Informe 1 Algoritmos y Estructuras de Datos III

Jazmín Alvazer Vico75/15jazminalvarezvico@gmail.com Marcelo Pedraza393/14marcelopedraza314@gmail.co
Uriel Jonathan Rozenberg838/12rozenberguriel@gmail.com Javier María Cortés Conde Tití252/15javiercortescondetito@gmail.com

Grupo
Alvarez Vico Jazmín
Cortés Conde Titó Javier María
Pedraza Marcelo
Rozenberg Uriel Jonathan

8 de septiembre de 2016

Índice

1. Problema 1: Cruzando el puente

1.1. Inrtoducción

En este problema, un grupo formado por arquéologos y caníbales debe cruzar un puente. El mismo sólo puede ser atravezado por dos personas a la vez, y tiene que ser cruzado con una linterna. Como el grupo sólo dispone de una linterna, siempre debe regresar alguno al lado del puente original. Además, en ningún lado del puente puede haber más caníbales que arquéologos. Cada individuo consta de la propiedad "velocidad", un número natural que indica cuánto tiempo tarda en atravezar el puente. Si dos personas atraviezan el puente juntas, lo hacen en el tiempo más grande entre los dos.

1.2. Explicación de la solución

Para resolver este problema, usamos la técnica de backtracking: vamos probando todos los caminos posibles que no generen situaciones en las cuales hay mas caníbales que arqueólogos en algún lado del puente, recortando esos caminos, y tomando el mínimo de los subcaminos recorridos.

Para empezar, todos los casos donde ya se comienze de un lado del puente con más caníbales que arqueólogos deben devolver -1. Además, todo caso en el cual arqueólogos = caníbales y arqueólogos¿=3 , es irresoluble: Supongamos que el problema es: el puente sólo aguanta a una persona, entonces deben pasar de a uno. Podemos ver que el conjunto de estados de este problema es un subconjunto del problema original. Es decir, si encuentramos una solución al problema original, esta solución va a contener todos los estados de mi subproblema. En resumen, si el subproblema no tiene solución, tampoco lo tiene el problema original. Comenzamos entonces con tres caníbales y tres arqueólogos de un lado del puente No podemos pasar un arqueólogo, porque generaría un enchastre del lado izquierdo del puente, entonces hacemos cruzar un canibal. Podríamos pasar cualquiera de los dos grupos ahora, pero pasar un caníbal nos lleva a un inminente fracaso; con dos caníbales de un lado, nunca podremos hacer cruzar un arqueólogo sin generar problemas. Por eso, sólo podemos hacer pasar a un arqueólogo Si pasamos un arqueólogo, quedan mas caníbales del lado original del puente; lo mismo pasa del otro lado si pasamos un caníbal. Llegamos a un callejón sin salida. Es fácil ver que con cantidades iguales mayores a 3 el problema se mantiene. Basta ver que independientemente de la cantidad de caníbales y arqueólogos del lado original del puente, podemos recorrer los pasos anteriores y llegar al mismo problema.

Una vez que decidimos si es posible pasar a todos o no, existen cuatro "movimientos"válidos: Pasar dos caníbales, volviendo un caníbal; pasar dos arqueólogos, y que vuelva un arqueólogo, y pasar un arqueólogo y un caníbal(esto es dos movimientos distintos, ya que puede volver o el arqueólogo o el caníbal). Un movimiento es óptimo si la persona que vuelve es la más rapida de su tipo, ya que de esta manera uno minimiza la cantidad de tiempo usado en llevar de regreso la linterna, siendo que cualquier otra combinación consume más tiempo.

1.3. Pseudocódico

2. Problema 2: Problemas en el camino

2.1. Introducción

En el problema enunciado, Nuestros exploradores se encuentran con una balanza de dos platos, y en el de la izquierda la llave que necesitan. Para que esta sea de utilidad, al tomarla deben mantener la posición de la balanza y para realizar esto cuentan con pesas de peso igual a las potencias de tres

(una pesa por potencia). Entonces, sabiendo el peso de la llave, necesitamos saber que pesas poner en cada plato para mantener el equilibrio original. Podemos pensar que el lado derecho de la balanza equivale a la operación de sustracción y el izquiero a la adición. De esta forma al agregar pesas de un lado o del otro estariamos sumando o restando su peso. Formalmente esto equivale a decir que, si tenemos un entero n queremos sumar o restar potencias de tres, sin repetirlas hasta alcanzar su valor.

2.2. Explicación de la solución

Para resolver este problema, Reescribimos P en la base ternaria, es decir, a este valor los notaremos T. Entonces tenemos $P = \sum_{i=0}^{n} a_i 3^i$ con $0 \leq a_i \leq 2$ y $n = log_3(P)$ entonces $T = a_1, a_2, ..., a_n$. Luego, se toman en orden uno a uno los elementos de T. si a_i es 0, no hacemos nada. Si a_i es 1, en la balanza izquierda ponemos la pesa que corresponde a 3^i. si $a_i = 2$, ponemos una pesa de 3^i en la balanza derecha y sumamos 1 a $a_{(i+1)}$ (y consecuentemente, se cambian los valores posteriores de T para que sigan en base ternaria)

2.2.1. Pseudocódigo

Nota: Para la implementación, en vez de crear el conjunto T, vamos tomando el resto de P y diviendolo por 3 entonces, el valor de la variable rem"sería el equivalente al a_i (i responde al numero de iteraciones ya hechas)

[H]**Pesas**(Natural: P) [1] pesa ← 1 P > 0 rem ← $r_3(P)$ P ← parte entera(P/3) rem = 1 pesa va para la balanza izquierda rem = 2 p← P+1 pesa va para la balanza derecha pesa ← pesa*3

2.2.2. Demostración de Correctitud

Primero probaremos un lema inductivamente: $\forall n \in N$ vale que $2 * 3^n = 3^{n+1} - 3^n$.

Caso base: queremos ver que $2 * 3^0 = 3^1 - 3^0$ $2 * 3^0 = 2 * 1 = 2$ y $3^1 - 3^0 = 3 - 1 = 2$ entonces $2 * 3^0 = 3^1 - 3^0$

Hipótesis Inductiva: $\forall n \in N$ vale que $2 * 3^n = 3^{n+1} - 3^n$

Paso Inductivo: queremos ver que $\forall n \in N$ vale que $2 * 3^{n+1} = 3^{n+2} - 3^{n+1}$

$2 * 3^{n+1} = 2 * 3^n * 3$

por hipotesis inductiva: $2 * 3^n * 3 = (3^{n+1} - 3^n) * 3 = 3^{n+1} * 3 - 3^n * 3 = 3^{n+2} - 3^{n+1}$

luego, como valen el caso base y el paso inductivo, entonces vale $2 * 3^n = 3^{n+1} - 3^n \forall n \in N$

Ahora para probar la correctitud del algoritmo, desarrollaremos algunos conceptos algebraicos. Por el algoritmo de división de Euclides sabemos que podemos escribir

$P = q * 3 + r_3(P)$ con $0 \leq r_3(P) \leq 2$ luego, podemos escribir $q = z * 3 + r_3(q)$ entonces $P = (z * 3 + r_3(q)) * 3 + r_3(P)$ y así recursivamente y aplicando la distributividad de la suma podemos descomponer p en las potencias de 3. finalmente obtendriamos $p = \sum_{i=0}^{n} a_i 3^i$ con $0 \leq a_i \leq 2$. Ahora nuestro problema es la aparición de $a_i = 2$ en la descomposición puesto que solo tenemos una pesa por potencia de tres. Pero por lo visto en el lema anterior, $2 * 3^i = 3^{i+1} - 3^i$ esto equivale a poner la pesa 3^{i+1} en la balaza izquierda y la 3^i en la derecha obteniendo el equivalente a 2 pesas 3^1 en la izquierda. aplicando este proceso recursivamente, obtenemos una descomposición de P sumando y restando sus potencias de tres sin repeticiones.

2.2.3. Demostración de Complejidad

El algoritmo consta de un ciclo, dentro de cada iteración, se calcula el resto y hace dos comparaciones todas estas operaciones se ejecutan en O(1), mientras que enviar la pesa a cada balanza es agregar un número al final de una lista, ya que el modelo de lista que usamos es "METERMODELO", esto se realiza en O(1). Entonces, dentro de cada iteración solo se hacen operaciones en O(1). Esto hace que la complejidad sea la cantidad de iteraciones que hace el ciclo.

Si nos abstraemos del if, el ciclo tiene como condición que P¿0, siendo P el valor de entrada. Como podemos observar P es modificado dividiendose por 3 por cada iteración. Por algebra básica sabemos que se necesesitaran como mucho $\log_3(P)+1 iteraciones para q P sea menor a 0. Ahora, tomando en cuenta el if hay un$ $a_1, ..., a_n donde P = \sum_{i=0}^{n} a_i 3^i$ con $0 \leq a_i \leq 2$ y $n = log_3 P$, siendo el valor de rem el valor de $a_i en la iesima iteracion. Cuando hacemos que P < -P+1 no s queda a que en la proxima iteracion rem valdria una mas, e$ $1) pasaria a valer una mas de lo q valdria en T. Ahora, siendo estos que para todo a_k valen entre 0 y 2, si a_(i+1) en vez de vale$ $2) valdria una mas tambien, y si a_(i+2) valia 2, pasaria a valer 0 y a_(i+3) valdria una mas y asi sucesivamente hasta llega$ $1) que entraria a valer 1 y a_n valdria 0. Luego no habria mas casos donde a_j (para i < j <= n) donde a_j sea 2, entonces no h$ $1 iteraciones$

Entonces, la complejidad del algoritmo es de $\mathcal{O}(Log_3(P))$. Luego, se puede probar que Log(P) es $\mathcal{O}(\sqrt{(P)})$

2.3. Experimentación

Al momento de la experimentacion inicialmente teniamos la idea de que, al no tener mejores ni peores casos a nivel de complejidad teorica no importaba mucho que valores de P se les pusiese, iba a ser siempre creciente.

2.3.1. Ánalisis complejidad teórica

Para el ánalisis complejidad teórica decidimos simplemente medir los tiempos del algoritmo aumentando el valor de entrada desde 1 hasta la cota puesta por el enunciado. Luego despues de probar un poco, logramos poner una funcion O(Log(n)) que se asemeje un poco a los resultados de las mediciones del algoritmo para poder compararlo. Terminamos usando la funcion $500 * log_3(x)$. Ademas, notamos que no quedaba muy claro como es que las mediciones se asemejaban a $log_3(x)$ y entonces decidimos cambiar la escala: En vez de presentar un grafico que muestre los ejes (X,Y) = (peso de la llave, mediciones) lo vamos a mostrar como (X,Y) = (Log(peso de la llave),Log(mediciones))

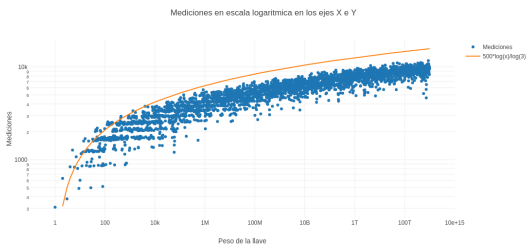

Figura 1: Mediciones de la implementación comparado con $500 * log_3 n$ en escala logaritmica en X e Y

2.3.2. Análisis

Luego realizado el grafico notamos que a pesar de de que las mediones simulaban cumplir las condiciones de la complejidad, esperabamos un comportamiento mas cercano a que se vaya subiendo en rectas horizontales es decir, que como se hacia la misma cantidad de iteraciones para cada valor del logaritmo en base 3 pero en vez de eso estas tenian un tipo de patron ligeramente diferente

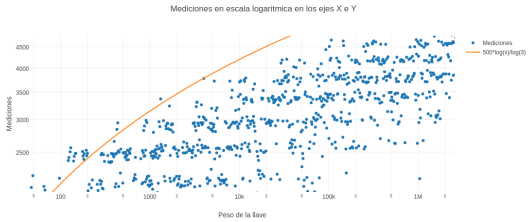

Figura 2: Mediciones de la implementación comparado con $500 * log_3 n$ zoom en escala logaritmica en X e Y

Si bien se organizaban como lineas rectas (a pesar de la escala) no se acomodaba su .ºrdenrespecto al logaritmo en base 3. Como conclusion terminamos pensando que esto se debia a la operacion de "meter la pesa en una balanza"que en la implentacion del problema se traducia a introducir un entero al final de una lista, lo cual terminaba costando $O(1)$

Para ver si esto era cierto, hicimos nuevas mediciones, tomando para los pesos solo 3 casos: -Potencias de 3: Que en la representacion ternaria, contaria solo con un 1 y puros ceros -Potencias de 3 menos 1:Que en la representacion ternaria, contaria con todos 2 -Suma de potencias de 3:Que en la representacion ternaria, contaria con todos 1

y como resultado del experimento nos quedo este grafico

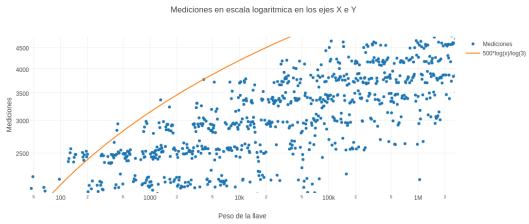

Figura 3: Comparaciones de la implementación comparando los casos en escala logaritmica en X e Y

En la imagen se puede notar que el caso de suma de potencias de 3 toma mucho mas que los otros dos casos, concluimos que esto se debe incialmente a que, en el caso de solo potencias de 3, se hace un solo acceso a la listar mientras q el caso de potencias de 3 menos 1 tal como seria el resultado al problema, se pone una pesa de la potencia de 3 del lado de la llave y una pesa de 1 en el otro, teniendo asi solo 2 accesos a la lista, cosa que en la implementacion esto se representa sumandole 1 luego de cada division por 3 al peso de la llave en cada iteracion.

Como conclusion final, a tomar en cuenta, consideramos que los mejores casos son los de potencias de 3 y los peores son los de sumas de potencias de 3.

3. Problema 3: Guardando el tesoro

3.1. Introducción

En este problema, los exploradores se encuentran frente a muchos tesoros que desearían poder llevarse. Para esto, cuentan con varias mochilas, cada una con una determinada capacidad de peso que puede cargar. Los tesoros son de distintos tipos, y cada tipo tiene un peso y un valor determinado. El objetivo es encontrar la manera optima de llenar las mochilas para poder llevar el mayor valor posibe.

Formalmente, tenemos un conjunto de elementos que tienen como propiedad dos naturales asociados (el peso y el valor). Entonces podemos inferir que existen dos criterios de ordenamiento asociados respectivamente a estos valores. Al mismo tiempo tenemos otro conjunto de elementos que posee como propiedad un natural asociado (capacidad). Nuestro objetivo es seleccionar una combinación de los primeros objetos, restringida por las capacidades dadas,de modo de maximizar la sumatoria del valor de los mismos.

3.2. Explicación de la solución

Inicialmente creimos que este problema podría resolverme mediante un algoritmo de BackTracking, sin embargo observamos que nunca lograríamos conseguir la complejidad pedida puesto que este tipo de algoritmos conlleva una complejidad exponencial sperior a la pedida. Finalmente pudimos resolverlo mediante prmathcalOamación dinámica. Nuestro algoritmo principal llama a dos funciones que utilizan la recursividad para podes obtener en un caso el valor óptimo, y en el otro "las mochilas llenas."Para facilitar el entendimiento del algoritmo introduciremos el concepto de "Hipercubo". Un hipercubo es el analogo n-dimensional de un cuadrado (n=2) o un cubo (n=3). En particular en nuestro problema tenemos un hipercubo de dimension cuatro, estaa figura se denomina "teseracto"sin embargo por comodidad seguiremos llamandole hipercubo en el resto del informe. Para poder imaginar este concepto, podriamos visualizarlo en nuestro mundo tridimencional como un vector con cubos adentro. la cantidad de cubos dependera del largo de nustro cuarta .ªrista".

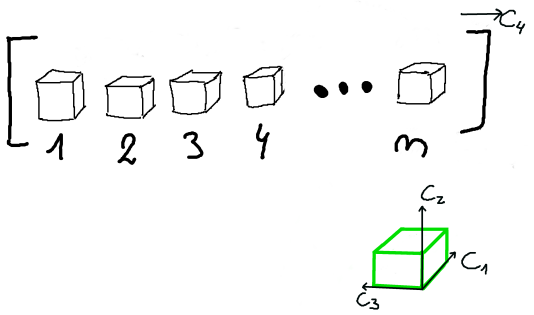

Figura 4: imagen explicativa del concepto de hipercubo

observación: nuestro algoritmo descarta los tesoros que tienen un peso mayor al maximo de capacidad entre las mochilas. Asumimos en el pseusocodigo que nuestra entrada cumple esa propiedad.

3.2.1. Pseudocódigo

[H]**guardandoTesoro**(mochilas: vector<mochila>, cofre: vector<tesoro>) [1] objXpesos ← Hi-

percubo() incializado en -1 (en las posiciones donde no puede haber objetos incializo en 0) $\mathcal{O}()$ sol= ValorOptimo(objXpesos,cofre,cofre.size-1,capacidades de las mochilas) LlenarMochilas(objXpesos, cofre, mochila1, mochila2, mochila3) return (sol, mochilas)

[H]**LlenarMochilas**(objetoXpeso: hipercubo, cofre:vector<tesoro>, m1,m2,m3:mochilas) [1] desde i= cofre.size-1 hasta i=0 obj $\leftarrow$ cofre[i] cap1 $\leftarrow$ m1.Capacidad cap2 $\leftarrow$ m2.Capacidad cap3 $\leftarrow$ m3.Capacidad $i = 0$ Agregar obj en cualquier mochila en la que entre. valM1 $\leftarrow$ obj.valor + ValorOptimo(objetoXPeso, cofre, i,cap1-obj.peso,cap2,cap3) valM2 $\leftarrow$ obj.valor + ValorOptimo(objetoXPeso, cofre,i,cap1,cap2-obj.peso,cap3) valM3 $\leftarrow$ obj.valor + ValorOptimo(objetoXPeso, cofre,i, cap1, cap2,cap3-obj.peso) MeterEnCorrecta(valM1,valM2,valM3,m1,m2,m3,obj)

[H]**ValorOptimo(objXpeso:hipercubo, cofre:vector<tesoro>, objeto:int, peso1: int, peso2:int, peso3:int)** [1] pesoObj $\leftarrow$ cofre[objeto].peso $\mathcal{O}(1)$ pesoVal $\leftarrow$ cofre[objeto].valor $\mathcal{O}(1)$

$peso1 < 0 \vee peso2 < 0 \vee peso3 < 0$ return -1 O(1)

objetoxPesos[objeto][peso1][peso2][peso3] $\neq$ -1 return objetoxPesos[objeto][peso1][peso2][peso3] $\mathcal{O}(1)$ objeto = 0 val $\leftarrow$ 0 O(1) peso1 $\geq$ pesoObj $\vee$ peso2 $\geq$ pesoObj $\vee$ peso3 $\geq$ pesoObj val $\leftarrow$ valorObj O(1) objetoxPesos[objeto][peso1][peso2][peso3] $\leftarrow$ val $\mathcal{O}(1)$ return val $\mathcal{O}(1)$

PosiblesSolus $\leftarrow$ vector<int> $\mathcal{O}(1)$ sinObj $\leftarrow$ ValorOptimo(objetoxPesos, cofre, objeto -1, peso1, peso2, peso3) PosiblesSolus.Agregar(sinObj) peso1-pesoObj $\geq$ 0 objen1 = valorObj + ValorOptimo(objetoxPesos, cofre, objeto - 1, peso1 - pesoObj, peso2, peso3) PosiblesSolus.Agrergar(objen1) peso2 - pesoObj $\geq$ 0 objen2 = valorObj + ValorOptimo(objetoxPesos, cofre, objeto - 1, peso1, peso2 - pesoObj, peso3) PosiblesSolus.Agregar(objen2) peso3-pesoObj $\geq$ 0 objen3 = valorObj + ValorOptimo(objetoxPesos, cofre, objeto - 1, peso1, peso2, peso3 - pesoObj) PosiblesSolus.Agregar(objen3) valor=Max(PsiblesSolus) objetoxPesos[objeto][peso1][peso2][peso3] = valor return valor

3.2.2. Demostración de Correctitud

Presentaremos la función matemática que modela nuestro problema:

$f(o, p_1, p_2, p_3) = 0$
$f(n, p_1, p_2, p_3) = Max(V_n + Max(f(n-1, p_1 - p_n, p_2, p_3), f(n-1, p_1, p_2 - p_n, p_3), f(n-1, p_1, p_2, p_3 - p_n)), f(n-1, p_1, p_2, p_3))$
Donde n representa el numero de tesoro, V_n y P_n su valor y su peso respectivamente. p_1, p_2 *y* p_3 son las capacidades de cada mochila.

Para la demostración utilizaremos inducción global en n.

Caso base: queremos ver que $f(0, p_1, p_2, p_3)$ resulta ser el valor máximo que se puede obtener con 0 objetos: $f(0, p_1, p_2, p_3) = 0$ al tener 0 objetos el valor de los mismos es 0 de modo que es el máximo valos posible.

Hipotesis Inductiva: $\forall k < n$ vale que $f(k, p_1, p_2, p_3)$ da el valor máximo que se puede obtener con k objetos.

Ahora queremos ver que $f(n, p_1, p_2, p_3)$ da el valor óptimo para n objetos.

$f(n, p_1, p_2, p_3) = Max(V_n + Max(f(n-1, p_1 - p_n, p_2, p_3), f(n-1, p_1, p_2 - p_n, p_3), f(n-1, p_1, p_2, p_3 - p_n)), f(n-1, p_1, p_2, p_3))$
por hipótesis inductiva (como $n-1 \leq k$ para algun k) sabemos que $f(n-1, p_1 - p_n, p_2, p_3), f(n-1, p_1, p_2 - p_n, p_3), f(n-1, p_1, p_2, p_3 - p_n) y f(n-1, p_1, p_2, p_3)$ son los valores optimos que se pueden conseguir con n-1 objetos restando (o no) el peso del objeto n de modo de luego poder guardarlo en alguna mochila (o no). utilizaremos los renombres $V_{01}, V_{02}, V_{03}, V_{00}$ respectivamente. Entonces tenemos:
$f(n, p_1, p_2, p_3) = Max(V_n + Max(V_{01}, V_{0,2}, V_{03}), V_{00})$
Podemos ver que esta función compara el valor optimo de llenar las mochilas con n-1 tesoros y el tesoro n, con el valor optimo de llenar las mochilas con n-1 tesoros sin el tesoro n (de esta forma se tiene cuenta el caso en el que el mejor valor se obtuviene de poner algun objeto de los n-1 anteriores que impide luego meter el tesoro n).
Como la función Max devuelve el mayor valor, el resultado sera el óptimo.
Como valen P(0)..p(k) $\forall k < n$ y vale p(n) vale p(n) $\forall n \in N \cup 0$

3.2.3. Demostración de Complejidad

Como se observa en el inciso () la complejidad del algoritmo GuardarTesoro es $\mathcal{O}(\prod_{i=1}^{3} K_i * T)$ donde K_i representa la capacidad de cada mochila y T la cantidad de tesoros. Ademas esta complejidad es aportada por el algoritmo ValorOptimo, entonces basaremos nuestra demosrtación en el mismo.

Primero haremo unas observaciones preliminares:

- El volumen de un cubo es es $\prod_{i=1}^{3} A_i$ donde cada A_i es una arista que representa el eje de la altura, el ancho o el largo. Como explicamos en la introducción gracias a nuestro modelado, sabemos que llenar una posición del cubo nos cuesta $\Theta(1)$ entonces llenarlo entero nos costará el equivalente al volumen del mismo. Podriamos visualizarlo como subdividir un cubo en cubos pequeños de dimension 1X1X1.

- Llenar un hipercubo con un valor determinado (como ocurre en la linea 1 de guardandoTesoro) cuesta $\mathcal{O}(\prod_{i=1}^{3} K_i * T)$ ya que hay T cubos para llenar.

- Cuando se llama a LlenarMochilas desde GuardarTesoro cuesta $\mathcal{O}(T)$ ya que en en las lineas 9,10,11 cuando se llama a ValorOptimo, el hipercubo objetoXpeso cuenta con todos lo valores ya calculados entrando en el if (linea 6) que devuelve en $\mathcal{O}(1)$ el valor.

Volviendo a la demostración, tanto en el pseudocódigo como en la función matemática podemos ver que la recursión se realiza tantas veces como la cantidad total de tesoros. Para llenar un cubo, siempre debemos recurrir al cubo anterior, uno puede pensar que esto aportaría complejidad, sin embargo, al guardar estos valores solo construimos estos cubos una vez, y la complejidad por acceso es $\Theta(1)$.

Otra manera de verlo, es analoga a la explicacion de la complejidad de llenar un cubo. Al estar llenando un hipercubo la complejidad sera quivalente al hipervolumen del mismo, es decir, multiplicariamos las tres aristas anteriores por una que sería la cuarta dimensión(en este caso los tesoros).

De cualquier manera podemos concluir que la complejidad es $\mathcal{O}(\prod_{i=1}^{3} K_i * T)$

Ahora probemos que esta complejidad es menor a la pedida $(\mathcal{O}((\sum_{i=1}^{3} K_i)^3 * T))$

$\prod_{i=1}^{3} K_i = K_1 * K_2 * K_3$ sea $K_max = Max(K_1, K_2, K_3)$ y $K_o = K_1^3 + K_2^3 + K_3^3 - K_max^3$ entonces $K_1 * K_2 * K_3 < K_m^3 < K_m^3 + K_o = K_1^3 + K_2^3 + K_3^3 < (\sum_{i=1}^{3} K_i)^3$ entonces $\prod_{i=1}^{3} K_i * T < \sum_{i=1}^{3} K_i)^3 * T$

3.3. Experimentación

3.3.1. Resultados

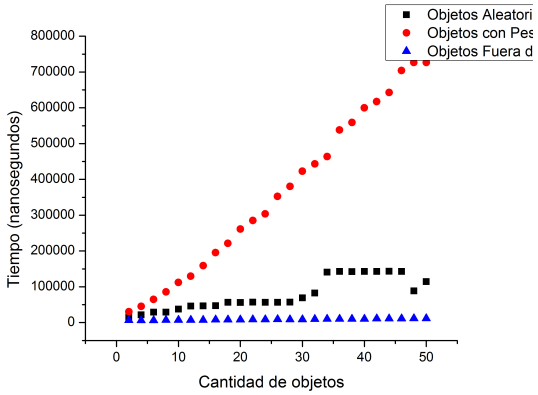

Figura 5: Comparación de los tres casos de nuestro algoritmo variando la cantidad de objetos

En esta figura podemos observar no solo la tendencia lineal de la complejidad del algoritmo sino también la variación de las pendientes en cada caso. Notemos que a mayor pendiente, la ejecución toma más tiempo, es decir es "más lenta". En particular podemos destacar que en el caso de que los objetos tengan todos peso mayor a la capacidad de las mochilas, la recta tiende a ser constante.

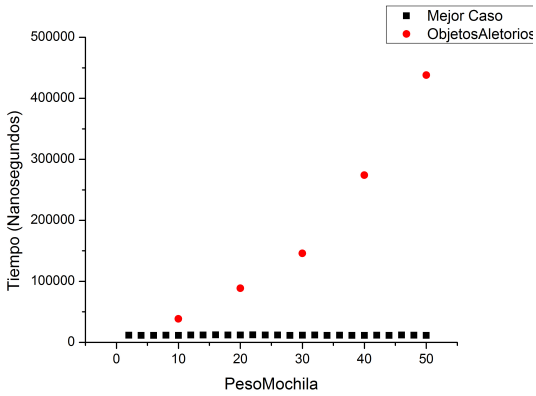

Figura 6

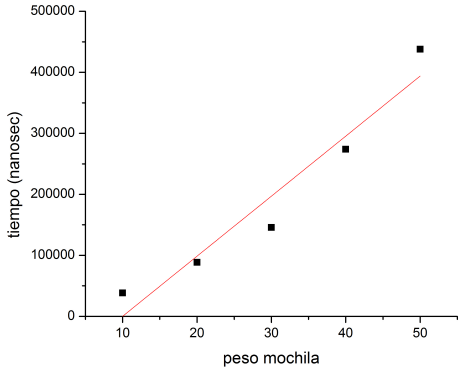

Figura 7

3.3.2. Análisis