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

1 Ejercicio 1

1.1 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 cruze 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.

1.2 Explicación de resolución del problema

Para solucionar este problema, se debe 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 busqueda 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 busqueda del mínimo a travez de backtracking.

Para implementar la solución se atomizo cada ciclo a 1 solo viaje: el envío de gente de A a B o el regreso de 'faroleros' de B a A. De esta forma, el backtracking se puede realizar en la desición de la gente que va de A a B, independientemente de la elección de los faroleros.

Dadas las restricciones del problema, se pudieron aplicar las siguientes podas sobre el arbol de posibilidades:

- 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', es decir que en algun lado hay más canibales que arqueologos, son obviados.
- Las secuencias de viajes que tomen más tiempo que una previamente calculada se descartan.
- Solo se consideran viajes de A a B en duplas y de una sola persona de B a A.

En base a la evaluación de todas las soluciones encontradas por el algoritmo, nos quedamos con aquella que menor tiempo acumule con los viajes.

1.3 Algoritmos

Algoritmo 1 CRUZANDO EL PUENTE

```
1:  function EJ1(in : Integer, in : List<Integer>)→ out res: Integer
2:      creo bool exitoBackPar con valor verdadero                          //O(1)
3:      creo bool exitoBackLampara con valor verdadero                      //O(1)
4:      while exitoBackLampara ∨ exitoBackPar do                            //O(n!^3 * n)
5:          if ∃ parPosible() ∧ tiempoMinimo() then                         //O(C(n,2))
6:              par ← dameParPosible()                                      //O(1)
7:              enviarPar(par)                                              //O(1)
8:          else
9:              exitoBackLampara ← backtrackRetorno(farolero)               //O(1)
10:         end if
11:         if ∃ faroleroPosible() ∧ tiempoMinimo()  then                   //O(n)
12:             retornarLampara(farolero)                                   //O(1)
13:         else
14:             exitoBackPar ← backtrackPar(par)                            //O(1)
15:         end if
16:         if pasaronTodos() then                                          //O(1)
17:             guardarTiempo()                                             //O(1)
18:         end if
19:     end while
20: end function
```

Complejidad: $O(n!^3 * n)$

- **parPosible**: Se busca por un par de personas que esten en el lado A y que al pasarlos al lado B no generen un desbalance entre canibales y arqueologos. El tiempo de ejecución de esta función es lineal en la cantidad de pares validos existentes (es decir $\binom{n-i}{2}$ pares). Vale notar que cuanto más personas pasen para el lado B, la cantidad de pares disminuye.

- **dameParPosible**: Habiendolo calculado con "parPosible" solo se lo debe devolver

- **tiempoMinimo**: Devuelve verdadero si el tiempo de construccion de la solucion es menor al ya encontrado, de no serlo, fuerza al algoritmo a retroceder en la rama. Al registrar por cada paso efectuado el tiempo empleado, esta función solo compara el tiempo con el minimo logrado en tiempo constante.

- **enviarPar**: Si se tiene un arreglo de cada estado del Escenario, entonces este paso implica marcar la decisión tomada en el paso actual y avanzar 1 paso, todo esto pudiendose efectuar en tiempo constante.

- **backtracRetorno**: Devuelve a la instancia Escenario a un paso anterior, y devuelve FALSE si no fue posible realizar la acción (si no hay más ramas para seguir el backtracking). Retornar la instancia a un paso anterior, de tener guardado el estado anterior en un arreglo, es simplemente retroceder el arreglo 1 posición, osea tiempo constante.

- **faroleroPosible**: Similar a "parPosible", busca entre las personas que se encuentran en el lado B, aquellas que no hayan sido evaluadas para el cruce y que el cruce no genere un desbalance entre personajes. La búsqueda de esta persona es en tiempo lineal a la cantidad de personas totales; vale notar que la única vez que se ejecuta n-1 va ser cuando en el lado B estén todas las personas, la cantidad de faroleros a evaluar, comienza en el paso 1 con 1 farolero y aumenta hasta n-1 (de a 1 por paso).

- **retornarLampara**: De forma similar a "enviarPar", se marca la persona elegida de farolero en el estado del sistema y se avanza al siguiente estado, todo esto en tiempo constante.
- **backtrackPar**: Efectua las mismas operaciones que el "backtrackFarolero" en O(1)
- **pasaronTodos**: Chequea en tiempo constante si la cantidad de personas en el lado B es igual a la cantidad total de personas. Esto se puede lograr facilmente si se tiene un arreglo con la cantidad de personas en cada lado.
- **guardarTiempo**: Si el tiempo tomado en alcanzar la solucion en la rama fue menor a todos los ya obtenidos por analizar cada rama, guarda el valor. Dado que se guarda por cada paso tomado el tiempo que se toma en realizarlo, esta operacion solo compara el valor tomado contra el minimo obtenido, en O(1)

1.4 Análisis de complejidades

Realizando pasos 2-1, cada solución construída tendrá **a lo sumo n-1 pasos**, siendo n la cantidad total de personas en la comitiva.

En el primer paso se analizan $\binom{n}{2}$ parejas posibles a enviar y por cada una de ellas se eligirá de entre 2 personas para ser el farlero.

En el segundo paso se tendran $\binom{n-1}{2}$ parejas posibles y 3 faroleros para elegir.

Para el (i)-esimo paso se tendrán $\binom{n-(i-1)}{2}$ parejas e i+1 faroleros.

Por cada una de las posibilidades de cada paso, se deberan analizar la cantidad total de posibilidades que haya en el paso subsiguiente, para dar lugar asi al analisis de todas las combinaciones posibles, es decir: siendo que el segundo paso se ejecuta $2 * \binom{n}{2}$ veces, el tercero $2 * \binom{n}{2}\binom{n-1}{2} * 3$, podemos ver que en combinación, todos los pasos demandarían:

$$\prod_{i=1}^{n-1} \binom{n-(i-1)}{2} * (i+1) = \prod_{i=1}^{n-1} \binom{n-(i-1)}{2} * \prod_{i=0}^{n-1} (i+1)$$

$$n! \prod_{i=0}^{n-2} \binom{n-i}{2} = n! \prod_{i=0}^{n-2} \frac{(n-i)(n-(i-1))}{2}$$

$$\frac{n!}{2^{n-2}} . \prod_{i=0}^{n-2} (n-i) . \prod_{i=0}^{n-2} (n-(i-1)) = \frac{n!}{2^{n-2}} . n! . \frac{(n+1)!}{2} \leq \frac{n!^2 . (n+1)!}{2^{n-1}} \in O(n.n!^3)$$

NOTA: PREGUNTAR A LOS PROFES SI NO ES UNA COTA GROSERA, CAPAS SE PODIA PLANTEAR UNA MAS ADECUADA....LO CUAL NOS ROMPE LOS TESTS PERO NOS SALVA DE DESAPROBAR, SI LO VEMOS CON TIEMPO.

1.5 Demostración de correctitud

La elección de las podas tomadas en el backtracking se debieron a las siguientes razones:

Siendo el conjunto total de combinaciones existentes contenedor de tuplas de elementos repetidos, (por ejemplo, para las personas P1 y P2, existe la tupla <P1, P2> y <P2, P1>), es trivial ver que evaluar estos casos carece de sentido, ya que es exactamente el mismo resultado el que se obtendra con una que con otra.

La consigna del ejercicio plantea la necesidad de mantener un balance entre los personajes involucrados. En una aproximación con fuerza bruta, se desestimarían las soluciones que conlleven algun desbalance en la sucesion de traslados de los personajes, para lo cual primero se deberían calcular todas las posibilidades y luego analizarlas una por una, recorriendo cada uno de sus pasos, buscando algun desbalance. Para evitar tener que analizar, al final de la construccion de las soluciones, si son o no válidas, se decide chequear en cada paso de la construccion de cada solución, si el mismo produce una instancia balaceada. Al podar de esa forma, solo nos quedaremos al final del algoritmo con las soluciones que son validas.

Teniendo ya las soluciones luego de ejecutar el algoritmo, se debería encontrar aquella que produzca el mínimo tiempo de viaje de los personajes. Esto quiere decir que habrá soluciones que estemos desestimando que consumieron tiempo en ser construidas. Por lo tanto, al calcular una solucion factible, se guarda el tiempo logrado, y todas las siguientes soluciones, si al construirse sobrepasan este tiempo, entonces quedan automáticamente desestimadas.

Por ultimo se analizan las posibilidades que se tiene al efectuar un paso. Para esto se considera al envio de personas y al retorno del farol como un PASO, notando **n-m** al envio de n personas al lado "B" y al retorno de m personas al lado "A", dando lugar a las siguientes posibilidades: 1-1, 1-2, 2-1, 2-2.

Notemos que para hallar una solución, es necesario que, en algun punto, la cantidad de personas en el lado "A" decremente, y se incremente en el lado "B", de forma tal que terminen todas del lado "B"; en caso contrario la secuencia de pasos nunca terminará.
Observando las opciones disponibles, se observa que, por paso, puede decrementarse la cantidad de personas del lado "A" en 1 como máximo, con lo cual, en la sucesion de pasos que representa una solución, hay una subsucesion en la que cada paso decrementa en 1 la cantidad de personas en "A". Analizando los casos posibles, es evidente ver que el único paso que puede construir esta subsucesión es el 2-1, ya que los demás mantienen o incrementan la cantidad en "A".

Podemos ver, entonces, que de poder construir una sucesión plenamente con pasos 2-1 (pasos efectivos), ésta será minima en tiempo empleado. Utilizar backtracking solo con el 2-1 (backtracking 2-1), entre 2 pasos consecutivos, se evalúan todas las combinaciones de 2 conjuntos con diferencia en 1 en su cantidad de elementos. Resta ver que al sacar las demas posibilidades no se pierden soluciones:

De utilizar el paso 2-1 conjunto al 1-1, la distancia entre pasos efectivos se puede incremetar. Como al utilizar el backtracking 2-1 se puede obtener una transicion entre estos 2 pasos de forma más directa, el agregar el 1-1 solo incrementa el tiempo de la solución,con lo cual puede podarse.

De utilizar el 2-1 con el 2-2, se produce el mismo efecto que en el caso anterior.

De utilizar el 2-1 con el 1-2, se estarìa volviendo a un paso anterior en cantidad de personas de un lado y otro, el cual ya habría sido evaluado en el caso del backtracking 2-1.

1.6 Experimientos y conclusiones

1.6.1 Test

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

Todos los arqueologos y canibales presentan la misma velocidad

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

Para este tipo de testeo mostraremos a continuación un ejemplo del mismo, exponiendo su respectivo resultado. Además veremos más adelante, que por el desarrollo de nuestro algoritmo y sus respectivas podas este será el peor caso en referencia a la perfomance del mismo.

Con un:
Cantidad de arqueologos : 4
Cantidad de canibales : 2
Velocidad de arqueologos : 10 10 10 10
Velocidad de canibales : 10 10
Obtuvimos el siguiente resultado:
Velocidad de cruce total : 90

No hay canibales

Esta versión se da cuando M = 0.
Para este tipo de testeo mostraremos a continuación un ejemplo del mismo, exponiendo su respectivo resultado.

Con un:
Cantidad de arqueologos : 5
Cantidad de canibales : 0
Velocidad de arqueologos : 15 10 5 2 20
Velocidad de canibales :
Obtuvimos el siguiente resultado:
Velocidad de cruce total : 56

Todos los arqueologos y canibales presentan velocidades distintas

Este caso se da cuando $V_i \neq W_j \ \forall$ (i,j) $\leftarrow$ [0..6]
Aquí veremos, un ejemplo del conjunto de test de este tipo, exponiendo su respectivo resultado.

Con un:
Cantidad de arqueologos : 3
Cantidad de canibales : 2
Velocidad de arqueologos : 2 4 6
Velocidad de canibales : 1 3 5
Obtuvimos el siguiente resultado:
Velocidad de cruce total : 18

Hay un canibal cada dos arqueologos

Este tipo de caso se cumple cuando N = 2 * M
Un ejemplo que simboliza el conjunto de test de este tipo es el siguiente:

Con un:

Cantidad de arqueologos : 4
Cantidad de canibales : 2
Velocidad de arqueologos : 3 6 9 12
Velocidad de canibales : 1 2
Obtuvimos el siguiente resultado:
Velocidad de cruce total : 33

Hay mas canibales que arqueologos

Este tipo de caso se cumple cuando N $<$ M
A continuación enunciaremos, un ejemplo del conjunto de test de este tipo, exponiendo su respectivo resultado.

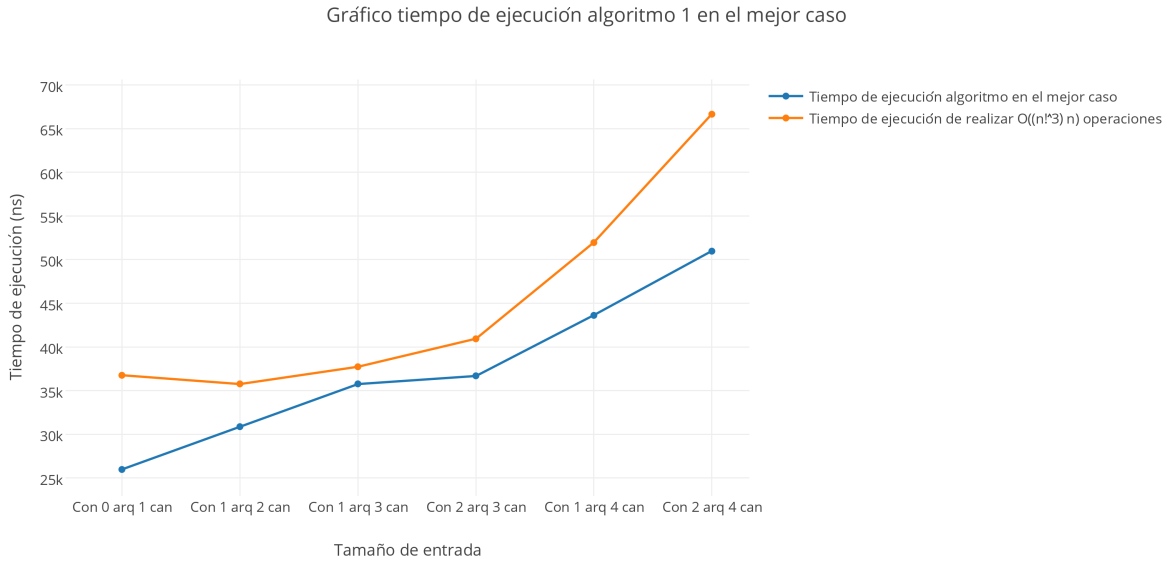
Con un:
Cantidad de arqueologos : 2
Cantidad de canibales : 3
Velocidad de arqueologos : 3 6
Velocidad de canibales : 1 2 5
Obtuvimos el siguiente resultado:
Velocidad de cruce total : *NO HAY SOLUCIÓN*

1.6.2 Performance De Algoritmo y Gráfico

En lo que sigue, mostraremos buenos y malos casos para nuestro algoritmo, y a su vez, daremos el tiempo estimado según la complejidad del algoritmo calculada anteriormente.

Luego de varios experimentos, pudimos llegar a la conclusión que uno de los tipos de casos que resulta más beneficioso para nuestro algoritmo es en el cual **hay más canibales que arqueologos**, esto se da ya que nuestro algoritmo va chequeando todos los posibles pares y como en la primera corrida ya encuentra que no es posible termina.

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



Gráfico

1.1 - *Mejor Caso*

Si a esto lo dividimos por la complejidad propuesta obtenemos:

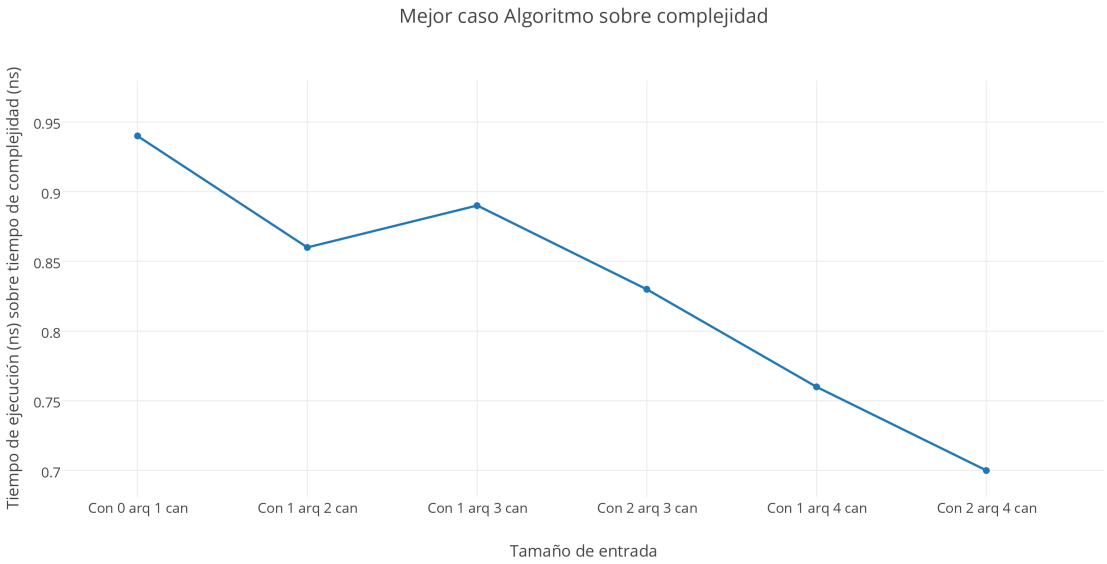


Gráfico 1.2 - *Mejor Caso / Complejidad*

Para realizar esta división realizamos 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 consisos.

Luego de realizar dicha división, el promedio de dichos valores resultante fue el siguiente:

Promedio	0.83

Se puede observar en la figura 1.2 que luego de realizar la división de los tiempos la funcion resultante permanece siempre por debajo de 1 lo cual corrobora para el mejor caso que el mismo se encuentra correctamente acotado por la complejidad calculada anteriormente.

Luego, uno de los peores casos para nuestro algoritmo es en el cual tanto **los canibales como los arqueologos presentan la misma velocidad**, esto se da así ya que nuestro algoritmo chequea todos los pares posibles y como todos pueden ser solución no se podrá efectuar ningún tipo de poda.

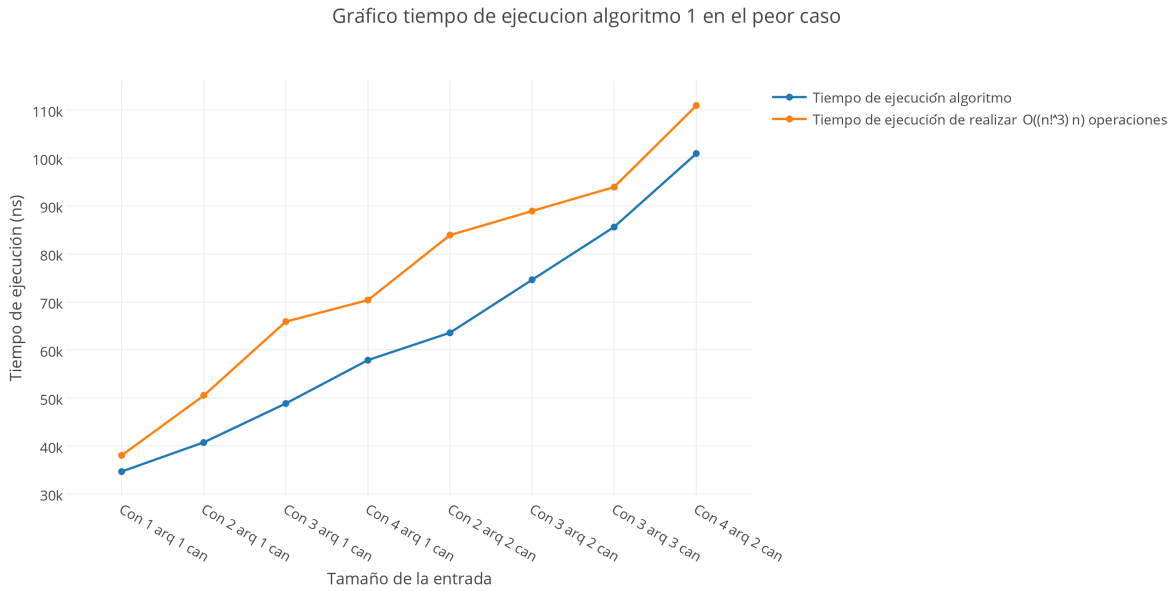


Gráfico 1.3 - *Peor Caso*

Si a esto lo dividimos por la complejidad propuesta obtenemos:

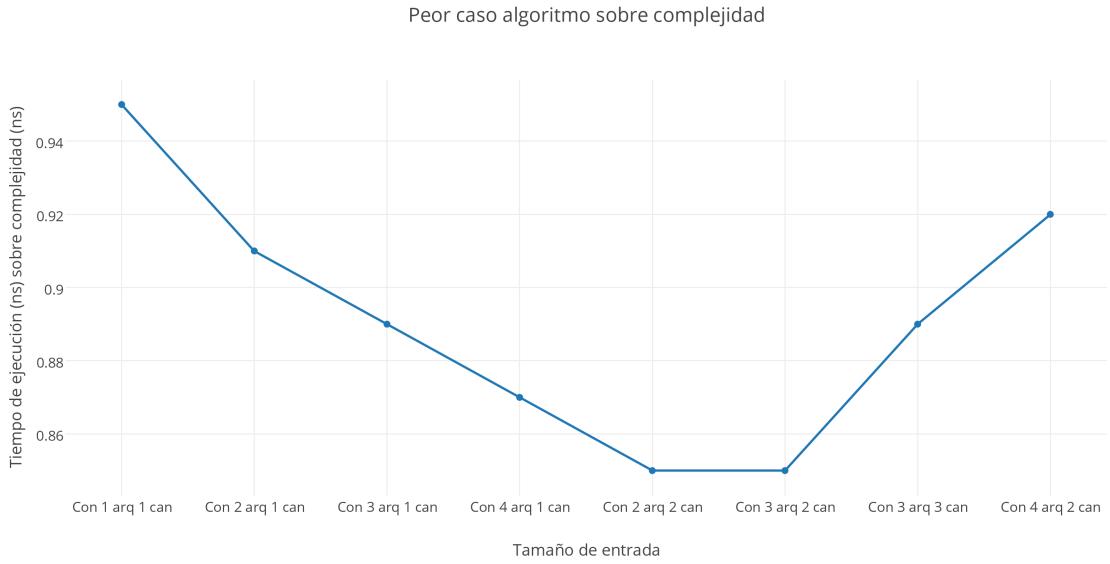


Gráfico 1.4 - *Peor Caso / Complejidad*

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.

Luego de realizar dicha división, el promedio de dichos valores resultante fue el siguiente:

Promedio	0.89

Podemos observar en la figura 1.4 como la función resultante se mantiene por debajo de 1, a pesar de tener casos en los cuales la función llega a un mínimo el cual se encuentra muy por debajo de 1, lo cual se da por la implementación del algoritmo, donde el mismo resulta favorecido al tener igualdad de grupos (canibales y arqueologos).

Luego de lo mostrado, podemos ver que ya sea en el peor caso nuestro algoritmo en función al tiempo de ejecución queda asintotizado por debajo de la función del tiempo de la complejidad .

2 Ejercicio 2

2.1 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 esta vacio.

Para poder quitar la llave y nivelar dicha balanza, tendremos unas pesas donde sus pesos son en potencia 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.

2.2 Explicación de resolución del problema

Para solucionar este problema y poder quitar la llave y dejar equilibrado como se encontraba anteriormente realizamos un algoritmo el cual explicaremos a continuación:

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

Como las pesas, presentan un peso en potencia de 3, creamos una variable denominada *sumaParcial* como la palabra lo indica iremos sumando las potencias desde 3^0 hasta un 3^i, donde dicha suma sea igual al valor de entrada denominado P o en su defecto el inmediato mayor al mismo.

Luego de tener dicha *sumaParcial* guardada crearemos un array que nombramos *sumasParciales* de tamaño $i+1$ el cual iniciaremos vacio. Una vez creado el mismo, llenaremos el array con cada una de las sumas parciales desde el valor que finalizo i hasta 0 de la forma que nos quede *sumasParciales[i]* = *sumaParcial* donde *sumaParcial* será *sumaParcial*-3^{i-1}.

Finalizado esto, realizaremos una busqueda binaria para llevar el valor de p a 0. Para un trabajo más sencillo guardamos el valor inicial de P en la variable *equilibrioActual* a la cual le iremos restando y/o sumando el valor de nuestras pesas.

Dicha busqueda binaria la realizaremos en un ciclo que irá desde el valor en módulo de *equilibrioActual* e iteraremos el mismo hasta que sea 0. Luego, como en toda busqueda binaria, trabajaremos con nuestro array *sumasParciales* chequeando si en la mitad del array nuestra *sumaParcial* es mayor o igual al valor en módulo de *equilibrioActual*. En caso de que fuese verdadero, chequeamos si el valor de *equilibrioActual* es mayor o menor a 0. Si es menor a 0, sumaremos nuestra pesa correspondiente al índice en el que estamos de nuestro array *sumasParciales*, al valor de *equilibrioActual* y guardaremos nuestra pesa en el *arrayD* que simboliza al plato derecho de la balanza. Si es mayor a 0, en vez de sumarla la restamos y la guardamos en el array *arrayI* que simboliza el otro plato.

Siguiendo el razonamiento de la busqueda binaria, volveremos a partir nuestro array en dos y haremos el mismo chequeo.

En caso de que el valor en módulo de *equilibrioActual* sea mayor que la mitad de *sumasParciales* nos quedaremos con la mitad más grande del arreglo e iteraremos nuevamente.

Una vez que llegamos a 0 y por consiguiente salimos de dicho ciclo, tendremos nuestras pesas ordenadas de mayor a menor en $arrayD$ y en $arrayI$, bastará con invertir los arreglos para que queden de menor a mayor y devolver los mismos, finalizando así nuestro algoritmo.

2.3 Algoritmos

Algoritmo 2 BALANZA

```
1: function ALGORITMO(in LongLong : P)→ out S : Long Long out T : Long Long out arrayI :
   List<Long Long> out arrayD : List<Long Long>
2:     creo variable long long equilibrioActual = P                                  //O(1)
3:     creo variable long long i = 0                                                 //O(1)
4:     creo variable long long sumaParcial = 0                                       //O(1)
5:     while sumaParcial < P do                                                      //O(√P)
6:         sumaParcial ← sumaParcial + 3^i                                           //O(1)
7:         i++                                                                       //O(1)
8:     end while
9:     creo long long size = i+1                                                     //O(1)
10:    creo long long sumasParciales[size]                                           //O(√P)
11:    while i ≥ 0 do                                                                //O(√P)
12:        sumasParciales[i] ← sumaParcial                                           //O(1)
13:        sumaParcial ← sumaParcial - 3^(i−1)                                       //O(1)
14:        i–                                                                        //O(1)
15:    end while
16:    creo long long middle = size/2                                                //O(1)
17:    while |equilibrioActual| > 0 do                                               //O(lg(√P))
18:        if sumasParciales[middle] ≥ |equilibrioActual|
19:            ∧ sumasParciales[middle-1] < |equilibrioActual| then                  //O(1)
20:            creo long long potencia = sumasParciales[middle]-sumasParciales[middle-1]  //O(1)
21:            if equilibrioActual < 0 then                                          //O(1)
22:                equilibrioActual ← potencia + equilibrioActual                    //O(1)
23:                arrayD ∪ potencia                                                 //O(1)
24:            else
25:                equilibrioActual ← equilibrioActual -potencia                     //O(1)
26:                arrayI ∪ potencia                                                 //O(1)
27:            end if
28:            size ← middle                                                         //O(1)
29:            middle ← middle/2                                                     //O(1)
30:        end if
31:        if sumasParciales[middle] < |equilibrioActual| then                       //O(1)
32:            middle ← middle + size/2                                              //O(1)
33:        end if
34:        if sumasParciales[middle-1] ≥ |equilibrioActual| then                     //O(1)
35:            size ← middle                                                         //O(1)
36:            middle ← middle + size/2                                              //O(1)
37:        end if
38:    end while
39:    devolver(armadoBalanza)                                                       //O(√P)
40: end function
```

Complejidad: $O(\sqrt{P})$

Algoritmo 3 armadoBalanza

```
1: function ARMADOBALANZA( : )→ out S: Integer out T: Integer out arrayI: List<Integer>
   out arrayD: List<Integer>
2:     invertir(arrayD)                                   //O(√P)
3:     invertir(arrayI)                                   //O(√P)
4:     devolver arrayD.tamaño                             //O(1)
5:     devolver arrayI.tamaño                             //O(1)
6:     devolver(arrayD)                                   //O(√P)
7:     devolver(arrayI)                                   //O(√P)
8: end function
```

Complejidad: $O(\sqrt{P})$

2.4 Análisis de complejidades

Nuestro algoritmo como mencionamos anteriormente presenta 3 ciclos predominantes de los cuales uno corresponde a la busqueda binaria.
El primero de ellos consta en recorrer desde 3^0 hasta 3^i donde la suma de estos sea igual a P o en su defecto el inmediato mayor. Por lo tanto, como la suma se realiza en O(1), mostraremos que recorrer hasta un i donde la suma de dichos valores sea igual o inmediatamente mayor a P es menor o igual a $\sqrt{P}$.

Si $i = 0 \Rightarrow$ terminamos.
Luego sea $3^i \geq P \geq 3^{i-1}$ con $i > 0$. Queremos ver que $i \leq \sqrt{P}$:
Sabemos que $P \geq 3^{i-1} \Rightarrow \sqrt{P} \geq \sqrt{3^{i-1}}$
Veamos que $\sqrt{3^{i-1}} \geq i \Rightarrow 3^{i-1} \geq i^2$. Para $i = 1$ tenemos que $3^{1-1} \geq 1$ siempre. Luego, para $i > 1$ como 3^{i-1} es creciente y mayor o igual que i^2 se cumple siempre esta desigualdad. Por lo tanto queda probado que recorrer hasta un i tal que

$$\sum_{x=0}^{i} 3^i \geq P$$

se encuentra en el orden de $O(\sqrt{P})$.

Luego, creamos un long long *size* inicializado en i+1 y un array *sumasParciales* de tamaño size inicializado vacio, por lo tanto, la creación de la variable size y el array vacio insumirán O(1) y $O(\sqrt{P})$.

Siguiendo el desarrollo del algoritmo, pasamos a nuestro segundo ciclo, en el cual llenaremos el array *sumasParciales*, como vimos anteriormente iterar desde el valor i hasta 0 es $O(\sqrt{P})$, y dentro de dicho ciclo lo único que hacemos es ir guardando en la posición i-esima del array el valor de sumaParcial - 3^{i-1} y a sumaParcial le guardamos el valor de sumaParcial - 3^{i-1}. Como estas dos operaciones se realizan en O(1), nuestro segundo ciclo terminará insumiendo $O(\sqrt{P})$.

Luego, nuestro tercer y último ciclo, corresponde a la búsqueda binaria, la cual se realizá en $O(\lg(\sqrt{P}))$ como en toda busqueda binaria, trabajaremos con nuestro array *sumasParciales* chequeando si en la mitad del array nuestra sumaParcial es mayor o igual al valor en módulo de *equilibrioActual*. En caso de que fuese verdadero, chequeamos si el valor de *equilibrioActual* es mayor o menor a 0. Si es menor a 0, sumaremos nuestra pesa correspondiente al índice en el que estamos de nuestro array *sumasParciales*, al valor de *equilibrioActual* y guardaremos nuestra pesa en el *arrayD* que simboliza al plato derecho de la balanza. Si es mayor a 0, en vez de sumarla la restamos y la guardamos en el array *arrayI* que simboliza el otro plato. Siguiendo el razonamiento de la busqueda binaria, volveremos a partir nuestro array en dos y haremos el mismo chequeo.

En caso de que el valor en módulo de *equilibrioActual* sea mayor que la mitad de *sumasParciales* nos quedaremos con la mitad más grande del arreglo e iteraremos nuevamente.
Una vez llegado a 0 el valor de *equilibrioActual* saldremos del ciclo. Como describimos dentro del ciclo realizaremos sumas, restas y chequeos los cuales se realizaran todos en O(1), por lo tanto. Nuestro tercer ciclo insumirá $O(\lg(\sqrt{P}))$.

Fuera de este último ciclo, tendremos nuestras pesas ordenadas de mayor a menor en $arrayD$ y en $arrayI$, bastará con invertir los arreglos para que queden de menor a mayor y devolver los mismos, finalizando así nuestro algoritmo. Dicho invertir costará O(#elementosArrayD) y O(#elementosArrayI), que como demostramos anteriormente en el caso de que todas las pesas fueran a parar a un único plato y naturalmente a un único array $O(\#elementos) \leq O(\sqrt{P})$.

Por lo tanto, nuestro algoritmo realizará en su defecto 3 ciclos (como vimos, se puede dar el caso de invertir el array y que estén todas las pesas en un único array) $O(\sqrt{P})$ y el ciclo de la busqueda binaria $O(\lg(\sqrt{P}))$, nos queda que la complejidad total de nuestro algoritmo es $O(\sqrt{P})$.

Complejidad total: $O(\sqrt{P})$ [O(1) + O(1)] + O(1) + $O(\sqrt{P})$ + $O(\sqrt{P})$ [O(1) + O(1)] + $O(\lg(\sqrt{P}))$[O(1) + O(1) + O(1) + O(1) + O(1) + O(1)] = $\mathbf{O(\sqrt{P})}$

2.5 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, la segunda, en donde realizamos un ciclo que va disminuyendo el valor *equilibrioActual* hasta llegar a 0.

Este algoritmo utiliza una propiedad particular que es la de poder generar todos los números entre 0 y $\sum_{i=0}^{n}(3^i)$ con potencias de 3 diferentes (El 0 se incluye por definición). Pero para que esto sea válido debemos demostrarlo.

Veamos para empezar que podemos generar todos los números entre $[0, 1]$ que son 0 y 1.

Este es un caso bastante trivial, por lo tanto veamos que sucede en el intervalo

$$[0, \sum_{i=0}^{1}(3^i)] = [0, 4] \tag{1}$$

- 4 = 3+1
- 2 = 3-1

Parece pues que para el caso base, es decir el primer intervalo, podemos generarlos todos con potencias de 3 diferentes.

Asumamos pues que esto vale para $i = n \in \mathbb{N}$ y demostremos que vale para $n + 1$

Es decir, puedo generar todos los números en el intervalo

$$[0, \sum_{i=0}^{n}(3^i)] \tag{2}$$

Veamos que podemos lograr lo mismo para

$$[0, \sum_{i=0}^{n+1}(3^i)] \tag{3}$$

Pues veamos que

$$[0, \sum_{i=0}^{n+1}(3^i)] = [0, \sum_{i=0}^{n}(3^i)] + 3^{n+1} \tag{4}$$

Con lo cual ya puede verse que los numeros entre 0 y $\sum_{i=0}^{n}(3^i)$ podemos generarlos por hipótesis inductiva (2.5).

Luego notemos que

$$3^{n+1} = 3 * 3^n. \tag{5}$$

Como $3^n < \sum_{i=0}^{n}(3^i)$, 3^n está dentro del intervalo de la hipótesis inductiva (2.5) así que tambien podemos formarlo con potencias de 3, y en particular cualquier número menor a 3^n usando la misma hipótesis.

Luego podemos generar cualquier $x \in \mathbb{N}$, $x \leq \sum_{i=0}^{n+1}(3^i)$.

Por lo tanto queda probado que la hipótesis inductiva vale $\forall n \in \mathbb{N}$

Además notemos que el número más cercano a x será la *maxima potencia de* 3 *en* $[0, \sum_{i=0}^{n+1}(3^i)]$ es decir 3^{n+1}.
Pues este es el intervalo que genera a x y sabemos que $\sum_{i=0}^{n}(3^i) < 3^{n+1}$ y que $\sum_{i=0}^{n}(3^i) < x$.

Y como al restar x por 3^{n+1} obtenemos un número más pequeño que $\sum_{i=0}^{n}(3^i)$ en módulo, pues (el caso negativo es simétrico)

$$x < \sum_{i=0}^{n+1}(3^i) = x < \sum_{i=0}^{n}(3^i) + 3^{n+1} = x - 3^{n+1} < \sum_{i=0}^{n}(3^i) \tag{6}$$

Entonces nunca se repetiran potencias de 3 en el proceso.
Con esto se concluye que se pueden generar todos los números mediante potencias de 3 únicas.

2.6 Experimientos y conclusiones

2.6.1 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:

El valor de entrada P es de la forma 3^i para un i←[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.

El valor de entrada P es de la forma

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

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

El valor de entrada P es de la forma $3^i + R$ para (i, R) $\leftarrow$[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 mayor y de esta forma se itera mas de una vez el segundo y tercer ciclo.

El valor de entrada P es impar

Este caso se cumple cuando se recibe un P el cual el mismo es de la forma $PMOD2 = 1$.

El valor de entrada P es par

Este caso se cumple cuando se recibe un P el cual el mismo es de la forma $PMOD2 = 0$.

2.6.2 Performance De Algoritmo y Gráfico

Acorde a lo solicitado, mostraremos los mejores y peores casos para nuestro algoritmo, y además, daremos el tiempo estimado según la complejidad del algoritmo calculada anteriormente.

Luego de chequear varios instancias, pudimos llegar a la conclusión que uno de los tipos de casos que resulta más beneficioso para nuestro algoritmo es en el 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 observacion desarrollamos el siguiente grafico con las instancias:

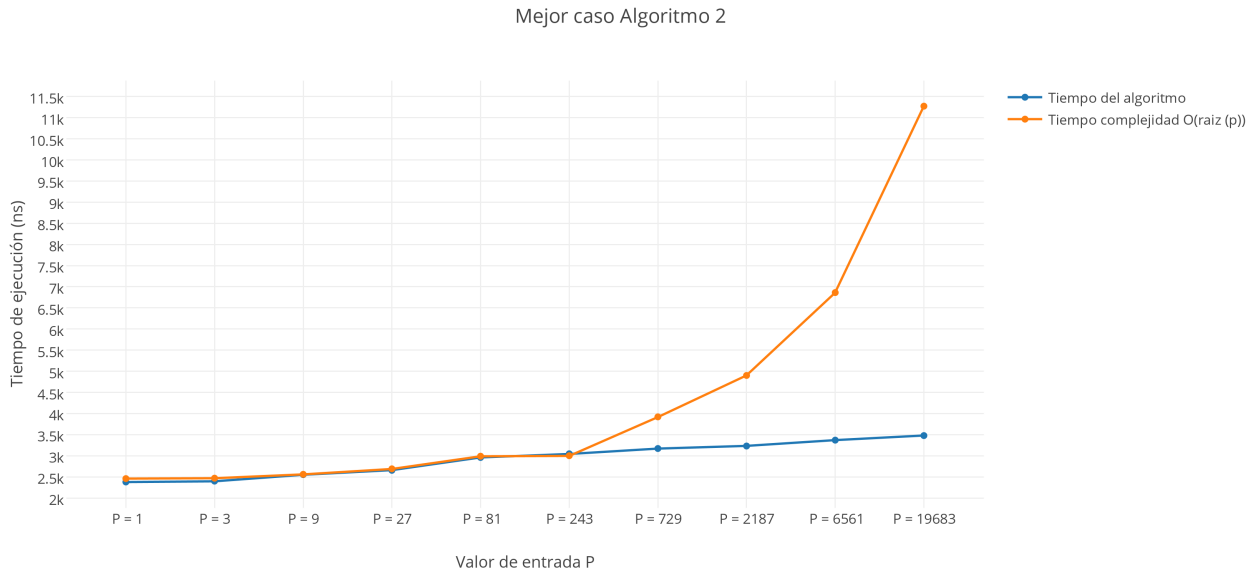


Gráfico 2.1 - MejorCaso

Y dividiendo por la complejidad de nuestro algoritmo llegamos a:

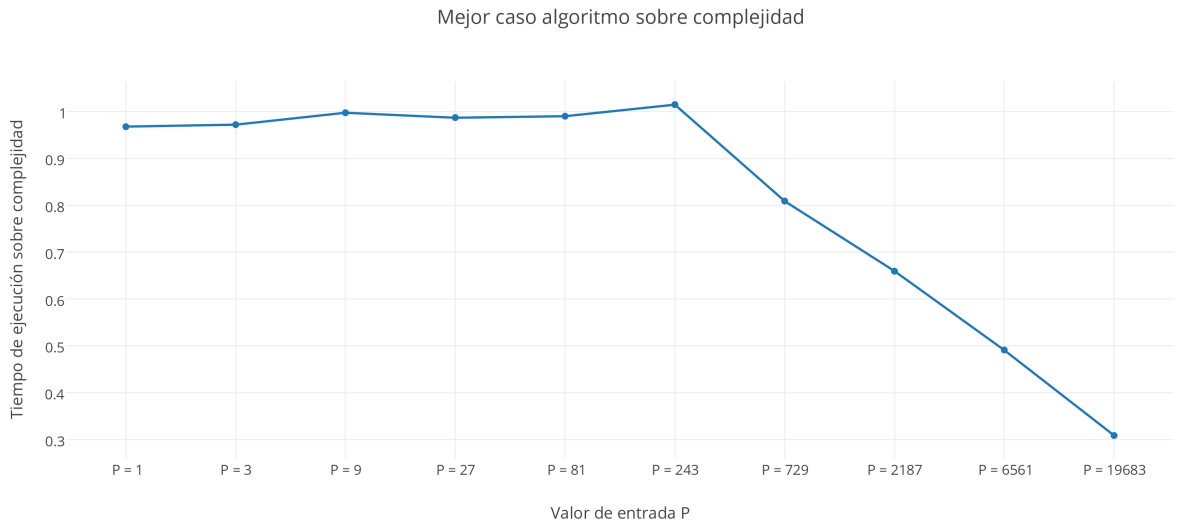


Gráfico 2.2 - MejorCaso/Complejidad

Como se puede ver, en el gráfico 2.1 cuando el valor de entrada P crece el tiempo de la función de la complejidad tiende a crecer muy rápido por lo cual mostraremos estas instancias en los gráficos y más adelante mostraremos una tabla con los valores restantes.

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, como luego de realizar la división por la complejidad cuando el n aumenta el valor tiende a 0.

A continuación mostraremos una tabla con los 20 datos de medición mas relevantes y mostraremos un promedio de la totalidad de las instancias probadas.

Tamaño(n)	Tiempo(t)	$\sqrt{P}$)	$t/\sqrt{P}$)
3^{10}	3730,82	18626	0,200
3^{11}	4428,36	31370	0,141
3^{12}	4685,66	86265	0,054
3^{13}	4999,42	124006	0,040
3^{14}	5460,14	188705	0,028
3^{15}	5756,1	356824	0,016
3^{16}	5891,5	265657	0,022
3^{17}	6026,9	400446	0,015
3^{18}	6162,3	1617468	0,003
3^{19}	6297,7	2542364	0,002
3^{20}	6433,1	2608534	0,002
3^{21}	6568,5	3956914	0,001
3^{22}	6703,9	7124699	9 x 10^{-4}
3^{23}	6839,3	11467843	5 x 10^{-4}
3^{24}	6974,7	19803192	3 x 10^{-4}
3^{25}	7110,1	35212754	2 x 10^{-4}
3^{26}	7245,5	59285080	1 x 10^{-4}
3^{27}	7380,9	103014535	7,1 x 10^{-5}
3^{28}	7516,3	178188535	4,2 x 10^{-5}
3^{29}	7651,7	308181445	2,4 x 10^{-5}
3^{30}	7787,1	438174355	1,7 x 10^{-5}
Promedio			0,026

Promedio total conseguido: 0,026

Verificando el peor caso, llegamos a la conclusión que el tipo de caso en el que resulta menos beneficioso trabajar con nuestro algoritmo 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 una tabla comparativa con los valores más relevantes y además dos gráficos los cuales mostraremos a continuación:

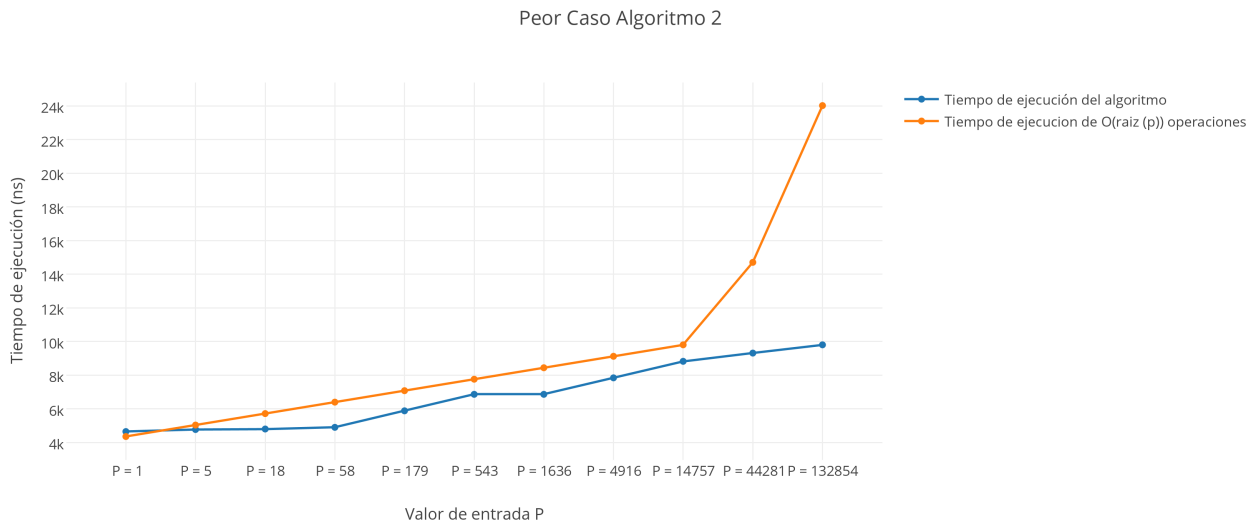


Gráfico 2.3 - *PeorCaso*

Dividiendo por la complejidad propuesta llegamos a:

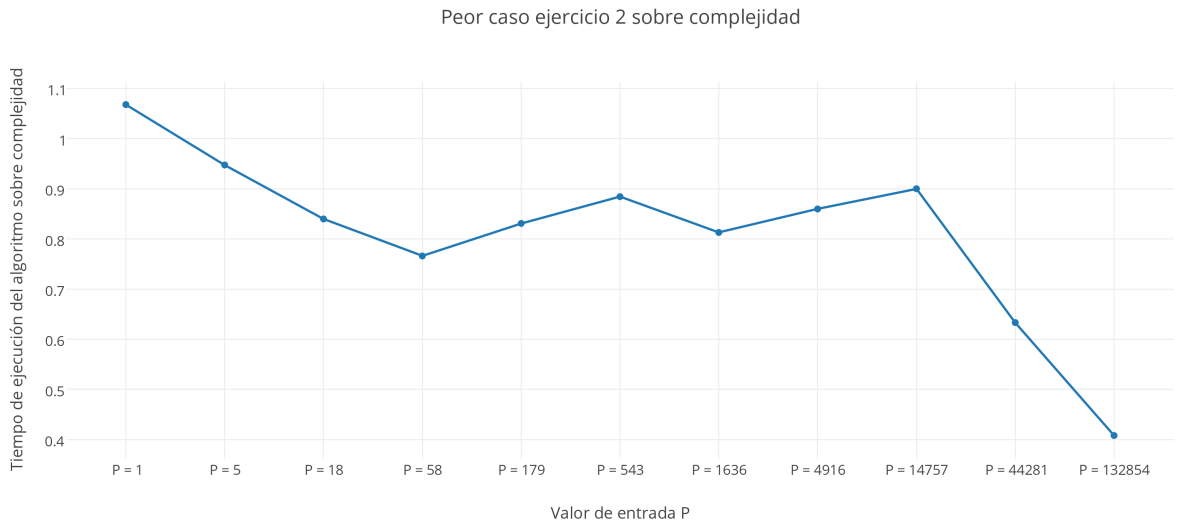


Gráfico 2.4 - *PeorCaso/Complejidad*

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.

Se puede observar que a pesar de tardar varios nanosegundos este tipo de caso, al dividir por la complejidad complejidad teorica la función resultante tiende a 0 quedando comparativamente por encima del mejor caso.

A continuación mostraremos una tabla de valores de las ultimas 20 instancias y mostraremos el promedio total conseguido .

Tamaño(n)	Tiempo(t)	$\sqrt{P}$)	$t/\sqrt{P}$)
1	4650	4355	1.067
5	4771	5036	0.947
18	4801	5717	0.839
58	4901	6398	0.766
179	5882	7079	0.830
543	6862	7760	0.884
1636	6862	8441	0.812
4916	7842	9122	0.859
14757	8823	9803	0.900
44281	9312	14704	0.633
132854	9803	24017	0.408
398574	18625	39702	0.469
1195735	18625	67150	0.277
3587219	19115	57837	0.330
10761672	26468	132339	0.200
32285032	26468	203410	0.130
96855113	31860	307319	0.103
290565357	32839	548960	0.059
581130733	39211	910195	0.043
1743392200	41535	1567476	0.026
5230176601	53915	2659024	0.020
Promedio			0,530

Promedio total conseguido: 0,530

Se puede observar como el peor caso presenta un promedio mayor que el mejor caso, concluyendo lo que enunciamos inicialmente.

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.

3 Ejercicio 3

3.1 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.

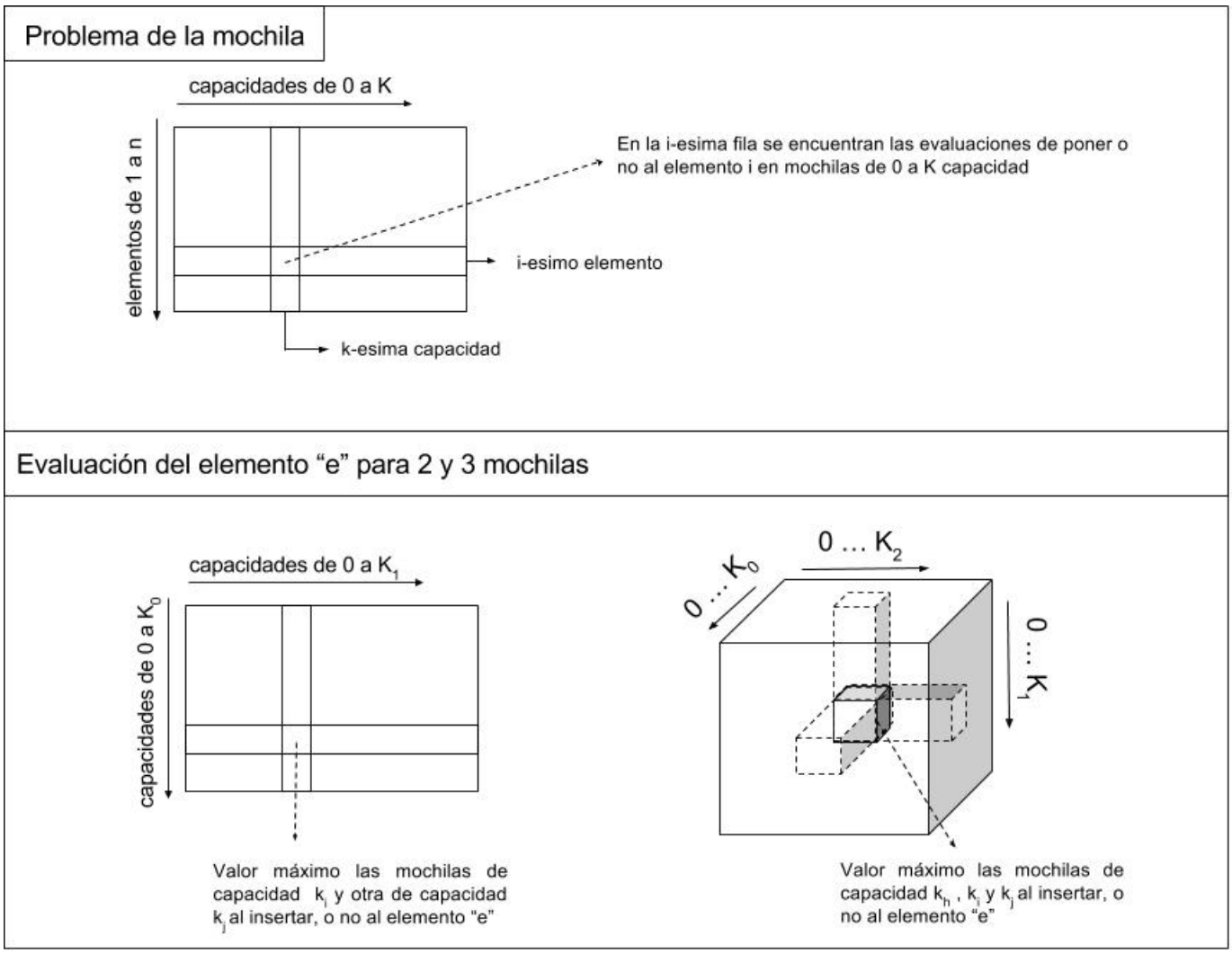
3.2 Explicación de resolución del problema

Obteniendo el valor máximo

El problema de la mochila analiza, para cada objeto, el máximo valor que se puede obtener para cada capacidad posible de la mochila. Si llamamos K a la capacidad de la mochila, se evaluará introducir un elemento e en la capacidad k con $1 \leq k \leq K$.

En cada evaluación se decide entre no meter el elemento, con lo cual el valor máximo de la mochila que tenga la capacidad k será el calculado con algún elemento anterior (o cero en el caso de que sea el primero en ser evaluado); o meter el elemento, con lo cual, al valor del elemento se le suma el valor de la mochila de capacidad k - $peso(e)$ que sea máxima. 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.

Extendiendonos a 2 mochilas, al evaluar la k posibilidad de la primer mochila (con capacidad K_a), se debe tener en cuenta que tambien está la posibilidad de utilizar la segunda mochila (con capacidad K_b). Esto nos muestra que por cada elemento debemos calcular $K_a \times K_b$ combinaciones de capacidades. Sin contar el primer elemento, el resto calculará sus combinaciones en base a las del elemento que le precedió en la evaluación, siguiendo la idea del algoritmo original de una mochila. Para obtener el valor final, se buscará en la matriz resultante de evaluar el último elemento, a la combinación de capacidades entre ambas mochilas que resulte máxima.



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

Recuperando los elementos utilizados

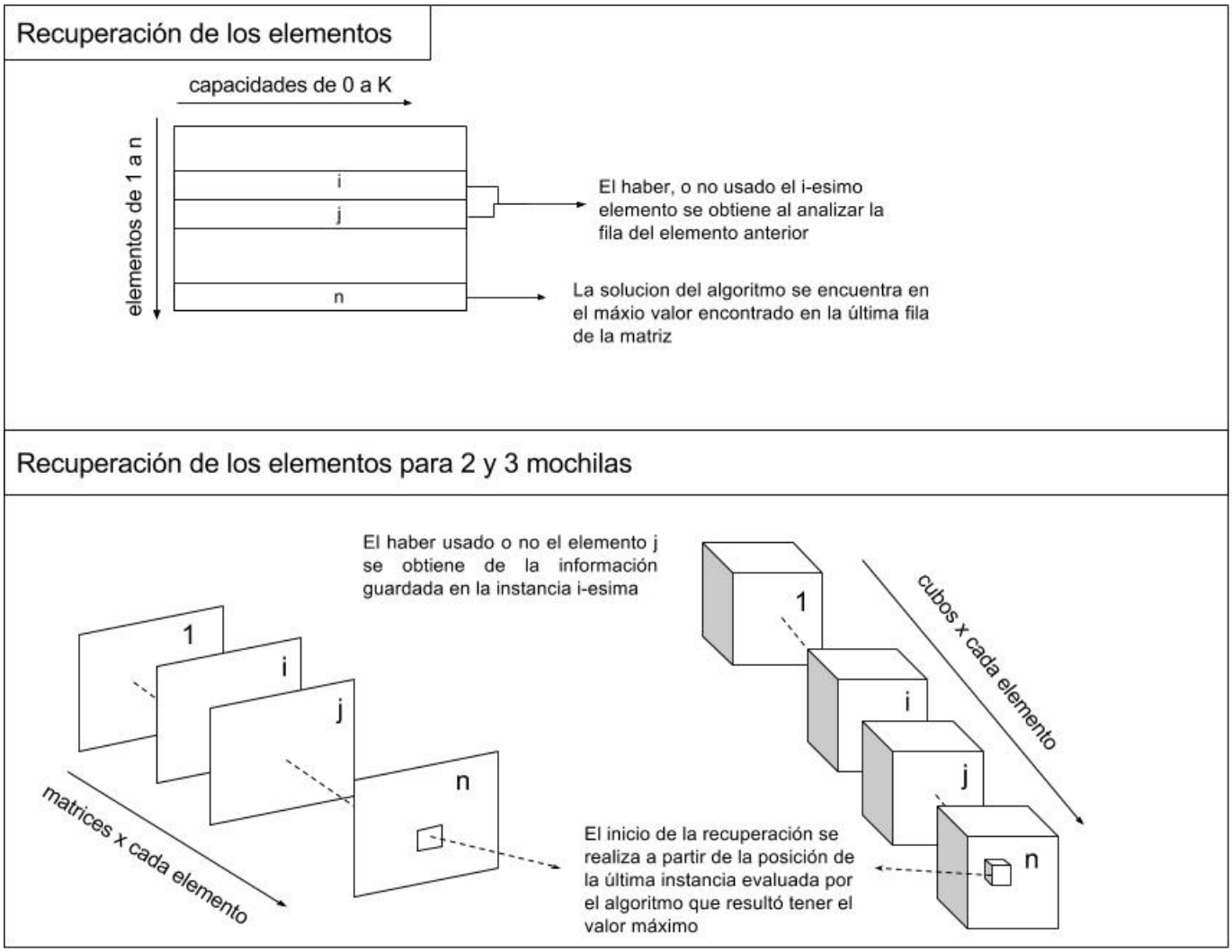
Siendo que la consigna pide los objetos involucrados en la solución, es necesaria una forma de, a partir de los valores máximos obtenidos, deducir los elementos que fueron usados y el lugar donde fueron colocados para lograr el resultado.

Analizando la última matriz del caso de 2 mochilas (sin perder generalización para 3 mochilas), podemos notar que por cada objeto que iteramos para intentar encontrarle un lugar en las mochilas, deja el valor previo, de cada combinacion de capacidades, sin tocar cuando el objeto no es utilizado. Al ser utilizado, se cambia el valor por el nuevo máximo alcanzado. 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 calculo de cada elemento. Llamando a las matrices del elemento i M, y N a la del elemento j subsiguiente a i (ambas de dimensión $K_a \times K_b$), se tiene que si el maximo 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, opto 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 que 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.

3.3 Algoritmos

A continuación se detalla el pseudo-código de la parte principal del algoritmo:

Algoritmo 4 Mochilas

```
1: function EJ3(in Lista < mochila >: MOCHILAS in Lista < objeto >: OBJETOS)→ out S: Integer
   out elementos: List<Integer>
2:     if #(MOCHILAS) == 1 then                                                    //O(1)
3:         maximizoSimple(MOCHILAS[1],OBJETOS)                                    //O(K(M[1]) * N)
4:     end if
5:     if #(MOCHILAS) == 2 then                                                    //O(1)
6:         if capacidad[M_1] == capacidad[M_2] then                                //O(1)
7:             maximizoSimple(MOCHILAS[1], OBJETOS)                               //O(K(M[1]) * N)
8:             maximizoSimple(MOCHILAS[2], OBJETOS)                               //O(K(M[2]) * N)
9:         else
10:            maximizoDoble(MOCHILAS[1],MOCHILAS[2], OBJETOS)                    //O(M * N)
11:        end if
12:    end if
13:    if #(MOCHILAS) == 3 then                                                    //O(1)
14:        if capacidad[M_1] == capacidad[M_2] ∧ capacidad[M_2] == capacidad[M_3] then  //O(1)
15:            maximizoSimple(MOCHILAS[1], OBJETOS)                               //O(K(M[1]) * N)
16:            maximizoSimple(MOCHILAS[2], OBJETOS)                               //O(K(M[2]) * N)
17:            maximizoSimple(MOCHILAS[3], OBJETOS)                               //O(K(M[3]) * N)
18:        else
19:            maximizoTriple(MOCHILAS,OBJETOS)                                   //O(K(M) * N)
20:        end if
21:    end if
22:    devolver arrayMaximo.size                                                   //O(N)
23:    devolver arrayMaximo                                                        //O(N)
24: end function
```

Complejidad: $O(K(M) * N)$

Algoritmo 5 Mochilas

```
1: function MaximizoSimple(in Lista < mochila >: MOCHILAS in Lista < objeto >: OBJETOS)→
   out max: Integer out arrayUsados: List<Integer>
2:     while i < #(OBJETOS) do                                                    //O(N)
3:         while j < capacidad(MOCHILAS[1]) do                                    //O(K(M[1]))
4:             if j == 0 then                                                     //O(1)
5:                 matriz[OBJETOS[i]][j] ← 0                                      //O(1)
6:             end if
7:             if j > 0 ∧ peso(OBJETOS[i]) ≤ j ∧ ¬(Usado(OBJETOS[i])) then    //O(1)
8:                 if i == 0 then                                                 //O(1)
9:                     matriz[OBJETOS[i]][j] ← OBJETOS[i]                         //O(1)
10:                else
11:                    matriz[elemento][j] ← MAX(valor(OBJETOS[i]) +
12:                    matriz[i-1][j - peso(OBJETOS[i])], matriz[i-1][j])         //O(1)
13:                end if
14:            else
15:                if i == 0 then                                                 //O(1)
16:                    matriz[OBJETOS[i]][j] ← 0                                  //O(1)
17:                else
18:                    matriz[elemento][j] ← matriz[i-1][j]                       //O(1)
19:                end if
20:            end if
21:            j++                                                                //O(1)
22:        end while
23:        i++                                                                    //O(1)
24:    end while
25:    columnaActual ← K(M[1])                                                    //O(1)
26:    filaActual ← N-1                                                           //O(1)
27:    while filaActual ≥ 0 ∧ columnaActual ≥ 0 do
28:        if ¬(vacia(columnaActual)) ∧ filaActual == 0 then                      //O(1)
29:            if matriz[filaActual][columnaActual] then                          //O(1)
30:                arrayUsados ∪ matriz[filaActual][columnaActual]                //O(1)
31:            end if
32:            filaActual–                                                        //O(1)
33:        end if
34:        if matriz[filaActual-1][columnaActual] ≠ matriz[filaActual][columnaActual] then //O(1)
35:            arrayUsados ∪ matriz[filaActual][columnaActual]                    //O(1)
36:            columnaActual ← columnaActual - peso(filaActual)                   //O(1)
37:            filaActual–                                                        //O(1)
38:        else
39:            filaActual–                                                        //O(1)
40:        end if
41:    end while
42:    max ← matriz[filaActual-1][columnaActual]                                  //O(1)
43:    devolver max                                                               //O(1)
44:    devolver arrayUsados                                                       //O(N)
45: end function
```

Complejidad: $O(K(M[1]) * N))$

Algoritmo 6 Mochilas

```
1: function MaximizoDoble(in Lista < mochila >: MOCHILAS in Lista < objeto >: OBJETOS)→
   out max: Integer out arrayMaximo: List<Integer>
2:     cap1 ← capacidad(M[1])                                              //O(1)
3:     cap2 ← capacidad(M[2])                                              //O(1)
4:     creo entero h ←                                                     //O(1)
5:     while h < #(OBJETOS) do                                             //O(N)
6:         peso ← peso(OBJETOS[h])                                         //O(1)
7:         valor ← valor(OBJETOS[h])                                       //O(1)
8:         i ← cap2                                                        //O(1)
9:         while i ≤ 0 do                                                  //O(K(M[1]))
10:            j ← cap1                                                    //O(1)
11:            while j ≤ 0 do                                              //O(K(M[2]))
12:                creo entero m0 ← matriz[j][i]                           //O(1)
13:                creo entero m1 y m2 inicializados en 0                  //O(1)
14:                if j ≥ peso then                                        //O(1)
15:                    m1 ← matriz[j-peso][i]                              //O(1)
16:                end if
17:                if i ≥ peso then                                        //O(1)
18:                    m2 ← matriz[j][i-peso]                              //O(1)
19:                end if
20:                if j ≥ peso ∧ m1 + valor ≥ m0 ∧ m1 + valor ≥ m2 + valor then   //O(1)
21:                    matriz[j][i] ← m1 + valor                           //O(1)
22:                else
23:                    if i ≥ peso ∧ m2 + valor ≥ m0 ∧ m2 + valor ≥ m1 + valor then   //O(1)
24:                        matriz[j][i] ← m2 + valor                       //O(1)
25:                    end if
26:                    if matriz[j][i] > max then                          //O(1)
27:                        max ← matriz[j][i]                              //O(1)
28:                        jMax ← j                                        //O(1)
29:                        iMax ← i                                        //O(1)
30:                        max ← h                                         //O(1)
31:                    end if
32:                end if
33:                i++                                                     //O(1)
34:            end while
35:            j++                                                         //O(1)
36:        end while
37:        h++                                                             //O(1)
38:    end while
39:    i ← max                                                             //O(1)
40:    devolver organizarYDevolverDoble(MOCHILAS, OBJETOS, max,iMax,jMax)  //O(N)
41: end function
```

Complejidad: $O([K(M[1]) + K(M[2])] * N))$

Algoritmo 7 Mochilas

```
1: function organizarYDevolverDoble(in Lista < mochila >: MOCHILAS in Lista < objeto >: OBJETOSin Integer : maxin Integer : iMaxin Integer : jMax)→ out max : Integer out arrayMaximo: List<Integer>
2:     while i > 0 do                                                          //O(i)
3:         peso ← peso(OBJETOS[i])                                             //O(1)
4:         valor ← valor(OBJETOS[i])                                           //O(1)
5:         creo matriz matriz1 ← matriz[j][i]                                  //O(N)
6:         creo matriz matriz2 ← matriz[j][i-1]                                //O(N)
7:         if matriz1[jMax][iMax] ≠ matriz2[jMax][iMax] then                   //O(1)
8:             if jMax ≥ peso ∧
9:                 [iMax] matriz2[jMax - peso][iMax] + valor ==
10:                matriz1[jMax][iMax] then                                    //O(1)
11:                jMax ← jMax - peso                                          //O(1)
12:            else
13:                iMax ← iMax - peso                                          //O(1)
14:            end if
15:        end if
16:        i–                                                                  //O(1)
17:    end while
18:    if i == 0 ∧ max then                                                    //O(1)
19:        peso ← peso(OBJETOS[0])                                             //O(1)
20:        valor ← valor(OBJETOS[0])                                           //O(1)
21:        if jMax ≥ peso then                                                 //O(1)
22:            arrayMochila1 ∪ objeto(peso, valor)                             //O(1)
23:        else
24:            arrayMochila2 ∪ objeto(peso, valor)                             //O(1)
25:        end if
26:    end if
27:    arrayMaximo ∪ arrayMochila1                                             //O(N)
28:    arrayMaximo ∪ arrayMochila2                                             //O(N)
29:    devolver arrayMaximo                                                    //O(N)
30:    devolver max                                                            //O(1)
31: end function
```

Complejidad: O(N)

Algoritmo 8 Mochilas

```
1: function MaximizoTriple(in Lista < mochila >: MOCHILAS in Lista < objeto >: OBJETOS)→
   out max: Integer out arrayMaximo: List<Integer>
2:     creo enteros cap1, cap2 y cap3 inicializados con las capacidades de cada mochila  //O(1)
3:     while h < #(OBJETOS) do                                                           //O(N)
4:         peso ← peso(OBJETOS[h])                                                       //O(1)
5:         valor ← valor(OBJETOS[h])                                                     //O(1)
6:         x ← cap3                                                                      //O(1)
7:         while x ≤ 0 do                                                    //O(K(M[3]))
8:             i ← cap2                                                              //O(1)
9:             while i ≤ 0 do                                                //O(K(M[2]))
10:                j ← cap1                                                          //O(1)
11:                while j ≤ 0 do                                            //O(K(M[1]))
12:                    creo entero m0 ← matriz[j][i][x]                              //O(1)
13:                    creo entero m1, m2 y m3 inicializados en 0                    //O(1)
14:                    if j ≥ peso then                                              //O(1)
15:                        m1 ← matriz[j-peso][i][x]                                 //O(1)
16:                    end if
17:                    if i ≥ peso then                                              //O(1)
18:                        m2 ← matriz[j][i-peso][x]                                 //O(1)
19:                    end if
20:                    if x ≥ peso then                                              //O(1)
21:                        m3 ← matriz[j][i][x-peso]                                 //O(1)
22:                    end if
23:                    if j ≥ peso ∧ m1 + valor ≥ m0 ∧
24:                        m1 + valor ≥ m2 + valor ∧ m1 + valor ≥ m3 + valor then    //O(1)
25:                        matriz[j][i][x] ← m1 + valor                              //O(1)
26:                    end if
27:                    if i ≥ peso ∧ m2 + valor ≥ m0 ∧
28:                        m2 + valor ≥ m1 + valor ∧ m2 + valor ≥ m3 + valor then    //O(1)
29:                        matriz[j][i][x] ← m2 + valor                              //O(1)
30:                    end if
31:                    if x ≥ peso ∧ m3 + valor ≥ m0 ∧
32:                        m3 + valor ≥ m1 + valor ∧ m3 + valor ≥ m2 + valor then    //O(1)
33:                        matriz[j][i][x] ← m3 + valor                              //O(1)
34:                    end if
35:                    if matriz[j][i][x] > max then                                 //O(1)
36:                        max ← matriz[j][i][x]                                     //O(1)
37:                        creo enteros jMax, iMax y xMax inicializados con j i x    //O(1)
38:                        max ← h                                                   //O(1)
39:                    end if
40:                    x++                                                           //O(1)
41:                end while
42:                i++                                                               //O(1)
43:            end while
44:            j++                                                                   //O(1)
45:        end while
46:        h++                                                                       //O(1)
47:    end while
48:    i ← max                                                                       //O(1)
49:    devolver organizarYDevolverTriple(MOCHILAS, OBJETOS, max,iMax,jMax, xMax)
   //O(N)
50: end function
```


Complejidad: $O([K(M[1]) + K(M[2])] * N))$

Algoritmo 9 Mochilas

```
1: function organizarYDevolverTriple(in Lista < mochila >: MOCHILAS in Lista < objeto >: OBJETOSin Integer : maxin Integer : iMaxin Integer : jMaxin Integer : xMax)→ out max: Integer out arrayMaximo: List<Integer>
2:     while i > 0 do                                                                    //O(i)
3:         peso ← peso(OBJETOS[i])                                                       //O(1)
4:         valor ← valor(OBJETOS[i])                                                     //O(1)
5:         creo matriz matriz1 ← matriz[j][i][x]                                         //O(N)
6:         creo matriz matriz2 ← matriz[j][i-1][x]                                       //O(N)
7:         if matriz1[jMax][iMax][xMax] ≠ matriz2[jMax][iMax][xMax] then                 //O(1)
8:             if jMax ≥ peso ∧
9:                 matriz2[jMax - peso][iMax][xMax] + valor ==
10:                matriz1[jMax][iMax][xMax] then                                        //O(1)
11:                jMax ← jMax - peso                                                    //O(1)
12:            end if
13:            if iMax ≥ peso ∧
14:                matriz2[jMax][iMax-peso][xMax] + valor ==
15:                matriz1[jMax][iMax][xMax] then                                        //O(1)
16:                iMax ← iMax - peso                                                    //O(1)
17:            end if
18:            if xMax ≥ peso ∧
19:                matriz2[jMax][iMax][xMax-peso] + valor ==
20:                matriz1[jMax][iMax][xMax] then                                        //O(1)
21:                xMax ← xMax - peso                                                    //O(1)
22:            end if
23:        end if
24:        i–                                                                            //O(1)
25:    end while
26:    if i == 0 ∧ max then                                                              //O(1)
27:        peso ← peso(OBJETOS[0])                                                       //O(1)
28:        valor ← valor(OBJETOS[0])                                                     //O(1)
29:        if jMax ≥ peso then                                                           //O(1)
30:            arrayMochila1 ∪ objeto(peso, valor)                                       //O(1)
31:        end if
32:        if iMax ≥ peso then                                                           //O(1)
33:            arrayMochila2 ∪ objeto(peso, valor)                                       //O(1)
34:        end if
35:        if xMax ≥ peso then                                                           //O(1)
36:            arrayMochila3 ∪ objeto(peso, valor)                                       //O(1)
37:        end if
38:    end if
39:    arrayMaximo ∪ arrayMochila1                                                       //O(N)
40:    arrayMaximo ∪ arrayMochila2                                                       //O(N)
41:    arrayMaximo ∪ arrayMochila3                                                       //O(N)
42:    devolver arrayMaximo                                                              //O(N)
43:    devolver max                                                                      //O(1)
44: end function
```

Complejidad: $O(N)$

Aclaraciones: K = función capacidad() M = MOCHILAS N = # OBJETOS

3.4 Análisis de complejidades

El algoritmo por cada uno de los N elementos construye una matriz de $K_1 \times K_2 \times ... \times Km$, y las guarda para poder recuperar al final, los elementos involucrados en la solución.

Para construir estas matrices se recorre cada uno de sus $K_1 \times K_2 \times ... \times K_m$ indices para calcular el valor correspondiente en tiempo constante; habiendo n elementos, la construccion de las matrices se realiza en $O(N * \prod_{i=1}^{m} K_i)$. Con lo cual podemos ver lo siguiente: dado que la suma de la cantidad de cada tipo de elemento es la cantidad de elementos en si:

$$N * \prod_{i=1}^{m} K_i = \sum C_i * \prod_{i=1}^{m} K_i$$

Si tomamos la cota de 3 mochilas dada por la consigna, y acotamos las capacidades de las mochila por la de aquella de mayor capacidad (capacidad K) se tiene:

$$\sum C_i * \prod_{i=1}^{3} K_i \leq N * \prod_{i=1}^{3} K_i = \sum C_i * \prod_{i=1}^{3} K = \sum C_i * K^3$$

Podemos concluir que el algoritmo respeta la cota requerida:

$$\sum C_i * K^3 \in O(\sum C_i * (\sum K_i)^3)$$

3.5 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 tecnica de programación dinámica en un problema de optimización (con lo cual utilizamos el principio de optimalidad de Bellman) 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 facil de ver, y por lo tanto abordaremos esta pseudo-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 $0 \leq k \leq K$ capacidades, siendo K la capacidadad total de la mochila. Puede describirse de la siguiente manera:

$$S(i, k) = \text{máximo valor en el paso } i \text{ para una capacidad disponible } k$$

A cada paso 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 utilizo el mismo 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 hacer en cada paso.

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 $valor(e_i)$ 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 - $peso(e_i)$ 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.

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

$$S(i,k) = max(S(i-1, k - peso(e_i)) + valor(e_i), S(i-1,k)) \tag{7}$$

Como puede verse, cada subproblema es recursivo y ofrece una solución óptima para casos más pequeños. Es decir, con menos objetos.

Al final, en el último paso, es decir para el último objeto y luego de haber analizado cada una de las capacidades, obtendremos el óptimo total que será el resultado de resolver $S(n, K)$, siendo n la cantidad de objetos total.

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 cuales de las mochilas disponibles nos conviene ingresarlo.

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

Como mencionamos el subproblema que analizamos es:

$$S(i, k_1, k_2) = \text{máximo valor en el paso } i \text{ dado que dispongo de dos capacidades } k_1 \text{ y } k_2$$

Lo que haremos como se mencionó, es decidir que haremos con cada objeto. Y esto se define como:

$$S(i, k_1, k_2) = \tag{8}$$

$$max(S(i-1, k_1 - peso(e_i), k_2) + valor(e_i), S(i-1, k_1, k_2 - peso(e_i)) + valor(e_i), S(i-1, k_1, k_2)) \tag{9}$$

Como puede verse, para dos mochilas (y tambien tres) podemos obtener el resultado del máximo general, haciendo un análisis exhaustivo de las posibilidades de manera similar a como se resuelve para una mochila, solo que viendo en dos (o tres) dimensiones, todas las posiblidades para menos objetos.

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.

Como se mencionó en la explicación y desarrollo del problema, el mismo se resuelve también con el uso de matrices, 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 $i - esimo$.

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, por el principio de optimalidad y debido al análisis exhaustivo, 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.

3.6 Experimientos y conclusiones

3.6.1 Test

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

Mochilas con capacidades idénticas y objetos distintos

Este caso se da cuando $K_i = K_j$ con i $\neq$ j , $1 \leq$ i ≤ 3 $1 \leq$ j ≤ 3

A continuación un ejemplo de este tipo de caso:
Con:
M = 3 N = 3
$K_1 = 15$ $K_2 = 15$ $K_3 = 15$
$C_1 = 3$ $P_1 = 5$ $V_1 = 10$
$C_2 = 2$ $P_1 = 3$ $V_1 = 5$
$C_3 = 5$ $P_1 = 4$ $V_1 = 2$

Obtuvimos el siguiente resultado:

$S = 50$
$Q = 10$
$Q_1 = 3$
$Q_2 = 2$
$Q_3 = 5$

Objetos idénticos con mochilas distintas

Este caso se da cuando $C_i = N$ con i = 1 y $K_h \neq K_j$ con h $\neq$ j , $1 \leq$ h ≤ 3 $1 \leq$ j ≤ 3

A continuación un ejemplo de este tipo de caso:

Con:
M = 3 N = 1
$K_1 = 5$ $K_2 = 10$ $K_3 = 15$
$C_1 = 10$ $P_1 = 5$ $V_1 = 10$

Obtuvimos el siguiente resultado:

$S = 60$
$Q = 6$
$Q_1 = 6$

Mochilas distintas con objetos distintos

Este caso se da cuando

$$\sum_{i=1}^{x} C_i = N$$

y $K_h \neq K_j$ con h $\neq$ j , $1 \leq$ h ≤ 3 $1 \leq$ j ≤ 3

A continuación un ejemplo de este tipo de caso:

Con:
M = 3 N = 3
$K_1 = 6$ $K_2 = 12$ $K_3 = 15$
$C_1 = 3$ $P_1 = 2$ $V_1 = 8$
$C_2 = 2$ $P_1 = 5$ $V_1 = 5$
$C_3 = 5$ $P_1 = 4$ $V_1 = 2$

Obtuvimos el siguiente resultado:

$S = 42$
$Q = 9$
$Q_1 = 3$
$Q_2 = 2$
$Q_3 = 4$

Mochilas con capacidades idénticas y objetos iguales

Este caso se da cuando $K_i = K_j$ con i $\neq$ j , $1 \leq$ i ≤ 3 $1 \leq$ j ≤ 3 y $C_h = N$ con h = 1

A continuación un ejemplo de este tipo de caso:
Con:
M = 3 N = 1
$K_1 = 15$ $K_2 = 15$ $K_3 = 15$
$C_1 = 10$ $P_1 = 3$ $V_1 = 6$

Obtuvimos el siguiente resultado:

$S = 60$
$Q = 10$
$Q_1 = 6$

3.6.2 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 del algoritmo calculada anteriormente.

Como la complejidad presenta dos variables marcadas, como son las capacidades de las mochilas y la cantidad de objetos, decidimos, para obtener datos más concisos, realizar los testeos dejando

constante la cantidad de objetos y/o las capacidades.

Luego de realizar varios experimentos, llegamos a la conclusión que, dejando constante la cantidad de objetos y trabajando con las capacidades, el mejor caso para nuestro algoritmo es en el cual **todas las mochilas presentan las mismas capacidades** ya que de esta forma nuestro algoritmo realiza la misma ejecución para cada una de las mochilas sin repetidos.
A continuación mostraremos un gráfico que representa lo enunciado:

Dividiendo por la complejidad calculada se obtuvo lo siguiente:

Se puede ver en el gráfico 3.1 que, dado nuestro algoritmo, al tener las capacidades iguales el mismo realizará la optimización de 3 matrices que simbolizan a cada mochila por separado, 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 $O(K_1 * N) + O(K_2 * N) + O(K_3 * N)$ y como $K_1 = K_2 = K_3$ nos queda que la complejidad para el mejor caso esta acotada por $O(K * N)$ con N constante.
Luego, en el gráfico 3.2, en el cual dividimos el tiempo de ejecución de dicho caso con el tiempo de realizar O(N*k) operaciones nos da una función resultante la cual tiende a 0 cuando la entrada aumenta corroborando lo que acabamos de decir.

Manteniendo el mismo razonamiento, el peor caso para nuestro algoritmo se da cuando hay **mochilas con capacidades distintas** ya que el algoritmo deberá decidir en que mochila colocar el objeto para que el resultado sea óptimo. Por consiguiente, mostraremos un gráfico que representa lo hablado:

Dividiendo por la complejidad calculada se obtuvo lo siguiente:

Se puede ver, tanto en el gráfico 3.2 como 3.4 la funciones resultantes siempre se mantienen por debajo de 1, con lo cual se ve que tanto en el mejor como en el peor caso nuestro algoritmo sigue estando acotado por la complejidad calculada.

Luego, como nuestro algoritmo calcula lo mismo para cualquier tipo de objeto, ya sean todos iguales o no, todos serán casos promedios, salvo el caso en el cual no haya ningún tipo de objeto que entre en alguna mochila, este caso podríamos tomarlo como él mejor ya que nuestro algoritmo corta

al ver que no puede colocar ningún objeto dentro de alguna mochila.
Dejando de lado este caso, el resto de los mismos tendrán un tiempo de ejecución similar.

Mostraremos en el siguiente gráfico de ejemplo lo hablado recientemente:

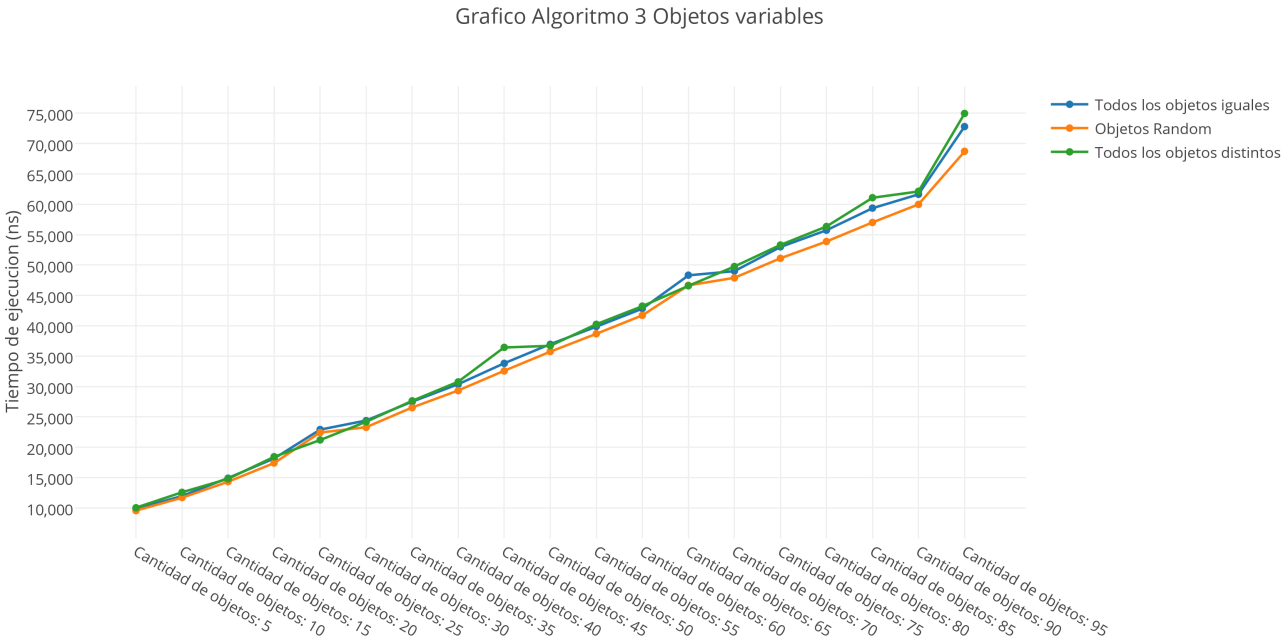


Gráfico 3.5 - *Objetos Variables y Capacidades Constantes*

Se puede ver en el gráfico 3.5 como las 3 funciones, las cuales simbolizan el tiempo de ejecución del algoritmo con todos los objetos iguales, todos distintos y random, presentan un tiempo de ejecución similar, como habíamos comentado anteriormente esto se da ya que nuestro algoritmo realiza las mismas operaciones para cualquier tipo de objeto en las que chequea si es óptimo o no agregar un elemento a alguna mochila.

Trabajando con los objetos variables y habiendo dejado constante la capacidad de las mochilas se llega a que la complejidad esta acotada por la cantidad de elementos multiplicada por la capacidad de las mochilas, en este caso constante para las 3 e igual a 20 en todas las corridas y entradas.

4 Aclaraciones

4.1 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.