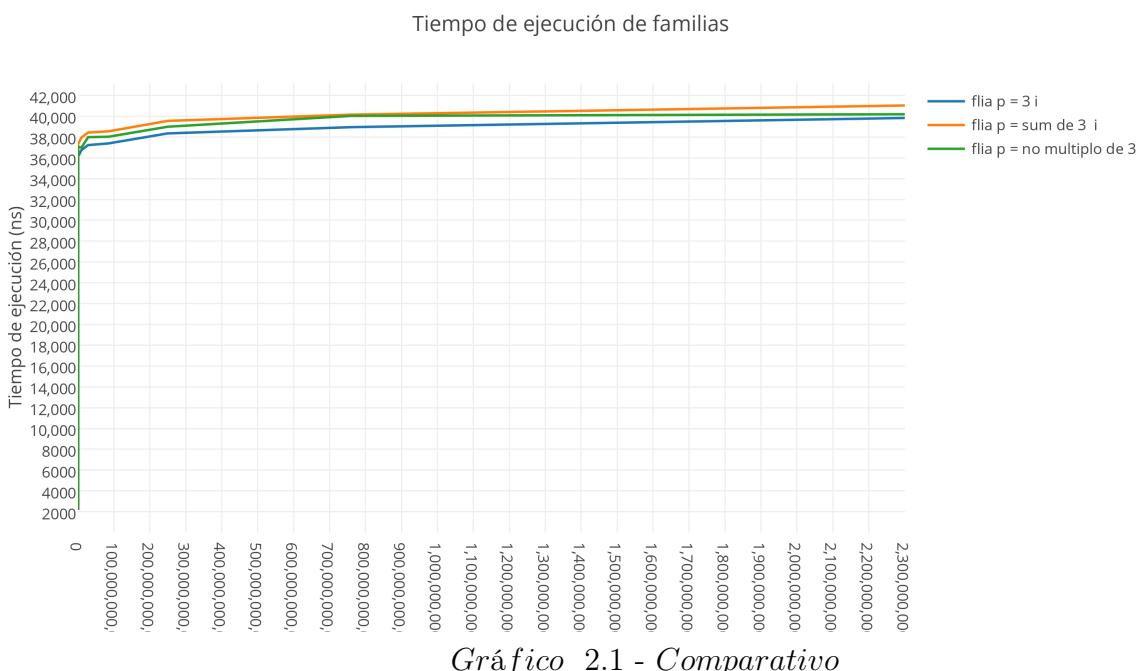
### Caso 4: El valor de entrada $P$ no es múltiplo de 3

Este caso se cumple cuando se recibe un  $P$  el cual el mismo es de la forma  $PMOD3 = 1$  o  $PMOD3 = 2$ .

#### 3.6.2 Performance De Algoritmo y Gráfico

Acorde a lo solicitado, mostraremos distintos tipos de familias de casos para nuestro algoritmo, y además, daremos el tiempo estimado según la complejidad calculada anteriormente.

A continuación mostraremos un gráfico de tiempos comparativo entre distintas familias de casos:



Se puede observar en el gráfico, tres funciones las cuales representan el tiempo de ejecución de las familias de casos:

- $P$  es igual a una potencia de 3

- 

$$\sum_{i=1}^n 3^i = P$$

- $P$  no es multiplo de 3

Como se observa en el gráfico la función representativa de la fña número 1, presenta una mejor performance en relación a las otras. Esto se debe a que nuestro algoritmo como va realizando sumas parciales y chequeando las potencias ve que la entrada es exactamente una única potencia por lo cual toma el valor final de la suma y finaliza su ejecución, mientras que para el segundo y tercer caso, como las entradas estan dadas por varias combinaciones de potencias indudablemente se necesitará para obtener dicha combinación recorrer todo el arreglo de sumas parciales más de una vez.

Luego de chequear dichas instancias, pudimos llegar a la conclusión que la familia de casos que presenta una mejor performance es en la cual se recibe  $P$  con un valor exactamente igual a  $3^i$  con  $0 \leq i \leq n$

Para llegar a dicha conclusión trabajamos con 30 instancias ya que  $3^{30}$  es el ultimo valor dentro del valor que puede tomar  $P$ .

Para una mayor observación desarrollamos el siguiente gráfico con las instancias:

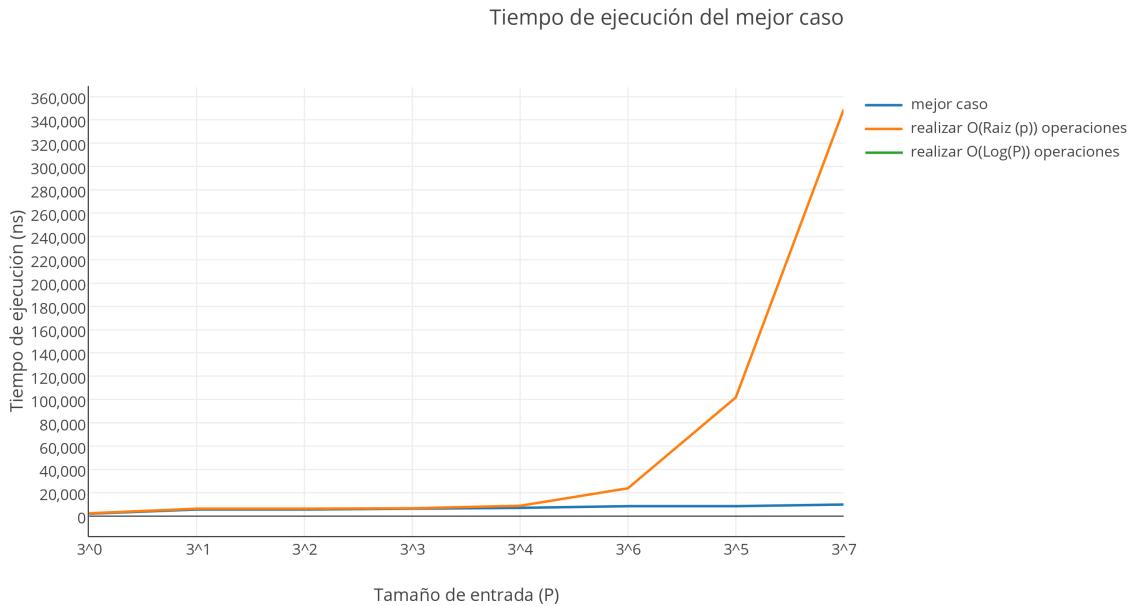


Gráfico 2.1 - MejorCaso

Como es posible observar en el gráfico, la función resultante de la cota teórica crece mucho más rápido que la de nuestro algoritmo la cual se tiende a ser constante, es por esto que a la hora de realizar el gráfico de mediciones se tomaron los primeros valores debido a que, cuando la función de  $\sqrt{P}$  toma valores muy grandes la misma es imposible de graficar contra nuestro algoritmo.

Además, como se puede apreciar debido a lo comentado no es posible apreciar las diferencias entre nuestro algoritmo y la cota  $O(\log(P))$  es por esto que mostraremos un gráfico entre ambos:

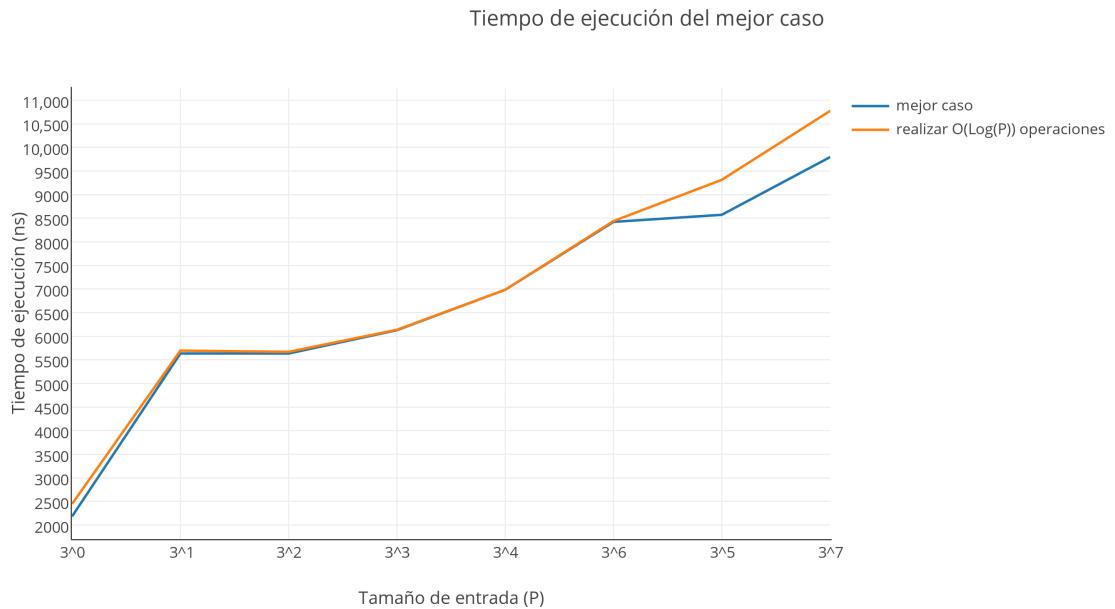


Gráfico 2.3 - MejorCaso

Luego, dividiendo por la complejidad teórica de nuestro algoritmo llegamos a:



Gráfico 2.4 - MejorCaso/Complejidad  $O(\sqrt{P})$

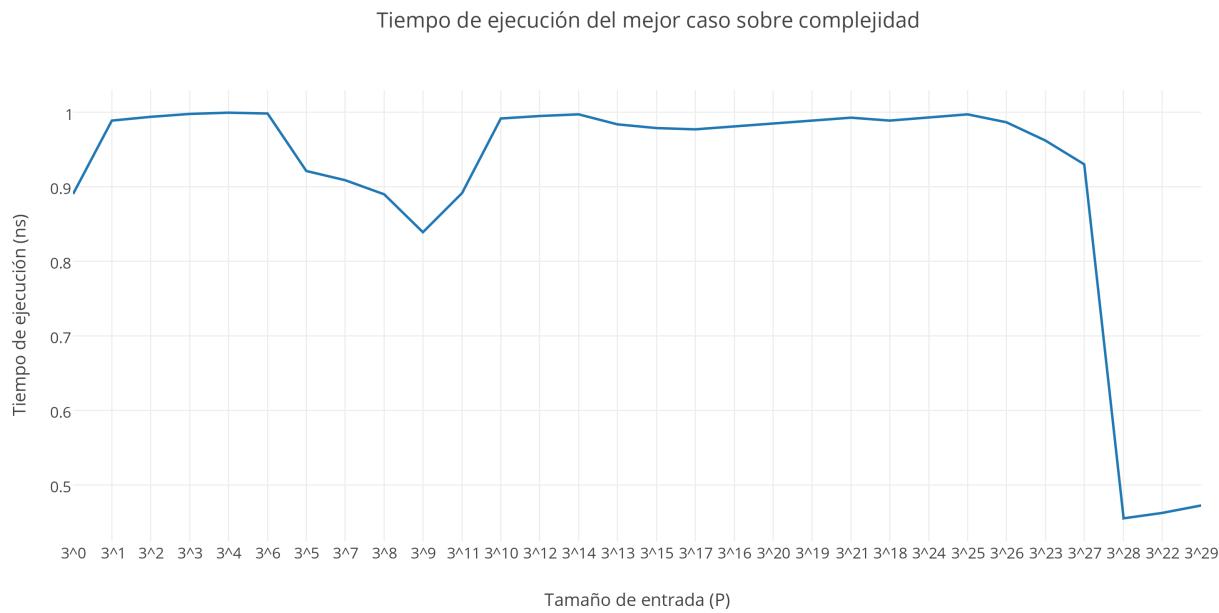


Gráfico 2.4 - MejorCaso/Complejidad  $O(\log(P))$

Para realizar esta experimentación nos parecio prudente, realizar un promedio con el mismo input de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más relevantes.

Se puede observar en el gráfico 2.4, como luego de realizar la división por la complejidad cuando el  $n$  aumenta el valor tiende a 0, tomando un valor máximo en 0.93. Y, en el gráfico 2.4 se ve como la función resultante tiene picos en los cuales llega a un máximo igual a 1 y cuando el valor de entrada aumenta la función tiende a 0.5. Por lo tanto, podemos concluir que para el mejor caso nuestro algoritmo se encuentra considerablemente por debajo de la cota teorica  $O(\sqrt{P})$  y asintotizado por la cota  $O(\log(P))$

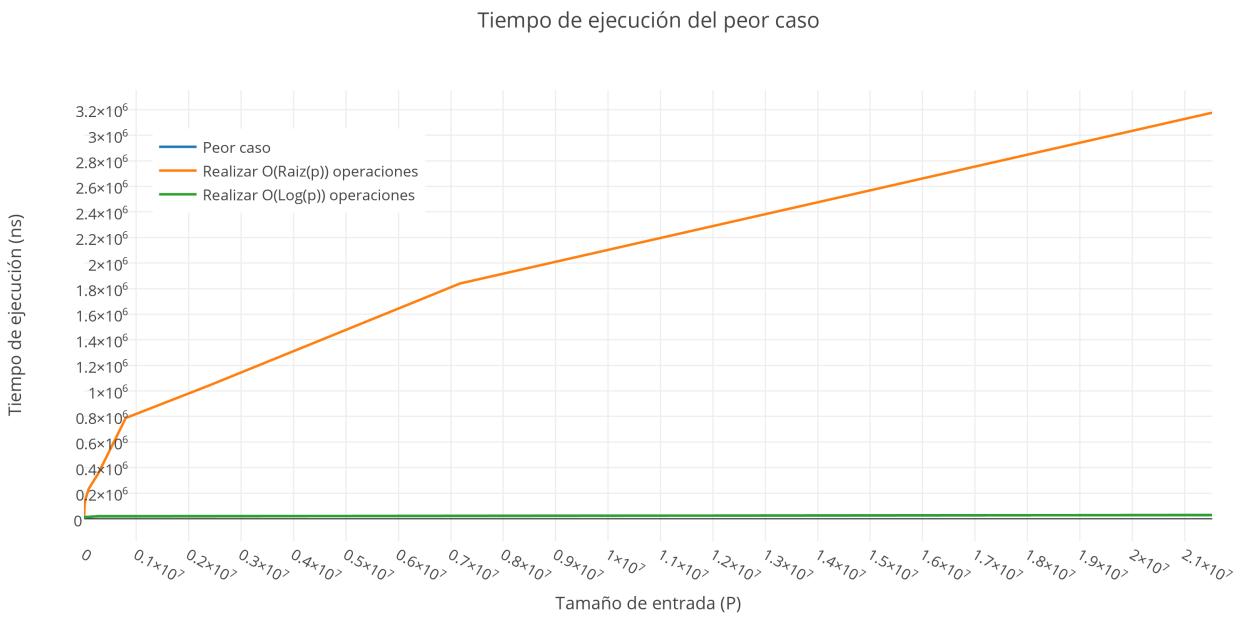
Luego, verificando el peor caso, llegamos a la conclusión que la familia de casos con las que resulta menos beneficioso trabajar será cuando el valor de entrada  $P$  sea de la forma

$$\sum_{i=1}^n 3^i = P$$

Realizando experimentos con un total de 20 instancias donde

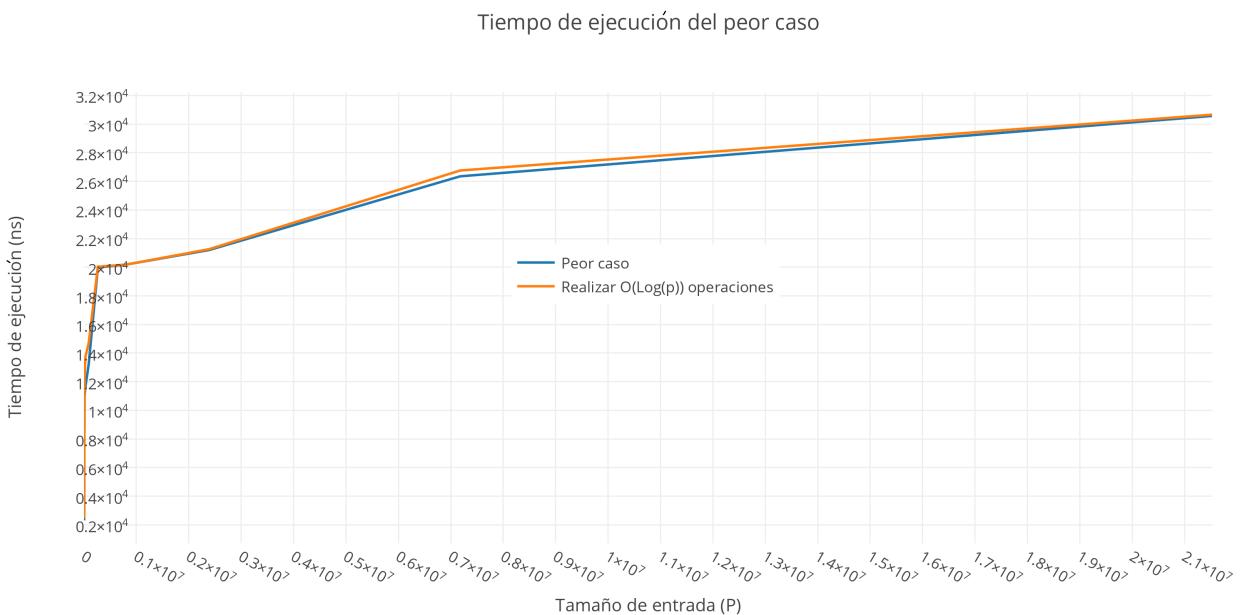
$$\sum_{i=1}^{20} 3^i = 5230176601$$

, desarrollamos dos gráficos los cuales mostraremos a continuación:



*Gráfico 2.5 - PeorCaso*

Se puede observar como la función representante de nuestro algoritmo y de  $O(\log(P))$  es mucho mejor que  $O(\sqrt{P})$  es por esto que realizamos un gráfico entre nuestro algoritmo y  $O(\log(P))$



*Gráfico 2.6 - PeorCaso*

Dividiendo por la complejidad propuesta llegamos a:

Tiempo de ejecución del peor caso sobre complejidad

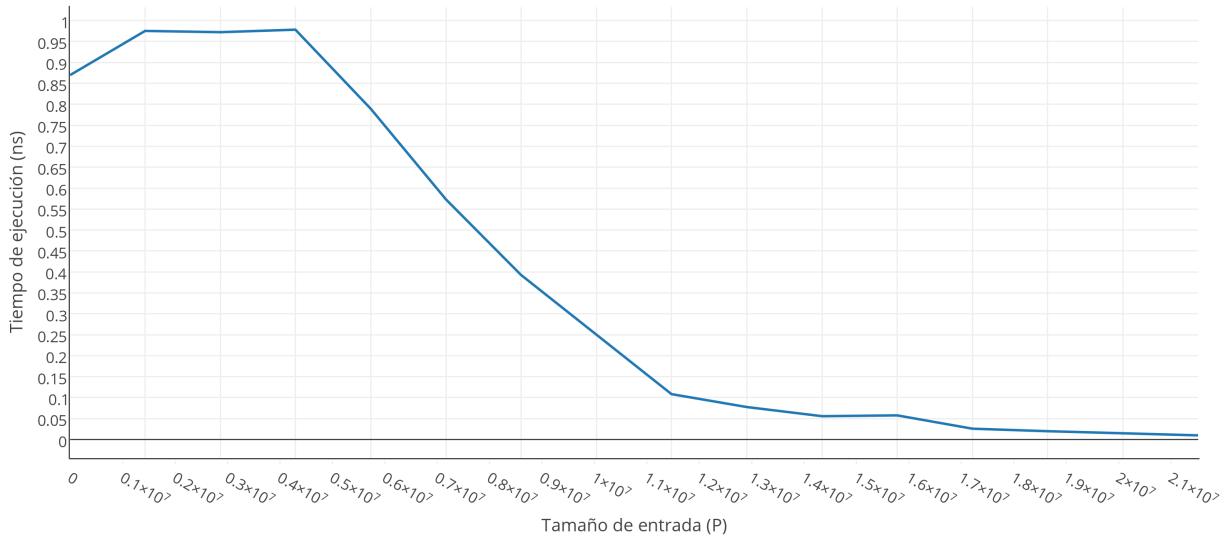


Gráfico 2.7 - Peor Caso/Complejidad  $O(\sqrt{P})$

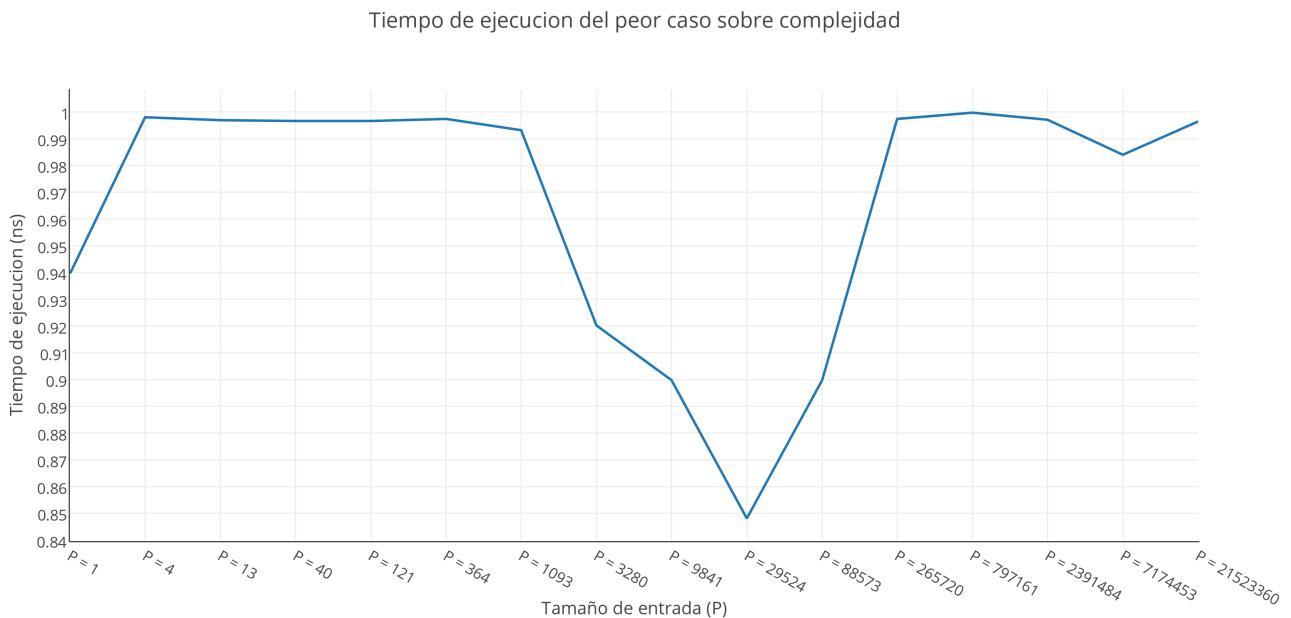


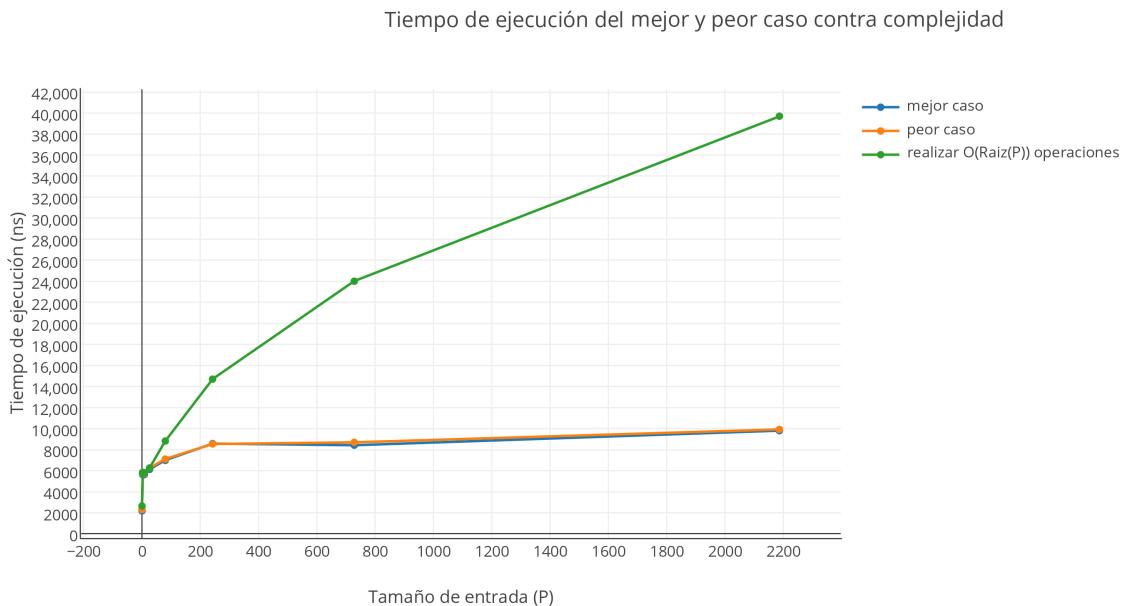
Gráfico 2.8 - Peor Caso/Complejidad  $O(\log(P))$

Para realizar esta experimentación nos parecio acorde, realizar un promedio con el mismo input de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más relevantes.

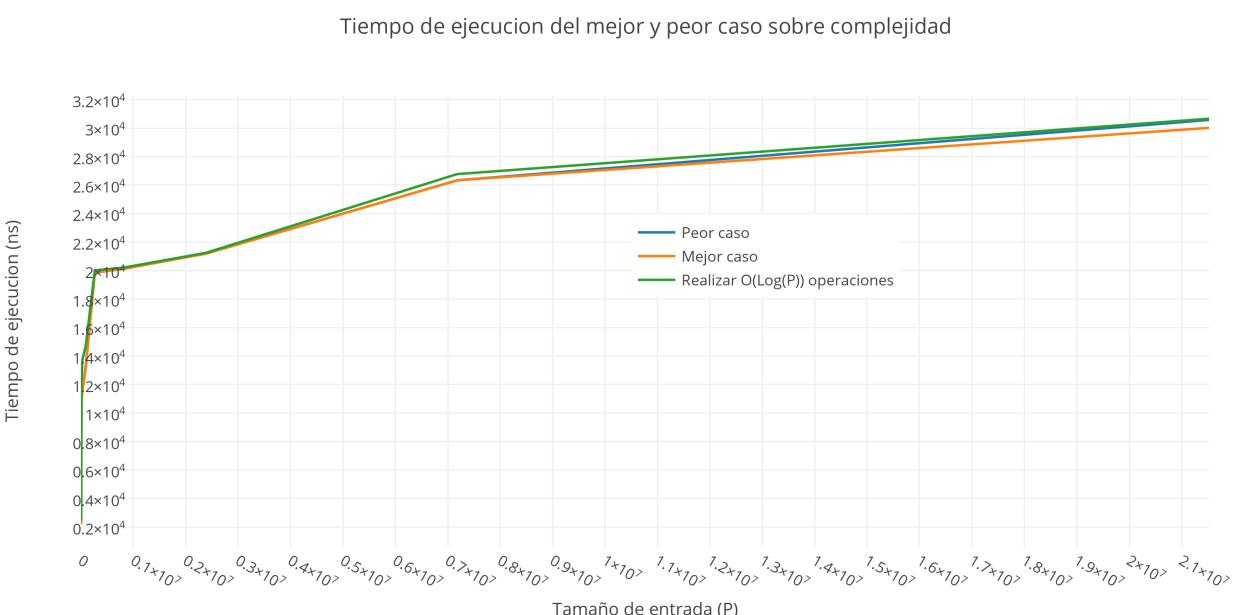
Como se puede observar en el gráfico 2.6 y 2.7, la función resultante de nuestro algoritmo es considerablemente mejor que la de la cota teórica  $O(\sqrt{P})$  y presenta un tiempo similar al de la función resultante de la cota  $O(\log(P))$ . Al igual que en el mejor caso, se gráfican las primeras

instancias para que el gráfico pueda ser más legible. Luego, en el gráfico 2.8 a pesar de tardar varios nanosegundos más que en el mejor caso, al dividir por la complejidad teórica la función resultante también tiende a 0 quedando comparativamente por encima del mejor caso.

Por último, mostraremos un gráfico comparativo entre el mejor y peor caso contra la complejidad que se solicito y la demostrada la cual es considerablemente inferior:



*Gráfico 2.8 - Comparativo*



*Gráfico 2.9 - Comparativo*

Luego de dichos experimentos y casos probados, se puede concluir que a pesar de utilizar todas las pesas como en el peor caso nos mantenemos dentro de la complejidad propuesta como habíamos mostrado en nuestro desarrollo de la complejidad.

## 4 Ejercicio 3

### 4.1 Descripción de problema

Luego de haber equilibrado la balanza, Indiana y compañía llegan a una habitación la cual se encuentra repleta de objetos valiosos.

Indiana y el grupo poseen varias mochilas las cuales soportan un peso máximo.

Nuestro objetivo en este ejercicio será ayudarlos a guardar la mayor cantidad posible de objetos valiosos en las mochilas teniendo en cuenta el valor de cada objeto y su peso.

### 4.2 Explicación de resolución del problema

#### Obteniendo el valor máximo

Si tenemos una sola mochila, nuestro algoritmo analiza para cada objeto el máximo valor que se puede obtener para cada capacidad posible de la mochila. Es decir, que si llamamos  $K$  a la capacidad total de la mochila y a  $k$  un número entero con  $1 \leq k \leq K$ , entonces para todos los  $k$  se evaluará introducir un elemento  $e$  en la capacidad posible  $k$ .

En cada evaluación se decide si me llevo el elemento en la mochila o no. Dado el caso:

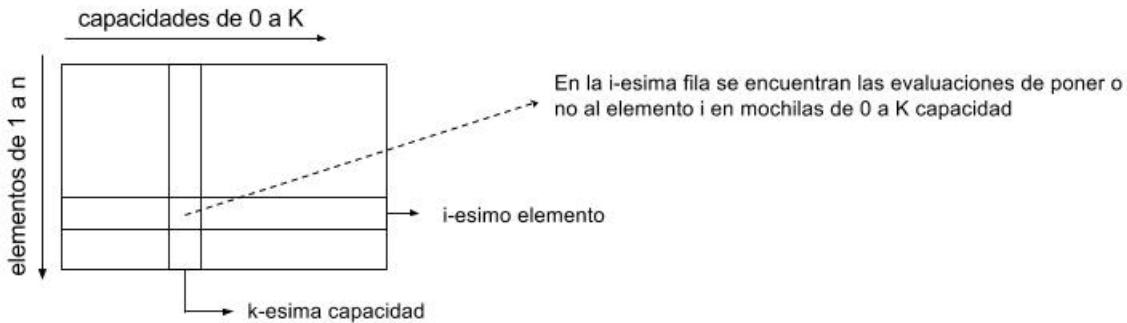
- Decido no meter el elemento: Se calcula el valor máximo de la mochila que tenga la capacidad  $k$  será el calculado solamente usando elementos anteriores. El valor será cero si no hay elementos anteriores o no es posible insertarlos en esa capacidad.
- Decido meter el elemento en la mochila: Al valor del elemento se le suma el valor máximo de la mochila de capacidad  $k - \text{peso}(e)$  y eso resulta en el valor máximo de la mochila con capacidad  $k$ .

De esta forma, se genera un subproblema que respeta el principio de optimalidad, con lo cual se puede aplicar programación dinámica para solucionar el problema.

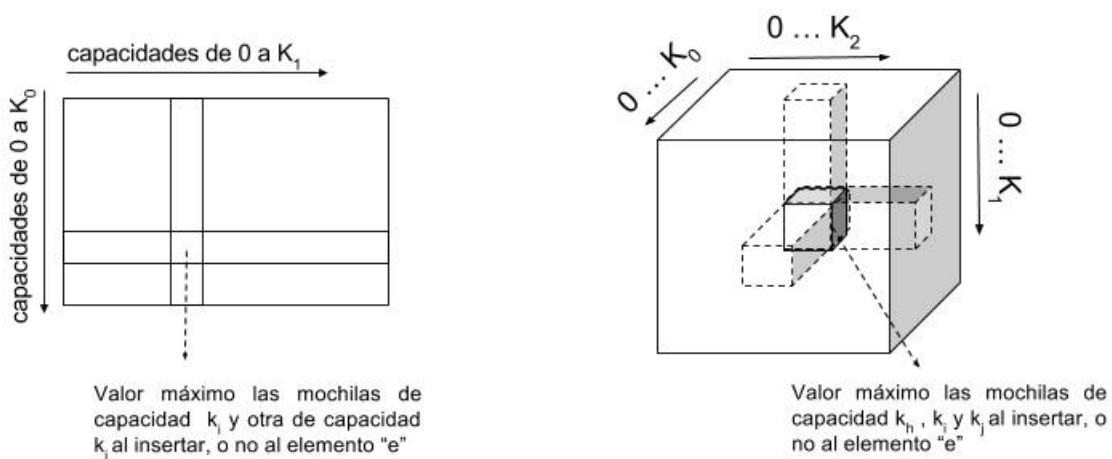
En el caso de 2 mochilas, al evaluar la  $k$  posibilidad de la primer mochila (con capacidad  $k_1$ ), se debe tener en cuenta que tambien está la posibilidad de utilizar la segunda mochila (con capacidad  $k_2$ ). Esto nos muestra que por cada elemento debemos calcular  $k_1 \times k_2$  combinaciones de capacidades. Siguiendo la idea del algoritmo original para una mochila, se calcularán sus combinaciones en base a las del elemento que le precedió en la evaluación. Para obtener el valor definitivo, luego de evaluar hasta el último elemento se selecciona la combinación de capacidades entre ambas mochilas que resulte máxima.

Como explicamos antes, ibamos a resolver este problema mediante programación dinámica. Esto quiere decir que necesitamos almacenar resultados de subproblemas previamente calculados. Nuestro subproblema es hallar el valor máximo para la suma de los valores de los objetos para un subconjunto de los elementos y con ciertas capacidades en las mochilas. Siendo  $M$  la cantidad de mochilas disponibles, debemos poseer una matriz de  $M$  dimensiones para cada objeto, permitiendo así guardar un valor máximo para cada combinación posible de capacidades de las mochilas.

### Problema de la mochila



### Evaluación del elemento "e" para 2 y 3 mochilas



Con la misma lógica podemos ver que para 3 mochilas donde se tienen capacidades  $K_1, K_2$  y  $K_3$ , cada elemento disponible evaluará  $K_1 \times K_2 \times K_3$  posibilidades. En este caso se buscará la solución en la matriz tridimensional de posibilidades obtenidas del último elemento evaluado.

### Recuperando los elementos utilizados

Siendo que la consigna pide los objetos involucrados en la solución, es necesaria una forma de, a partir de los valores máximos obtenidos, deducir los elementos que fueron usados y el lugar donde fueron colocados para lograr el resultado.

Analizando la última matriz del caso de 2 mochilas (sin pérdida de generalidad para 3 mochilas). Por cada objeto iterado, notamos las siguientes posibilidades en una celda específica de su matriz:

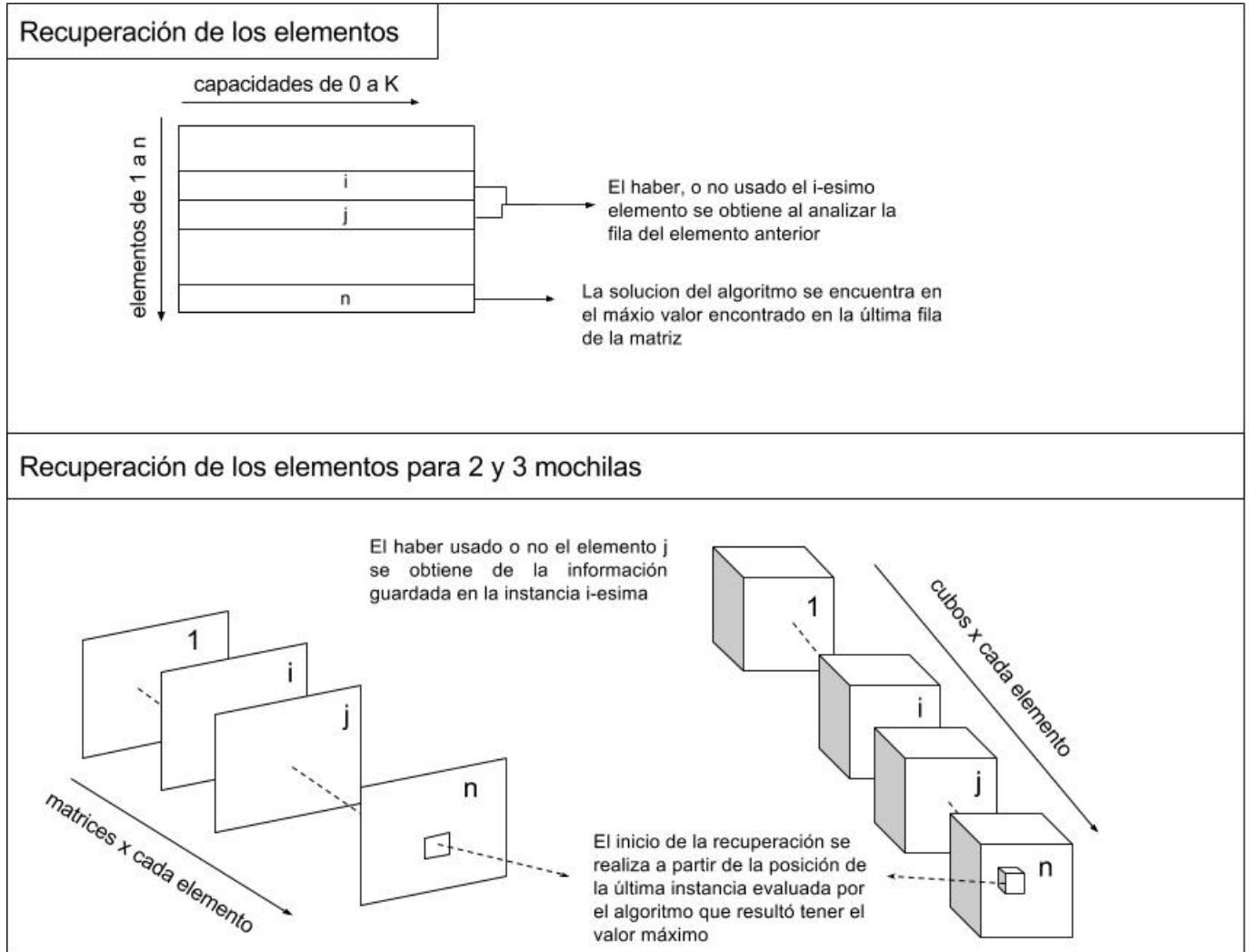
- Si el objeto no es utilizado en ese caso, el valor de la celda es igual al valor de la celda correspondiente en la matriz del objeto anterior.
- Si el objeto es utilizado, el valor de la celda es un nuevo máximo alcanzado. Por lo tanto el valor de la celda del objeto actual difiere (y en particular, es mayor) con el valor correspondiente al objeto anterior.

De utilizar 1 matriz solamente, y actualizarla por cada objeto, se perderá la posibilidad de saber si se usó o no a cada uno, ya que no se tendrá un "paso anterior" por haber sido reescrito en cada

paso subsiguiente.

Para recuperar estos estados, se resuelve guardar la matriz involucrada en el cálculo de cada elemento. Llamando a las matrices del elemento  $i$   $M_i$ , y  $N$  a la del elemento  $j$  subsiguiente a  $i$  (ambas de dimensión  $K_a \times K_b$ ), se tiene que si el máximo se encuentra en  $N_{yz}$  y es igual a  $M_{yz}$ , entonces quiere decir que el algoritmo al evaluar la posibilidad de meter al elemento  $j$  en esa combinación, optó por excluirlo, con lo cual no es perteneciente a la solución final. Si en cambio el valor es superior, indica que el valor guardado en la instancia  $M$  fue incrementado, y usado a  $j$  en la solución.

Finalmente, para saber en qué mochila fue introducido cada elemento, se busca en la instancia  $M$ , mochila a cuya capacidad se le restó el peso  $p(j)$  de  $j$ . Esto lo podemos saber cuando el valor de  $N_{yz} = M_{(y-p(j),z)}$  en el caso de que se encuentre en la mochila "a" o  $N_{yz} = M_{(y,z-p(j))}$  en caso contrario.



Para el caso de 3 mochilas, no se pierde generalidad: En vez de contar con matrices bidimensionales se utilizan tridimensionales, y se compara  $N_{xyz} = M_{(x-p(j),y,z)}$ ,  $N_{xyz} = M_{(x,y-p(j),z)}$ ,  $N_{xyz} = M_{(x,y,z-p(j))}$  para determinar la pertenencia de cada objeto a la solución.

### 4.3 Algoritmos

A continuación se detalla el pseudo-código de los algoritmos que resuelven el problema:

```

función BuscarMaximo()
    Para cada objetoActual ∈ objetos hacer
        objetoActual es último objeto que no fue optimizado
        matrizActual es una matriz de tamaño  $K_1 \times K_2 \times K_3$ 
        copiar a matrizActual la matriz de optimización de la iteracion previa  $O(K_1 \times K_2 \times K_3)$ 
        entero x ← capacidad(Mochila3)  $O(1)$ 
        Mientras  $x \geq 0$  hacer
            entero y ← capacidad(Mochila2)  $O(1)$ 
            Mientras  $x \geq 0$  hacer
                entero z ← capacidad(Mochila1)  $O(1)$ 
                Mientras  $z \geq 0$  hacer
                    entero m1 ← matrizActual[x-pesoObjetoActual][y][z]  $O(1)$ 
                    entero m2 ← matrizActual[x][y-pesoObjetoActual][z]  $O(1)$ 
                    entero m3 ← matrizActual[x][y][z-pesoObjetoActual]  $O(1)$ 
                    si objeto optimiza máximo total poniendolo en mochila1 entonces
                        fin si
                        guarda:  $O(1)$ 
                        para matrizActual[x][y][z] asignar m1 + valorObjetoActual  $O(1)$ 
                    sinó, si objeto optimiza máximo total poniendolo en mochila2 entonces
                        fin si
                        guarda  $O(1)$ 
                        para matrizActual[x][y][z] asignar m2 + valorObjetoActual  $O(1)$ 
                    sinó, si objeto optimiza máximo total poniendolo en mochila3 entonces
                        fin si
                        guarda  $O(1)$ 
                        para matrizActual[x][y][z] asignar m3 + valorObjetoActual  $O(1)$ 
                    si matrizActual[x][y][z] es nuevo máximo entonces
                        guardar nuevo máximo y las posiciones
                    fin si
                    decrementar z  $O(1)$ 
                fin ciclo
                decrementar y  $O(1)$ 
            fin ciclo
            decrementar x  $O(1)$ 
        fin ciclo
    fin para
    llenarMochilas()  $O(\#objetos)$ 
    devolver max
total: O(\#objetos × K1 × K2 × K3)
fin función

```

```

función llenarMochilas()
  Para cada matrizActual ∈ matrices hacer
    objetoActual es último objeto que optimizó matrizActual
    si matrizActual[x][y][z] ≠ matrizAnterior[x][y][z] entonces
      guarda: O(1)
      si el valor máximo obtenido viene de agregarlo en la mochila 1 de matrizAnterior
      entonces
        Mochila1 ∪ objetoActual
        guarda: O(1)
        fin si
        sinó, si el valor máximo obtenido viene de agregarlo en la mochila 2 de
        matrizAnterior entonces
          guarda: O(1)
          Mochila2 ∪ objetoActual
          guarda: O(1)
          fin si
          sinó, si el valor máximo obtenido viene de agregarlo en la mochila 3 de
          matrizAnterior entonces
            guarda: O(1)
            Mochila3 ∪ objetoActual
            guarda: O(1)
            fin si
            fin si
        fin para
        tota: O(#objetos)
      fin función

```

#### 4.4 Análisis de complejidades

El algoritmo por cada uno de los  $L = \sum_{i=0}^N C_i$  elementos construye una matriz de  $K_1 \times K_2 \times \dots \times K_m$ , y las guarda para poder recuperar al final, los elementos involucrados en la solución. El costo total será:

$$L * \prod_{i=1}^M K_i$$

Dentro del algoritmo, por cada objeto a optimizar, se copiará la matriz de la optimización hecha con los objetos del paso anterior con lo cual tendremos un costo por iteración de:

$$\prod_{i=1}^M K_i$$

Como el algoritmo simplemente recorre toda la matriz por cada objeto actualizando los valores con operaciones en  $O(1)$  el costo total final para todos los objetos será de:

$$L * \prod_{i=1}^M K_i$$

Si acotamos las capacidades de las mochilas por la de aquella de mayor capacidad (capacidad K) se tiene:

$$\sum_{i=0}^N (C_i) * \prod_{i=1}^M K_i \leq L * \prod_{i=1}^M K_i = L * \prod_{i=1}^M K = L * K^M$$

Dado que  $K^M < (\sum K_i)^M$  podemos concluir que el algoritmo respeta la cota requerida:

$$\sum_{i=0}^N (C_i) * K^M \subseteq O(\sum C_i * (\sum K_i)^M)$$

## 4.5 Demostración de correctitud

El problema de la mochila, en cualquier dimensión, se basa en maximizar un valor total teniendo en cuenta una o varias capacidades de las cuales no podemos excedernos. Como queden completadas las mochilas involucradas en cuanto a peso se refiere no dice nada, a priori, del valor total.

La idea de aplicar una técnica de programación dinámica en un problema de optimización es que podemos maximizar este valor total teniendo en cuenta lo mejor que pudimos hacer para subproblemas más pequeños.

En el caso de una dimensión, estos subproblemas serán más fácil de ver, y por lo tanto abordaremos esta demostración desde la perspectiva de una mochila en primera instancia.

Nuestro subproblema en este caso es: que es lo mejor que se pudo lograr con  $i$  objetos y capacidad  $k$  con  $0 \leq k \leq K$ , siendo  $K$  la capacidad total de la mochila. Puede describirse de la siguiente manera:

$$S(i, k) = \text{máximo beneficio obtenido con } i \text{ objetos para una capacidad disponible } k$$

A cada subproblema asociamos un objeto con el cual haremos un análisis exhaustivo de las posibilidades, las cuales serán, sea un objeto  $e_i$ , si utilizo o no utilizarlo dado que tengo  $k$  de capacidad disponible. Esta noción, junto con la idea de maximizar el valor total para la capacidad disponible nos da una idea de que es lo que tenemos que resolver en cada subproblema.

Si no ponemos el objeto, entonces deberemos ver que es lo mejor que se hizo con los objetos anteriores para esa misma capacidad, si no, al utilizar el objeto, tenemos que ver que es lo mejor que se logró sumando el valor del objeto actual  $\text{valor}(e_i)$  al valor dado por el resultado de lo mejor que se pudo lograr con los objetos anteriores para una capacidad de mochila disponible menor, la cual será  $k - \text{peso}(e_i)$  siempre que el peso del objeto sea menor que la capacidad disponible, dado que si no, estamos obligados a quedarnos con lo mejor que se pudo realizar con los anteriores  $i - 1$  objetos y la misma capacidad.

Luego, como los resultados a los subproblemas están bien definidos y son óptimos, el resultado para  $S(i, k)$  será quedarse con el mejor de los dos.

Lo que nos queda es un máximo que se calcula de la siguiente manera:

$$S(i, k) = \max(S(i - 1, k - \text{peso}(e_i)) + \text{valor}(e_i), S(i - 1, k)) \quad (13)$$

La solución a nuestro problema será el resultado de resolver  $S(n, K)$ , siendo  $n$  la cantidad de objetos total, la cual será óptima para  $n$  objetos.

Como puede verse, en cada paso, para un objeto se puede determinar que acción realizar, con esta misma idea y con la ayuda de una matriz que nos permite hacer uso de memorización de cada

paso, podemos obtener que objetos estuvieron involucrados en el óptimo final.

Para el caso de dos o tres mochilas estamos ante un problema de similar resolución, pero con más complicaciones relacionadas al logro de memorización para no perder información de cada paso realizado, la cual es fundamental para la obtención de los objetos de cada mochila.

Aquí la idea es obtener un máximo general sin importar si las mochilas maximizan o no por su parte.

Por cada objeto tendremos que decidir si no lo usamos, o en cual de las mochilas disponibles nos conviene ingresarla y esas son todas las opciones disponibles para un objeto.

Para simplificar la notación, utilizaremos el caso de dos mochilas, aunque lo que sigue es fácilmente adaptable a tres.

Como mencionamos el subproblema que analizamos es:

$$S(i, k_1, k_2) = \text{máximo beneficio obtenido con } i \text{ objetos dado que dispongo de dos capacidades } k_1 \text{ y } k_2$$

Si se ingresa el objeto  $i - \text{esimo}$  en la mochila  $j - \text{esima}$ , entonces tendremos que tomar lo mejor realizado con  $i - 1$  objetos y capacidad  $k_j - \text{peso}(e_i)$  y las capacidades restantes sin modificar siempre y cuando  $k_j \geq \text{peso}(e_i)$ . Si no se ingresa el objeto, se toma lo mejor realizado con  $i - 1$  objetos y todas las capacidades disponibles.

Como cada subproblema está bien definido y es óptimo, solo se tiene que elegir el mejor de todos ellos mediante un máximo.

$$S(i, k_1, k_2) = \quad (14)$$

$$\max(S(i - 1, k_1 - \text{peso}(e_i), k_2) + \text{valor}(e_i), S(i - 1, k_1, k_2 - \text{peso}(e_i)) + \text{valor}(e_i), S(i - 1, k_1, k_2)) \quad (15)$$

Por lo cual, la respuesta a este problema será  $S(n, K_1, K_2)$  siendo  $n$  la cantidad de objetos total y  $K_1$  y  $K_2$  las capacidades de las mochilas.

Además por lo explicado en la resolución del problema, también se hace uso de matrices para resolver los casos con dos y tres mochilas, pero cabe destacar el plural.

En un principio, para obtener el máximo, podríamos utilizar una única matriz. Pero dado que para cada objeto se la recorre completa, la información del máximo logrado para cada objeto puede no mantenerse, y además como los objetos no ocupan un lugar en la matriz, no podemos saber cual de estos lo logró.

Por lo tanto, hacemos uso de  $n$  matrices de dimensiones correspondientes a la capacidad de las mochilas involucradas.

De esta manera, el máximo logrado para  $i$  objetos estará alojado en la matriz correspondiente al objeto  $i - \text{esimo}$ .

De aquí en más, lo visto en la explicación ayuda a obtener los objetos en los casos para una y dos o tres dimensiones, teniendo en cuenta que es lo que sucede o no al tomar un objeto.

Por lo tanto, podemos asegurar que el máximo logrado es el óptimo y que los objetos pueden obtenerse como una consecuencia de las elecciones realizadas y la información correctamente almacenada.

## 4.6 Experimentos y conclusiones

### 4.6.1 Test

Para corroborar el correcto funcionamiento de nuestro algoritmo implementado desarrollamos los siguientes tests:

Dentro de los casos en los cuales las capacidades de las mochilas pueden ser o no idénticas y los objetos son iguales o no, pudimos separar ciertas familias de casos puntuales:

#### **Caso 1: No entra ningún objeto en las mochilas**

Este caso se da cuando  $P_i > K_j$  con,  $1 \leq i \leq N$  y  $1 \leq j \leq 3$

#### **Caso 2: Entra un único objeto en cada mochila**

Este caso se da cuando existe un único objeto para cada mochila que presenta un peso menor a la capacidad de las mismas

#### **Caso 3: Entra un único objeto en total**

Este caso se da cuando existe un único objeto que entra en alguna de las 3 mochilas, dejando así dos mochilas totalmente vacías.

#### **Caso 4: Entra una cantidad par de objetos en las mochilas**

#### **Caso 5: Entra una cantidad dispar de objetos en las mochilas**

Este caso, tambien lo denominaremos random ya que existirá la posibilidad que entren de 0 a n objetos en las mochilas

#### **Caso 6: Entran todos los objetos en las mochilas**

Este caso se da cuando todos los objetos ingresan en las mochilas de alguna manera óptima, obteniendo así la cantidad máxima posible de objetos la cual será la totalidad de los mismos

### 4.6.2 Performance De Algoritmo y Gráfico

En esta sección, mostraremos buenos y malos casos para nuestro algoritmo, y a su vez, daremos el tiempo estimado según la complejidad calculada anteriormente.

Como la complejidad presenta dos variables marcadas, como son las capacidades de las mochilas y la cantidad de objetos, hemos trabajado con las mismas en simultaneo y también fijando alguna de las dos variables.

Chequeando la familia de casos que enunciamos en el anterior inciso pudimos llegar a la conclusión que teniendo capacidades identicas en las mochilas o distintas nuestro algoritmo se comporta a nivel tiempo de ejecución de una manera similar, por lo tanto trabajaremos a continuación con capacidades distintas.

A continuación mostraremos dos gráficos que simbolizan el tiempo de ejecución de cada una de las familias de casos:

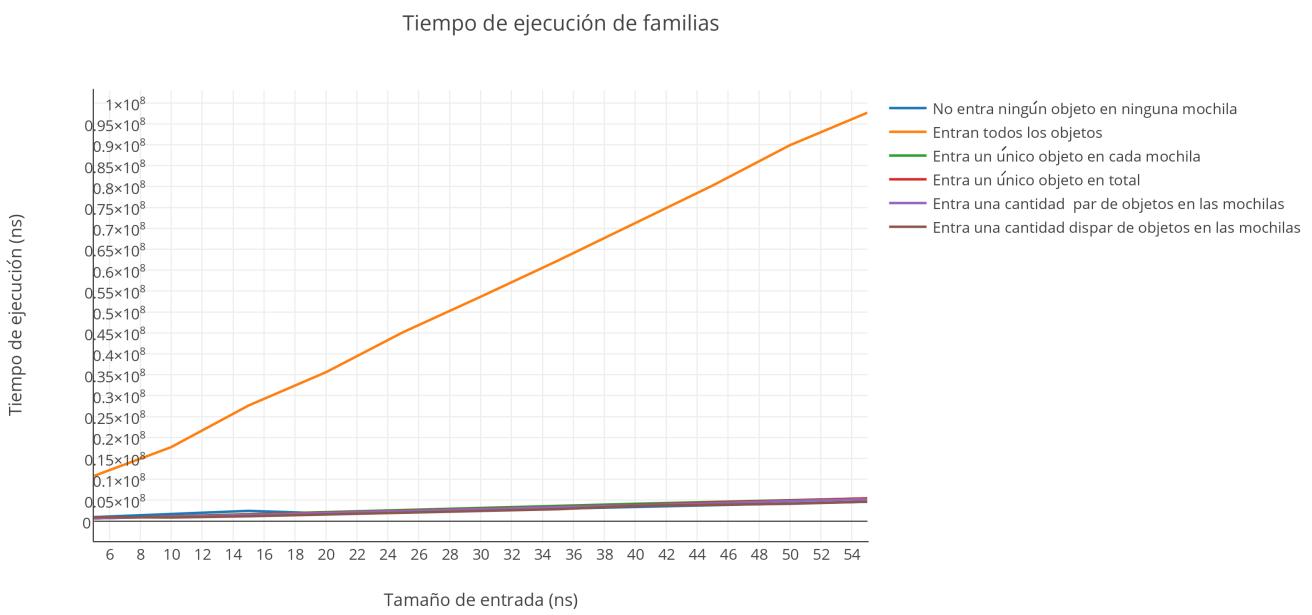


Gráfico 3.1 - Comparativo

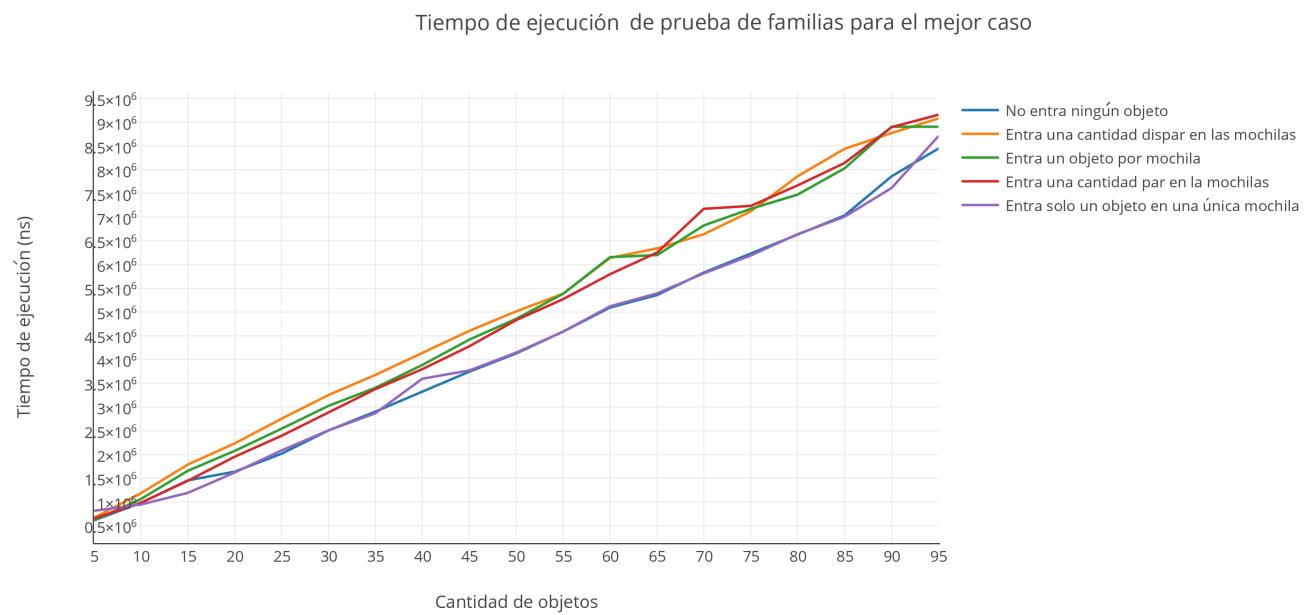


Gráfico 3.2 - Comparativo 2

Las familias de casos que fueron representadas en los gráficos han sido las siguientes:

- No entra ningún objeto en las mochilas
- Entra un único objeto en cada mochila
- Entra un único objeto en total
- Entra una cantidad par de objetos en cada mochila

- Entra una cantidad dispar de objetos en cada mochila
- Entran todos los objetos en las mochilas

Se puede observar en el gráfico 3.1 como la familia 6 de casos crece mucho más rápido que el resto de las familias generando así que en el gráfico 3.1 no se pueda distinguir de buena forma el resto de las 6 familias, es por esto que en el gráfico 3.2 solo fueron representadas estas 6.

Se puede observar entre el gráfico 3.1 y 3.2 en los cuales la familia **Entran todos los objetos en las mochilas** es el peor caso, debido a que nuestro algoritmo deberá chequear por cada objeto en que mochila colocarlo ya que tendrá la posibilidad de maximizar la totalidad colocando al objeto en alguna de las 3.

Luego, en el gráfico 3.2 se puede observar como tanto la familia de casos en las que no entra ningún objeto o entra únicamente uno se ven más beneficiadas por la implementación de nuestro algoritmo el cual verificará si es posible o no meter al objeto en cuestión dentro de alguna de las mochilas, y como el peso del mismo es superior a la capacidad de las 3 mochilas directamente lo descarta sin tener que realizar el chequeo de maximización.

Podemos concluir entonces que, el mejor caso para nuestro algoritmo es en el cual **No entra ningún objeto en las mochilas** ya que como mencionamos, solo se verificará su peso y al ser mayor descartará al objeto y no tendrá que realizar los chequeos de maximización.

A continuación mostraremos un gráfico que representa lo enunciado:

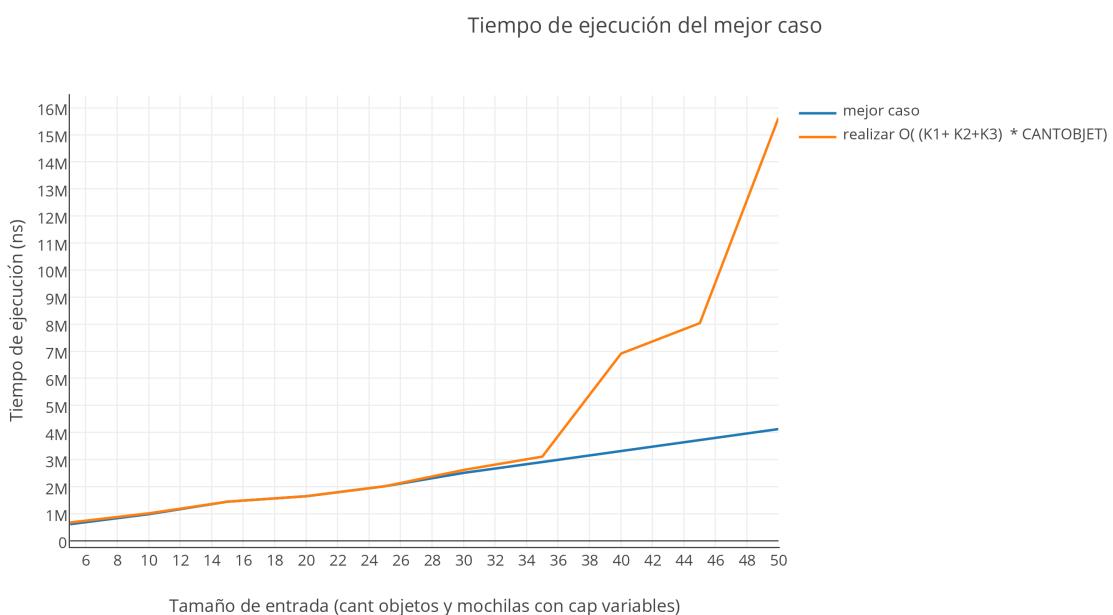
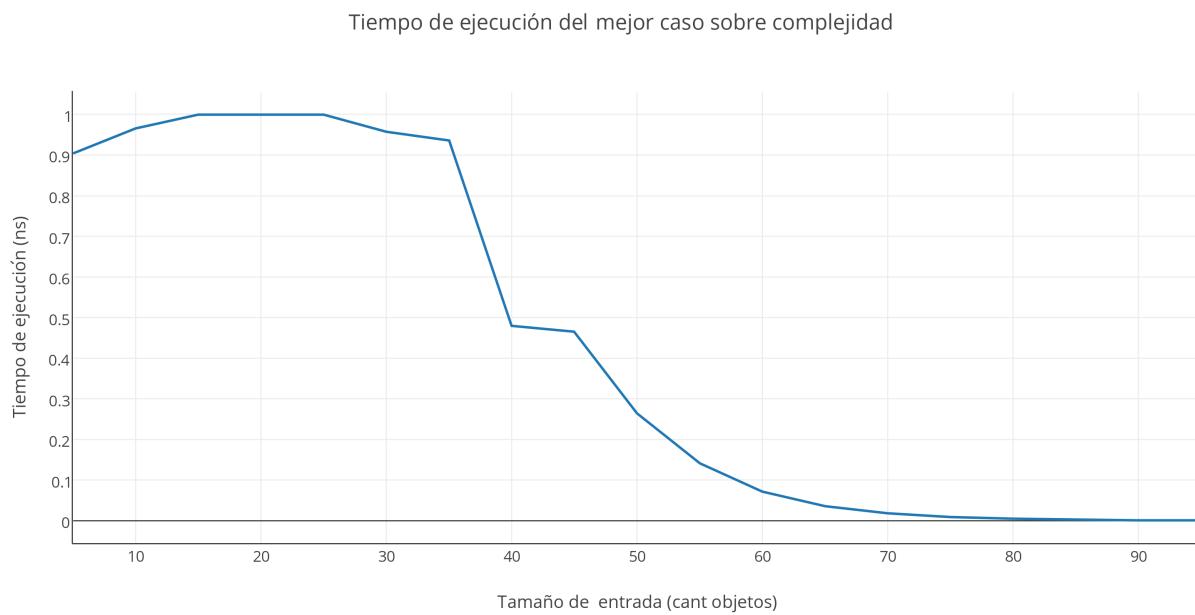


Gráfico 3.3 - Mejor caso del algoritmo

Dividiendo por la complejidad calculada se obtuvo lo siguiente:

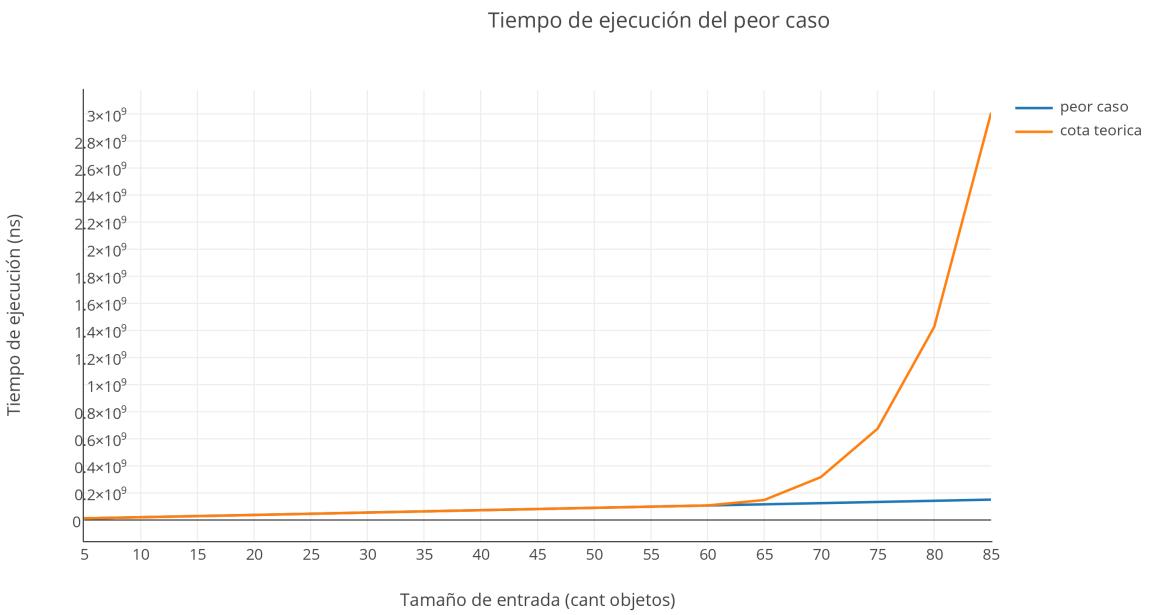


*Gráfico 3.4 - Mejor caso del algoritmo sobre complejidad*

Se puede ver en el gráfico 3.3 que, nuestro algoritmo, esta acotado por la función de la cota teórica, ya que el mismo realizara una cantidad sustancialmente menor de chequeos e iteraciones por verificar inicialmente si los objetos tienen la posibilidad o no de ingresar en las mochilas.

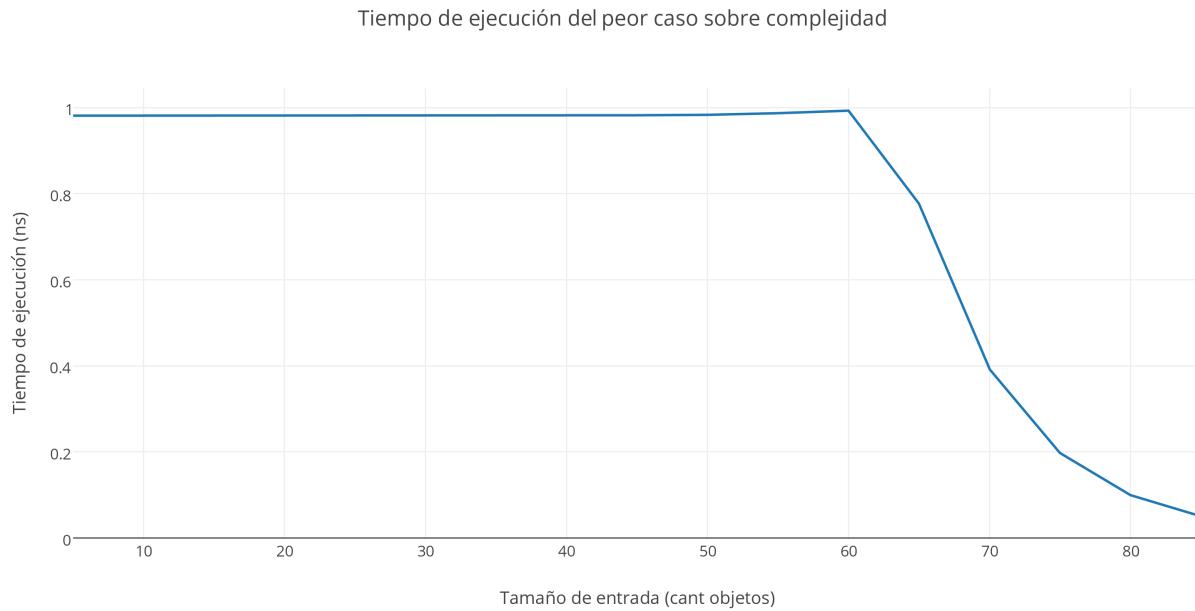
Luego, en el gráfico 3.4, en el cual dividimos el tiempo de ejecución de dicho caso con el tiempo de realizar  $O(\sum_{i=1}^3 K_i)$  operaciones nos da una función resultante la cual se encuentra por debajo de 1 y tiende a 0 cuando la entrada aumenta corroborando lo que enunciamos.

Manteniendo el mismo razonamiento, el peor caso se da cuando **Entran todos los objetos en las mochilas** ya que se deberá decidir en que mochila colocar el objeto para que el resultado sea óptimo. Por consiguiente, mostraremos un gráfico que representa lo hablado:



*Gráfico 3.5 - Peor caso del algoritmo*

Dividiendo por la complejidad calculada se obtuvo lo siguiente:



*Gráfico 3.6 - Peor caso del algoritmo sobre complejidad*

Es posible observar en el gráfico 3.4 que, al tener las posibilidades de ir metiendo cada uno de los objetos en las mochilas, el mismo realizará la optimización de 3 matrices que simbolizan a cada mochila anidadas, lo que nos dará una complejidad acotada por la capacidad de las mochilas por la cantidad de elementos que en esta evaluación es constante, simplificando lo dicho nos quedaría  $O(\sum_{i=1}^3 K_i)$

$$\sum_{i=1}^3 K_i$$

$*CantObjetos)$  .

Luego, en el gráfico 3.4, en el cual dividimos el tiempo de ejecución de dicho caso con el tiempo de realizar  $O(\sum_{i=1}^3 K_i$

$*CantObjetos)$  operaciones nos devuelve una función resultante la cual se encuentra por debajo de 1 y tiende a 0 cuando la entrada aumenta corroborando lo que acabamos de decir.

Por último un grafico comparativo entre el mejor y peor caso y la cota teorica nos otorga la siguiente percepción:

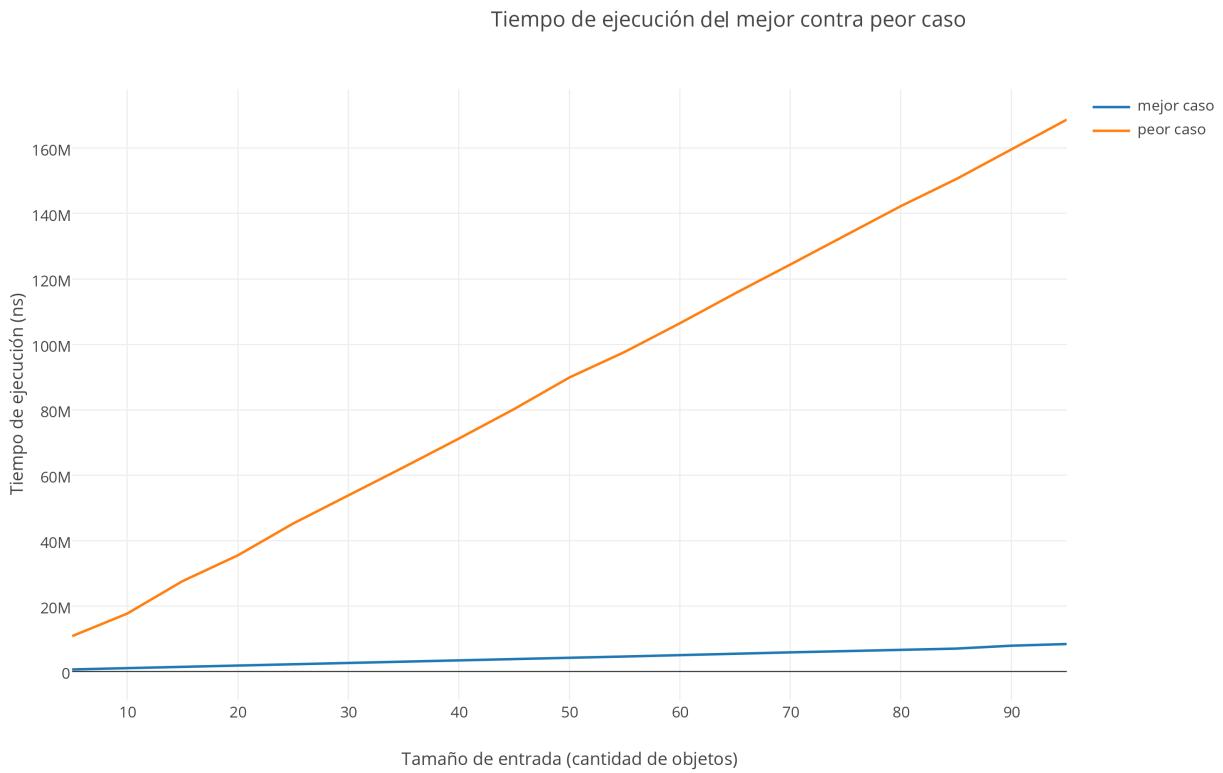


Gráfico 3.7 - Comparativo peor y mejor caso del algoritmo

### Tiempo de ejecución de familias contra cota

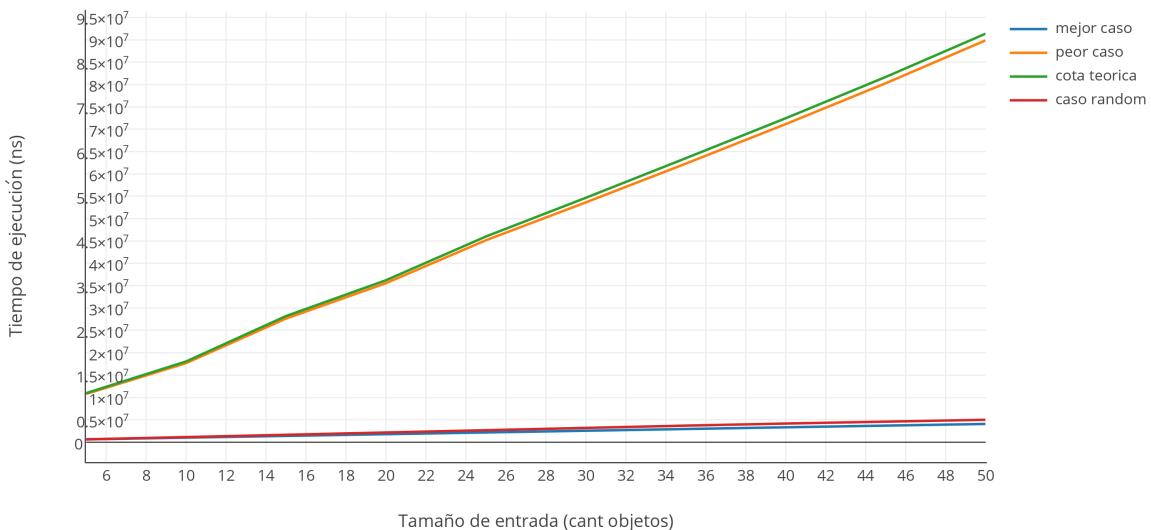


Gráfico 3.8 - Comparativo peor y mejor caso del algoritmo contra complejidad teórica

Es notorio que, en el gráfico 3.8 la funciones resultantes siempre se mantienen por debajo de la cota de complejidad calculada anteriormente, con lo cual se ve que tanto en el mejor como en el peor caso nuestro algoritmo sigue estando acotado por la misma.

## 5 Aclaraciones

### 5.1 Aclaraciones para correr las implementaciones

Cada ejercicio fue implementado con su propio Makefile para un correcto funcionamiento a la hora de utilizar el mismo.

El ejecutable para el ejercicio 1 sera ej1 el cual recibirá como se solicito entrada por stdin y emitirá su respectiva salida por stdout.

Tanto el ejercicio 2 como el 3, compilaran de la misma forma y podrán ser ejecutados con ej2 y ej3 respectivamente.