

Algoritmos y Estructuras de Datos III

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

Informe 1

Integrante	LU	Correo electrónico
Jazmín Alvazer Vico	75/15	jazminalvarezvico@gmail.com
Marcelo Pedraza	393/14	marcelopedraza314@gmail.com
Uriel Jonathan Rozenberg	838/12	rozenberguriel@gmail.com
Javier María Cortés Conde Titó	252/15	javiercortescondetito@gmail.com

Reservado para la cdra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Problema 1: Cruzando el puente	3
1.1. Introducción	3
1.2. Explicación de la solución	3
1.3. Pseudocódigo	3
1.4. Demostración de Correctitud	5
1.5. Demostración de Complejidad	6
1.6. Experimentación	6
2. Problema 2: Problemas en el camino	6
2.1. Introducción	6
2.2. Explicación de la solución	6
2.2.1. Pseudocódigo	7
2.2.2. Demostración de Correctitud	7
2.2.3. Demostración de Complejidad	8
2.3. Experimentación	8
2.3.1. Análisis complejidad teórica	8
2.3.2. Análisis	9
3. Problema 3: Guardando el tesoro	10
3.1. Introducción	10
3.2. Explicación de la solución	10
3.2.1. Pseudocódigo	11
3.2.2. Demostración de Correctitud	14
3.2.3. Demostración de Complejidad	14
3.3. Experimentación	15
3.3.1. Resultados y análisis	15

1. Problema 1: Cruzando el puente

1.1. Introducción

En este problema, un grupo formado por arqueólogos y caníbales debe cruzar un puente. El mismo sólo puede ser atravesado 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 arqueólogos. Cada individuo consta de la propiedad “velocidad”, un número natural que indica cuánto tiempo tarda en atravesar el puente. Si dos personas atraviesan el puente juntas, lo hacen en el tiempo más grande entre los dos. Formalmente queremos encontrar la solución óptima para minimizar el tiempo que toma atravesar un trayecto por dos conjuntos de individuos cuando uno de ellos no puede ser mayor al otro(exceptuando el caso en que uno de ellos es vacío).

1.2. Explicación de la solución

Para resolver este problema, usamos la técnica de backtracking: utilizamos una función recursiva para poder probar todos los caminos posibles que no generen situaciones en las cuales hay mas caníbales que arqueólogos en algún lado del puente. Finalmente elegimos el mínimo de los subcaminos recorridos.

Para empezar, todos los casos donde ya se comience de un lado del puente con más caníbales que arqueólogos no tienen solución. El caso en el cual no haya arqueólogos usamos una instancia aparte que conlleva muchas condiciones booleanas. Este caso se resuelve simplemente utilizando al caníbal más rápido para llevar a los demás. Para ver que esto es óptimo, basta ver que todos los individuos deben pasar al menos una vez. Para cualquier ida y vuelta la linterna, que vaya y vuelva el caníbal más rápido genera el mismo estado que cualquier otra opción, con un menor “costo”.

Para los demás casos hay que analizar los cinco “movimientos”válidos existentes para atravesar el puente: Pasar dos caníbales, dos pasar arqueólogos, pasar sólo un arqueólogo, pasar sólo un caníbal o pasar un arqueólogo y un caníbal. Decidimos pensar la ida y vuelta de la linterna como un sólo estado, para no tener que registrar la ubicación de la misma. Para guardar la información de posiciones vamos a registrar el estado del lado izquierdo(o inicial) del puente, ya que cualquier individuo que no esté de ese lado, necesariamente estará del otro. Utilizamos una matriz de dimensión $N+1 \times M+1$ y cuatro listas(N_A y M_A para indicar quiénes estan en el punto de partida y N_B y M_B para el punto de llegada). Cada vez que modifiquemos N_A y M_B se entrará en la recursión copiando la matriz y registrando en la misma el nuevo estado recorrido. Todos los resultados posibles se guardan en una lista para poder finalmente devolver el valor minimo de la misma el cual será nuestra solución. La función PuedenSalir verifica que, en caso de que crucen el puente los individuos indicados, no generará problemas en ninguno de los lados del puente.

1.3. Pseudocódigo

A lo largo de esta sección renombraremos tipos, para facilitar la lectura.

- vint es un arreglo de enteros.
- ins es una dupla (vint, vint).
- instancias es un conjunto de ins.

Tambien nos ahorraremos algunos pseudocódigos que son casos análogos entre Arqueólogos y caníbales.

```
cant(ls: vint)  $\rightarrow$  res : Nat
```

```
  aux  $\leftarrow$  0
```

```
  Desde i = 0 hasta Tam(ls)-1
```

```
    if(ls[i] != 0)
```

```
      aux  $\leftarrow$  aux+1
```

```
    fi
```

```
  res  $\leftarrow$  aux
```

```
esValida(instancia: ins)  $\rightarrow$  res : bool
```

```
  res  $\leftarrow$  (cant( $\pi_1$ (instancia))  $\geq$  cant( $\pi_2$ (instancia)))  $\vee$  (cant( $\pi_1$ (instancia)) = 0)
```

```
voyConA(In VelElem1:Nat, InOut mMinActual:Nat, InOut ArqA: vint, InOut ArqB: vint)
```

```
  Desde i = 0 hasta Tam(ArqA)-1
```

```
    if(ArqA[i] != 0)
```

```
      valElem  $\leftarrow$  ArqA[i]
```

```
      maximo  $\leftarrow$  max(valElem, velElem1)
```

```
      mMinActual  $\leftarrow$  mMinActual + maximo
```

```
      ArqB[i] = valElem
```

```
      ArqA[i] = 0
```

```
      VuelvoCon()
```

```
      ArqB[i] = 0
```

```
      ArqA[i] = valElem
```

```
    fi
```

BT(M: Matriz nula de dim($|N_A|+1$ x $|M_A|+1$), N_A, M_A, N_B, M_B : Listas)

```

if  $N_A$  es vacía then
    min  $\leftarrow$  mínimo de  $M_A$ 
    devolver min*( $|M_B| - 1$ ) + Sumatoria(  $M_B$  sin el elemento min)
end if
M[ $|N_A|+1$ ][ $|M_A|+1$ ]  $\leftarrow$  1
T  $\leftarrow$  []
while i= 0,1,2 do
    while j= 0,...,2-i do
        if  $\neg$ (ij0) PuedenSalir(i,j, $|N_A|$ ,  $|M_A|$ ,  $|N_B|$ ,  $|M_B|$ ) then
            se le quitan i elementos a  $N_A$  y se le agregan a  $N_B$ 
            se le quitan j elementos a  $M_A$  y se le agregan a  $M_B$ 
            (si i,j=1 se quita el más rápido, si i,j=2 se quitan el más rápido y el más lento)
            t  $\leftarrow$  Max(de los elementos transferidos)
            while k=0; k $\leq$ 2; k++ do
                while l=0; k+l $\leq$ 2; l++ do
                    if PuedenSalir(k,l, $|N_A|$ ,  $|M_A|$ ,  $|N_B|$ ,  $|M_B|$ )  $\wedge$  M[ $N_A+k$ ][ $M_A+l$ ]=0 then
                        se le quitan k elementos a  $N_B$  y se le agregan a  $N_A$ 
                        se le quitan l elementos a  $M_B$  y se le agregan a  $M_A$ 
                        (se quitan los dos más rápidos)
                        t=t+Max(de los elementos transferidos)+ BT(M, $N_A$ ,  $M_A$ ,  $N_B$ ,  $M_B$ )
                        T.Agregar(t)
                    end if
                end while
            end while
        end if
    end while
end while
devolver Min(T)

```

1.4. Demostración de Correctitud

Para mostrar la correctitud de nuestro algoritmo queremos ver dos cosas:

- Recorre todos los casos:

Si planteamos nuestras posibilidades como una matriz, vemos que todo lo pertinente al triángulo superior no tiene solución de manera trivial o se plantea un caso particular; estos casos son: los que hay sólo caníbales, o hay más caníbales que arqueólogos desde un comienzo. Entonces nos concentraremos en el triángulo inferior. Al ser nuestro algoritmo una función recursiva solo terminará al llegar al caso base. Podemos ver que nuestro caso base se alcanza al obtener las dos listas iniciales(N_A , M_A) vacías. Como vemos en el primer conjunto de for, creamos tuplas (i,j) tal que generan todas las posibles formas de pasar el puente, si esa tupla puede realmente pasar por el puente, se crea una copia de las listas N_A , M_A , N_B , M_B donde, dependiendo del i, se sacan esa cantidad de elementos la lista N_A y se ponen en N_B caso análogo con j y M 's. Al generar todas las instancias del cruce del puente, ahora tenemos que saber todas las formas de volver el puente. De una forma similar a la explicada antes los últimos dos whiles funcionan tal que generan todas las formas de volver del puente.

- No hace ciclos:

Nuestro algoritmo no admite ciclos puesto que tiene una condición booleana sobre la matriz, la cual no permite volver a una instancia anterior en las que esté el mismo conjunto de caníbales y arqueólogos que en los casos anteriores. La misma consiste en que sólo pueden regresar los individuos si y sólo si en la posición (i,j) (donde $i = N_A + (\text{los arqueólogos que regresen})$ y $j = M_A + (\text{los caníbales que regresan})$) hay un cero. Cada vez que se llama a la función recursivamente la posición inicial en el punto A es marcada con un 1. De esta forma es imposible realizar ciclos.

1.5. Demostración de Complejidad

Si pensamos todas nuestras posibilidades como un árbol de estados, haciendo que la función `idaz_vuelta` genera 3 hijos y 4 hijos respectivamente (la función `vuelta` genera 5 hijos, pero uno de ellos es siempre un estado por el que ya pasamos). Además, la altura nunca va a ser mayor que $n * m/2$, ya que tenemos una matriz para verificar que no pasemos dos veces por el mismo estado. Además, las operaciones de cada estado consisten en `recorrer` todas las posibilidades de individuos a mandar, siendo esto En resumen, la cantidad máxima de resultados es altura del árbol por la `amplitud`: Podemos ver que la complejidad es $12^{n*m/2}$

1.6. Experimentación

Para este problema generamos dos hipótesis: Suponemos que siempre que van dos individuos, si son del mismo tipo van los dos extremos (el más lento y el más rápido) y si son distinto tipo, pasa el mínimo de todo el grupo, y el máximo del otro tipo. En el caso de la vuelta, siempre se vuelve con los más veloces posibles. Queremos ver si esta estrategia sigue dando resultados óptimos, y cuál es la relación de complejidad

Por otro lado, suponemos que el conjunto particular de caníbales y arqueólogos en un lado de puente es indistinguible en los estados, y que alcanza con contemplar la cantidad. Es decir, si la hipótesis anterior se cumple, no sería necesario probar cada conjunto particular, reduciendo considerablemente la complejidad.

2. Problema 2: Problemas en el camino

2.1. Introducción

En el problema enunciado, Nuestros exploradores se encuentran con una balanza de dos platos; en el de la izquierda la llave que necesitan. Para que esta sea de utilidad, al tomarla deben mantener el equilibrio de la balanza, y para realizar esto cuentan con pesas de peso igual a las potencias de tres (una pesa por potencia). Sabiendo el peso de la llave, necesitamos saber qué 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 izquierdo a la adición. De esta forma, al agregar pesas de un lado o del otro estaríamos sumando o restando su peso. Esto equivale a decir que, dado un entero P queremos componerlo en sumas y restas de potencias de tres distintas.

2.2. Explicación de la solución

Para resolver este problema, Reescribimos P en la base ternaria, y a esta secuencia la 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 $T = a_1, a_2, \dots, a_n$. Luego,

tomamos en orden creciente los elementos de T. Si a_i es 0, no hacemos nada. Si a_i es 1, en la balanza izquierda colocamos la pesa que corresponde a 3^i . Si $a_i = 2$, colocamos 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 continúe 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 dividiéndolo por 3. Entonces, el valor de la variable "rem" sería el equivalente al a_i (i responde al número de iteraciones ya hechas)

Pesas(Natural: P)

```

1: pesa  $\leftarrow 1$ 
2: while P > 0 do
3:   rem  $\leftarrow r_3(P)$ 
4:   P  $\leftarrow$  parte entera(P/3)
5:   if rem = 1 then
6:     pesa va para la balanza izquierda
7:   else
8:     if rem = 2 then
9:       p  $\leftarrow$  P+1
10:      pesa va para la balanza derecha
11:    end if
12:  end if
13:  pesa  $\leftarrow$  pesa*3
14: end while
```

2.2.2. Demostración de Correctitud

Primero probaremos dos lemas: $\forall n \in \mathbb{N}$ vale que $2 * 3^n = 3^{n+1} - 3^n$.

lema 1: $2 * 3^n = 3^{n+1} - 3^n \iff 2 * 3^n + 3^n = 3^{n+1} \iff (2 + 1) * 3^n = 3^{n+1} \iff 3 * 3^n = 3^{n+1} \iff 3^{n+1} = 3^{n+1}$

lema2 : probaremos por inducción que ${}_0^n 3^i < 3^{n+1}$ caso base: $n0$. $3^0 < 3^1$ Paso Inductivo: Supongo que ${}_0^n 3^i < 3^{n+1}$ Quiero ver que ${}_0^{n+1} 3^i < 3^{n+2}$ ${}_0^{n+1} 3^i = {}_0^n 3^i + 3^{n+1}$ por Hipótesis Inductiva, ${}_0^n 3^i + 3^{n+1} < 3^{n+1} + 3^{n+1} < 3^{n+2}$

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 obtendríamos $p = \sum_{i=0}^n a_i 3^i$ con $0 \leq a_i \leq 2$.

Ahora nuestro problema es la aparición de $a_i = 2$ en la descomposición, puesto que sólo tenemos una pesa por potencia de tres.

Probaremos por inducción global que dado un natural P y su representación en potencias de dos a_i , $\forall n \in \mathbb{N}$ vale $\sum_0^n a_i * 3^i = \sum_0^{n+1} b_i * 3^i$ con $0 \leq a_i \leq 2$ y $-1 \leq b_i \leq 1$. Podemos ver que $b_{n+1} \neq -1$, ya que P esto generaría un número negativo, y p es natural.

caso base: $\sum_0^n a_i * 3^i = \sum_0^{n+1} b_i * 3^i$ con $0 \leq a_i \leq 2$ y $-1 \leq b_i \leq 1$ Existen tres opciones. En el caso que $a_0=0$, $b_0=0$ y $b_1=0$. En el caso $a_0=1$, $b_0=1$ y $b_1=0$. Finalmente, en el caso $a_0=2$, usamos el lema 1, lo que nos da $b_0 = -1$ y $b_1=1$

paso inductivo: supongo que $\sum_0^n a_i * 3^i \sum_0^{n+1} b_i * 3^i \text{ con } 0 \leq a_i \leq 2y - 1 \leq b_i \leq 1$ quiero ver que $\sum_0^{n+1} a_i * 3^i = \sum_0^{n+2} b_i * 3^i \text{ con } 0 \leq a_i \leq 2y - 1 \leq b_i \leq 1$
 $\sum_0^{n+1} a_i * 3^i \sum_0^n a_i * 3^i + a_{n+1} * 3^{n+1}$ Por Hipótesis Inductiva, $\sum_0^n a_i * 3^i + a_{n+1} * 3^{n+1} \sum_0^{n+1} b_i * 3^i + a_{n+1} * 3^{n+1} \sum_0^n b_i * 3^i + a_{n+1} * 3^{n+1} + b_{n+1} * 3^{n+1}$ Observemos $a_{n+1} + b_{n+1}$. Esta suma puede dar 0, 1, 2, 3. El caso 0 y 1 son triviales. Basta llamar al nuevo b_{n+1} a la suma y b_{n+2} 0 El caso 3, b_{n+1} pasa a ser 0, y b_{n+2} pasa a ser 1 En el caso 2, por lema 1 podemos decir que b_{n+1} pasa a ser -1, y b_{n+2} pasa a ser 1.

$$p = \sum_{i=0}^n a_i 3^i = \sum_{i=0}^{n-1} a_i 3^i + a_n * 3^n$$

por hipotesis inductiva, sabemos que para $0 \leq i \leq n - 1, -1 \leq a_i \leq 1$. Por Euclides, como explicamos antes, $0 \leq a_n \leq 2$ en los primeros dos casos es trivial la validez de la proposición. Veamos el caso $a_i = 2$: *Porel lema, podemos escribir* $2 * 3^n = 3^{n+1} - 3^n$ entonces podemos escribir $a_i p = \sum_{i=0}^n a_i 3^i = \sum$

Pero por lo visto en el lema anterior, $2 * 3^i = 3^{i+1} - 3^i$. Esto equivale a colocar la pesa 3^{i+1} en la balanza izquierda y la 3^i en la derecha, obteniendo el equivalente a 2 pesas 3^i 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 colocar la pesa en cada balanza es agregar un número al final de una lista. Como el modelo de lista que usamos es "List", 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.

Podemos ver en el pseudocódigo que sólo se incrementa a P cuando es congruente a 2 módulo 3. En caso contrario, es fácil ver que la complejidad es de $O(\log_3(P))$, ya que en cada iteración se divide a P por 3, quedándose con la parte entera. Supongamos el caso en el cual, para cada iteración, a P se la divide por 3, se calcula la parte entera y se le incrementa 1 (es decir, siempre se entra en la condición *rem2*). Se puede ver que ese incremento no genera un cambio en el orden de complejidad. Entonces, la complejidad del algoritmo es de $O(\log_3(P))$. Luego, se puede probar que $\log(P)$ es $O(\sqrt{P})$

2.3. Experimentación

Al momento de la experimentación inicialmente teníamos la idea de que, al no tener mejores ni peores casos a nivel de complejidad teórica, no importaba mucho que valores de P que se ingresaban, e iba a ser siempre creciente.

2.3.1. Análisis complejidad teórica

Para el análisis de la complejidad teórica decidimos simplemente medir los tiempos del algoritmo aumentando el valor de entrada desde 1 hasta la cota puesta por el enunciado. Después de observar los resultados, encontramos una función $O(\log(n))$ que se asemeje aproximadamente a los resultados de las mediciones del algoritmo para poder compararlo. Terminamos usando la función $500 * \log_3(x)$. Notamos que no quedaba muy claro cómo es que las mediciones se asemejaban a $\log_3(x)$. Entonces decidimos cambiar la escala: En vez de presentar un gráfico 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))

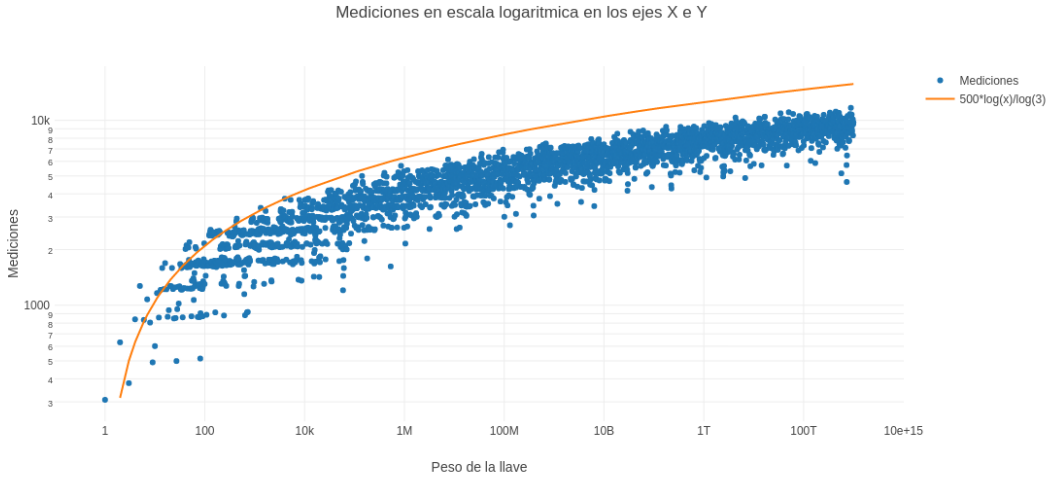


Figura 1: Mediciones de la implementación comparado con $500 * \log_3 n$ en escala logaritmica en X e Y

2.3.2. Análisis

Luego realizado el gráfico notamos que, a pesar de de que las mediciones simulaban cumplir las condiciones de la complejidad, esperábamos un comportamiento más cercano a que vaya aumentando en rectas horizontales, ya que se hacía la misma cantidad de iteraciones para cada valor del logaritmo en base 3. Pero en vez de eso éstas tenían un tipo de patrón ligeramente distinto.

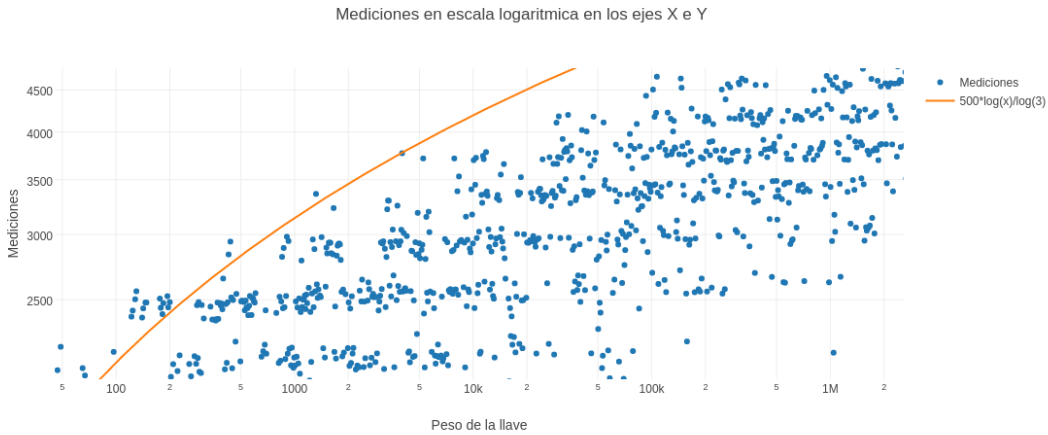


Figura 2: Mediciones de la implementación comparado con $500 * \log_3 n$ zoom en escala logaritmica en X e Y

Si bien se organizaban como líneas rectas (a pesar de la escala), no se acomodaba su "orden" respecto al logaritmo en base 3. Concluimos que esto se debe a la operación "meter la pesa en una balanza", que en la implementación del problema se traduce a introducir un entero al final de una lista, lo cual termina costando $\mathcal{O}(1)$

Para ver si esto era cierto, hicimos nuevas mediciones, tomando para los pesos solo 3 casos:

- Potencias de 3: Que en la representacion ternaria, contaría solo con un 1 y puros ceros
- Potencias de 3 menos 1: Que en la representacion ternaria, contaría con todos 2

-Suma de potencias de 3: Que en la representación ternaria, contaría con todos 1

y como resultado del experimento nos quedó este gráfico

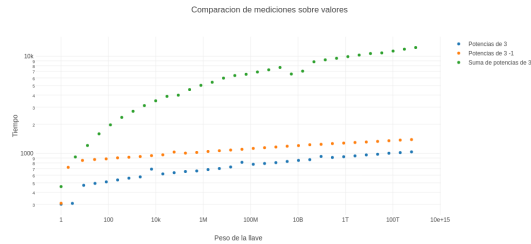


Figura 3: Comparaciones de la implementación comparando los casos en escala logarítmica en X e Y

En la imagen se puede notar que el caso de suma de potencias de 3 toma mucho más que los otros dos casos. Concluimos que esto se debe inicialmente a que, en el caso de sólo potencias de 3, se hace un sólo acceso a la lista, mientras que el caso de potencias de 3 menos 1 tal como sería el resultado al problema, se pone una pesa de la potencia de 3 del lado de la llave y una pesa de tamaño 1 en el otro, teniendo así sólo 2 accesos a la lista, cosa que en la implementación esto se representa sumándole 1 luego de cada división por 3 al peso de la llave en cada iteración.

Como conclusión 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 óptima de llenar las mochilas para poder llevar el mayor valor posible.

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 creímos que este problema podría resolverse 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 superior. Finalmente, pudimos resolverlo mediante programación dinámica. Nuestro problema se divide en dos subproblemas: Encontrar el valor óptimo, y luego llenar las mochilas con los objetos correspondientes.

Para obtener el valor óptimo, modelamos nuestro problema a partir de la siguiente función recursiva:

$$f(0, p_1, p_2, p_3) = 0$$

$$f(n, p_1, p_2, p_3) = \text{Max}(V_n + \text{Max}(f(n-1, p_1 - p_n, p_2, p_3), f(n-1, p_1, p_2 - p_n, p_3), f(n-1, p_1, p_2, p_3 - p_n)), f(n-1, p_1, p_2, p_3))$$

Donde n representa el número 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.

Esta función, al ser recursiva, para calcular su i -ésimo término requiere de los anteriores. En cada estado decide recursivamente si el mejor valor posible se obtiene de sólo guardar nuestro i -ésimo objeto o en alguna de las tres mochilas, o en ninguna mochila.

Para resolver nuestro primer problema el algoritmo se encarga de llenar un arreglo de cuatro dimensiones, donde el tamaño del arreglo principal es la cantidad de objetos y los tamaños de los otros tres arreglos anidados son las capacidades de las tres mochilas. Lo que representa cada posición, (x, y, z, w) , de este arreglo es el valor óptimo que pueden lograr los primeros x objetos con una combinación de mochilas de peso y, z, w . En el caso que el problema a resolver responda a dos mochilas, una de las variables y, z, w serán 0 para cualquier posición y un caso similar si solo hay que llenar una mochila.

Luego para llenar las mochilas utilizamos este arreglo (lleno luego de encontrar el valor óptimo) entonces para cada objeto i guardamos el valor óptimo es decir, la posición del arreglo $(i, \text{cap1}, \text{cap2}, \text{cap3})$.

Después para decidir en cuál mochila tendría que ser guardado revisamos la posición que corresponde al objeto anterior $(i-1)$ y para cada mochila, las otras tres coordenadas serán, la capacidad de la mochila que estamos revisando menos el peso del objeto i , y las otras dos se mantienen constantes. Si esto da cero, es porque encontramos la mochila a la cual corresponde el objeto i , si no ese objeto no debe ser metido en ninguna mochila.

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

3.2.1. Pseudocódigo

guardandoTesoro(mochilas: vector<mochila>, cofre: vector<tesoro>)

- 1: objXpesos \leftarrow 4-upla donde con todos los valores son -1 excepto en las posiciones donde no puede haber objetos que valen 0
 - 2: sol \leftarrow ValorOptimo(objXpesos, cofre, cofre.size-1, capacidades de las mochilas)
 - 3: LlenarMochilas(objXpesos, cofre, mochila1, mochila2, mochila3)
 - 4: return (sol, mochilas)
-

```
LlenarMochilas(objetoXPeso: hipercubo, cofre:vector<tesoro>, m1,m2,m3:mochilas)
1: desde i= cofre.size-1 hasta i=0
2:   obj ← cofre[i]
3:   cap1 ← m1.Capacidad
4:   cap2 ← m2.Capacidad
5:   cap3 ← m3.Capacidad
6:   Si  $i = 0$  hacer
7:     Agregar obj en cualquier mochila en la que entre.
8:   Si no hacer
9:     valM1 ← obj.valor + ValorOptimo(objetoXPeso, cofre, i, cap1-obj.peso, cap2, cap3 )
10:    valM2 ← obj.valor + ValorOptimo(objetoXPeso, cofre, i, cap1, cap2-obj.peso, cap3 )
11:    valM3 ← obj.valor + ValorOptimo(objetoXPeso, cofre, i, cap1, cap2, cap3-obj.peso )
12:    MeterEnCorrecta(valM1, valM2, valM3, m1, m2, m3, obj)
13:  finSi
```

ValorOptimo(objXpeso:4-upla, cofre:vector<tesoro>, objeto:int, peso1: int, peso2:int, peso3:int)

```
1: pesoObj  $\leftarrow$  cofre[objeto].peso
2: pesoVal  $\leftarrow$  cofre[objeto].valor
3: Si  $peso1 < 0 \vee peso2 < 0 \vee peso3 < 0$ 
4:   devolver -1
5: finSi
6: Si objetoxPesos[objeto][peso1][peso2][peso3]  $\neq$  -1
7:   devolver objetoxPesos[objeto][peso1][peso2][peso3]
8: finSi
9: Si objeto = 0
10:   val  $\leftarrow$  0
11:   Si  $peso1 \geq pesoObj \vee peso2 \geq pesoObj \vee peso3 \geq pesoObj$ 
12:     val  $\leftarrow$  valorObj
13:     objetoxPesos[objeto][peso1][peso2][peso3]  $\leftarrow$  val
14:     devolver val
15:   Si no
16:     devolver -1
17:   finSi
18: Si no
19:   PosiblesSolus  $\leftarrow$  vector<int>
20:   sinObj  $\leftarrow$  ValorOptimo(objetoxPesos, cofre, objeto -1, peso1, peso2, peso3)
21:   PosiblesSolus.Agregar(sinObj)
22:   Desde j=1 hasta j=3 hacer
23:     Si  $pesoj - pesoObj \geq 0$ 
24:       objenj = valorObj + ValorOptimo(objetoxPesos, cofre, objeto - 1, pesoj - pesoObj, los
        otros dos pesos)
25:       PosiblesSolus.Agrregar(objenj)
26:     finSi
27:   finDesde
28:   valor  $\leftarrow$  Max(PosiblesSolus)
29:   objetoxPesos[objeto][peso1][peso2][peso3]  $\leftarrow$  valor
30:   return valor
31:
```

3.2.2. Demostración de Correctitud

Presentaremos la función matemática que modela nuestro problema:

$$f(0, p_1, p_2, p_3) = 0$$
$$f(n, p_1, p_2, p_3) = \text{Max}(V_n + \text{Max}(f(n-1, p_1 - p_n, p_2, p_3), f(n-1, p_1, p_2 - p_n, p_3), f(n-1, p_1, p_2, p_3 - p_n)), f(n-1, p_1, p_2, p_3))$$

Donde n representa el número 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 valor posible.

Hipótesis 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.

$$f(n, p_1, p_2, p_3) = \text{Max}(V_n + \text{Max}(f(n-1, p_1 - p_n, p_2, p_3), f(n-1, p_1, p_2 - p_n, p_3), f(n-1, p_1, p_2, p_3 - p_n)), f(n-1, p_1, p_2, p_3))$$

por hipótesis inductiva (como $n-1 \leq k$ para algún 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 óptimos 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:

$$f(n, p_1, p_2, p_3) = \text{Max}(V_n + \text{Max}(V_{01}, V_{02}, V_{03}), V_{00})$$

Podemos ver que esta función compara el valor óptimo de llenar las mochilas con $n-1$ tesoros y el tesoro n , con el valor óptimo 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 obtiene de poner algún objeto de los $n-1$ anteriores que impide luego meter el tesoro n).

Como la función Max devuelve el mayor valor, el resultado será el óptimo.

Como valen $P(0) \dots p(k) \forall k < n$ y vale $p(n)$ vale $p(n) \forall n \in N \cup 0$

En nuestra demostración utilizamos el principio de optimalidad. Ahora probaremos que efectivamente queremos construir nuestra solución a partir de subsoluciones óptimas:

Supongamos que $f(n, p_1, p_2, p_3) = \text{Max}(V_n + H, V_{00})$ donde H es una subsolución del estado anterior que no es óptima. Sabemos que $H < \text{Max}(V_{01}, V_{02}, V_{03})$ (ya que si no sería óptima) entonces $V_n + H < V_n + \text{Max}(V_{01}, V_{02}, V_{03})$ (ya que estamos sumando números positivos) Luego $\text{Max}(V_n + H, V_{00}) \leq \text{Max}(V_n + \text{Max}(V_{01}, V_{02}, V_{03}), V_{00})$ entonces $f(n, p_1, p_2, p_3)$ no es óptimo, absurdo.

3.2.3. Demostración de Complejidad

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. Además, esta complejidad es aportada por el algoritmo ValorOptimo, entonces basaremos nuestra demostración en el mismo.

Primero haremos unas observaciones preliminares:

- Se puede pensar que el arreglo de cuatro dimensiones en realidad es un arreglo de cubos, donde

el índice de cada cubo es hasta que objeto se verifíco la solución optima.

- El volumen de un cubo 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. En este caso las aristas de los cubos serán las capacidades de las mochilas, sabemos que llenar una posición del cubo nos cuesta $\Theta(1)$, entonces llenarlo entero nos costará el equivalente al volumen del mismo.
- Llenar un hipercubo, o arreglo de cubos, con un valor determinado (como ocurre en la línea 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 $\Theta(O(T))$ ya que en las líneas 9,10,11 cuando se llama a `ValorOptimo`, el hipercubo `objetoXpeso` cuenta con todos los valores ya calculados entrando en el `if` (línea 6) que devuelve en $\Theta(1)$ el valor.

Volviendo a la demostración, tanto en el pseudocódigo como en la función matemática podemos ver que la recursión se realiza a lo sumo cuatro veces la cantidad total de tesoros, pero estas recursiones se hacen sobre valores distintos, posiciones distintas del arreglo y referenciamos al hipercubo. Para llenar un cubo, siempre debemos recurrir al cubo anterior, uno puede pensar que esto aportaría complejidad, sin embargo, al guardar estos valores sólo construimos estos cubos una vez, y la complejidad por acceso es $\Theta(1)$.

como se guardan los valores y en ninguna recursividad el algoritmo recalcula un valor, la complejidad está dada por llenar el arreglo de cubos. Concluyendo que la complejidad es $\mathcal{O}(\prod_{i=1}^3 K_i * T)$

Ahora probemos que esta complejidad es menor a la pedida ($\mathcal{O}((\sum_{i=1}^3 K_i)^3 * T)$)
quiero ver que $\mathcal{O}(\prod_{i=1}^3 K_i * T) \leq \mathcal{O}((\sum_{i=1}^3 K_i)^3 * T)$
si T es cero la desigualdad es verdadera, sino es equivalente probar que $\mathcal{O}(\prod_{i=1}^3 K_i) \leq \mathcal{O}((\sum_{i=1}^3 K_i)^3)$
Si expandimos la suma y la productoria nos queda :
 $K_1 * K_2 * K_3 \leq K_1^3 + K_2^3 + K_3^3 + 3 * K_1^2 * K_2 + 3 * K_1^2 * K_3 + 3 * K_2^2 * K_1 + 3 * K_2^2 * K_3 + 3 * K_3^2 * K_1 + 3 * K_3^2 * K_2 + 6 * K_1 * K_2 * K_3$.
 $0 \leq K_1^3 + K_2^3 + K_3^3 + 3 * K_1^2 * K_2 + 3 * K_1^2 * K_3 + 3 * K_2^2 * K_1 + 3 * K_2^2 * K_3 + 3 * K_3^2 * K_1 + 3 * K_3^2 * K_2 + 6 * K_1 * K_2 * K_3$.
como $K_i \geq 0$ para todo i , eso se cumple y demostramos que la complejidad es menor a la pedida.

3.3. Experimentación

La cota de complejidad de nuestro algoritmo es $\prod_{i=1}^3 K_i * T$. Es decir que depende de la capacidad y cantidad de mochilas y la cantidad de tesoros. En esta sección trataremos de respaldar esta cota mediante el análisis de los datos empíricos que obtuvimos a través del testeo de nuestro algoritmo. Con este objetivo a lo largo de los tests modificamos los inputs para observar de una variable por vez dejando las otras constantes y así poder analizar el tiempo de ejecución en cada caso. En cada test los valores se logran al promediar tres mil iteraciones sobre el mismo input, sobre que forma tiene el input se va a hablar más adelante.

3.3.1. Resultados y análisis

En nuestro primer experimento fijamos una mochila con capacidad constante (50) y fuimos aumentando la cantidad de tesoros de a dos. Así mismo, analizamos tres subcasos pertinentes: cuando los objetos están dados aleatoriamente (sin ninguna restricción sobre su peso), cuando el peso de los objetos se encuentra restringido a la capacidad de la mochila y cuando el peso de objetos es superior a la capacidad de la mochila.

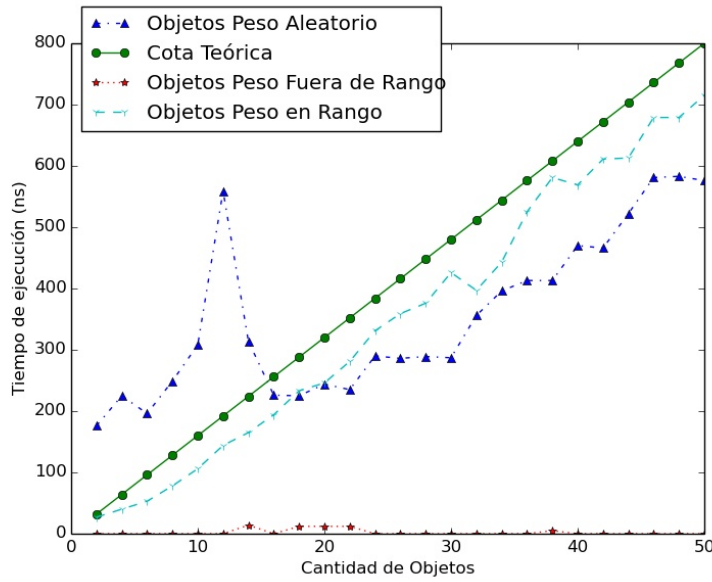


Figura 4: gráfico comparativo de los tres subcasos al variar T.

En esta figura podemos observar no solo la tendencia lineal en la complejidad del algoritmo sino también la variación del crecimiento del tiempo para los distintos casos. El hecho de que los gráficos sean lineales respaldan nuestra cota de complejidad ya que estaríamos variando el parámetro T mientras K_1 (la capacidad de la mochila) se mantiene constante. Estaríamos bajo la presencia de una función de tipo $Y = c * X$ con c constante. 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 y el tiempo de ejecución es casi nulo. Podemos concluir que este sería nuestro mejor caso, y se debe al hecho de que filtramos nuestra entrada para que no se tengan en cuenta estos tesoros. También se puede observar una sección donde los objetos con peso aleatorio tienen un tiempo de ejecución que los otros casos, esto bien puede resultar de los valores de los pesos. Si los pesos son menores en el caso de los pesos aleatorios a los pesos en rango, la cantidad de recursiones a las que se llama, son mayores. Observemos que en general lleva más tiempo obtener un resultado cuando los objetos tienen peso menor a la capacidad de la mochila.

Luego realizamos tests para poder ver el comportamiento del algoritmo al ir aumentando el peso de la mochila. Utilizamos un test para evaluar el comportamiento cuando los objetos son aleatorios y otro para cuando todos los tesoros tienen un peso mayor a la capacidad de la mochila, en ambos casos la cantidad de objetos es constante en 50 elementos. 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.

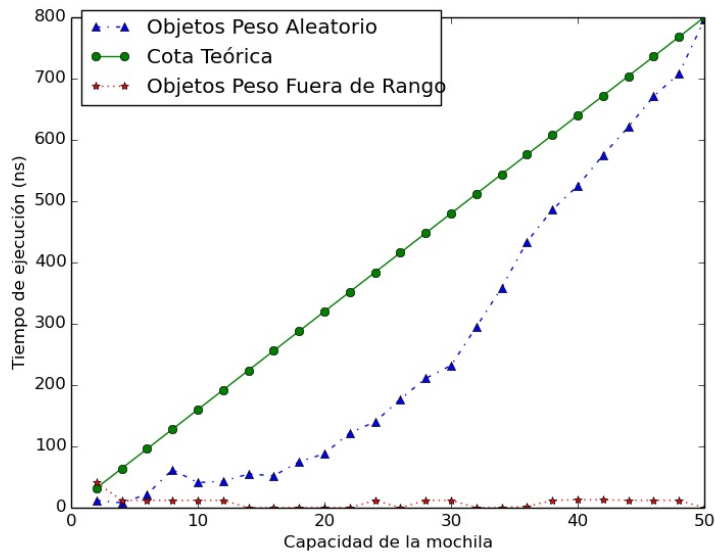


Figura 5: gráfico comparativo al variar K

Podemos observar que al tener los tesoros con peso fuera del rango de la capacidad de la mochila el tiempo se mantiene constante, prácticamente nulo igual que en el experimento anterior. Con los objetos aleatorios vemos que tiene cierta tendencia lineal como es de esperarse (ya que este caso es similar al analizando en la figura()) sin embargo algunos valores quedan distorcionados. Creemos que esto se debe a que los objetos son aleatorios.

Finalmente corrimos tests variando la cantidad de mochilas (todas con capacidad 50) manteniendo constantes los tesoros, estos siendo 50. Este gráfico va a tener escala logarítmica en el eje y para poder observar su comportamiento exponencial.

En el gráfico se trata de comparar como afecta la distribución de pesos en las mochilas. Como nuestra complejidad varía dependiendo del producto de las capacidades queríamos ver como era el tiempo de ejecución para distinta cantidad de mochilas con el mismo producto. tomamos cincuenta objetos con peso igual a la capacidad mínima del conjunto de mochilas en el que testamos y repetimos cada instancia mil veces tomando el promedio.

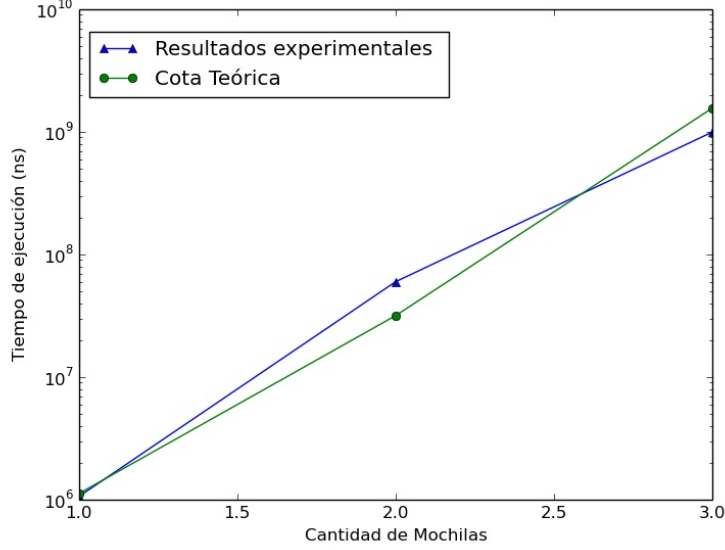


Figura 6: variación de la cantidad de mochilas.

En esta figura podemos observar el crecimiento exponencial del tiempo dependiendo de la cantidad de mochilas, como la escala es logarítmica y las curvas son casi líneas rectas entonces podemos deducir que el tiempo crece de forma exponencial. Esto concuerda con el hecho de que al tener todas las variables en 50 ($T = 50$, $K_1 = 50$, $k_2 = 50$, $k_3 = 50$) al ir aumentando la cantidad de mochilas estaríamos elevando la constante 50 (con K_1 tenemos $K_1 * T = 50^2$, con K_2 tenemos $K_1 * K_2 * T = 50^3$, etc). Este es el tipo de función exponencial $Y = 50^X$.

En el último gráfico se trata de comparar como afecta la distribución de pesos en las mochilas. Como nuestra complejidad varía dependiendo del producto de las capacidades queríamos ver como era el tiempo de ejecución para distinta cantidad de mochilas con el mismo producto. tomamos cincuenta objetos con peso igual a la capacidad mínima del conjunto de mochilas en el que testeamos y repetimos cada instancia mil veces tomando el promedio.

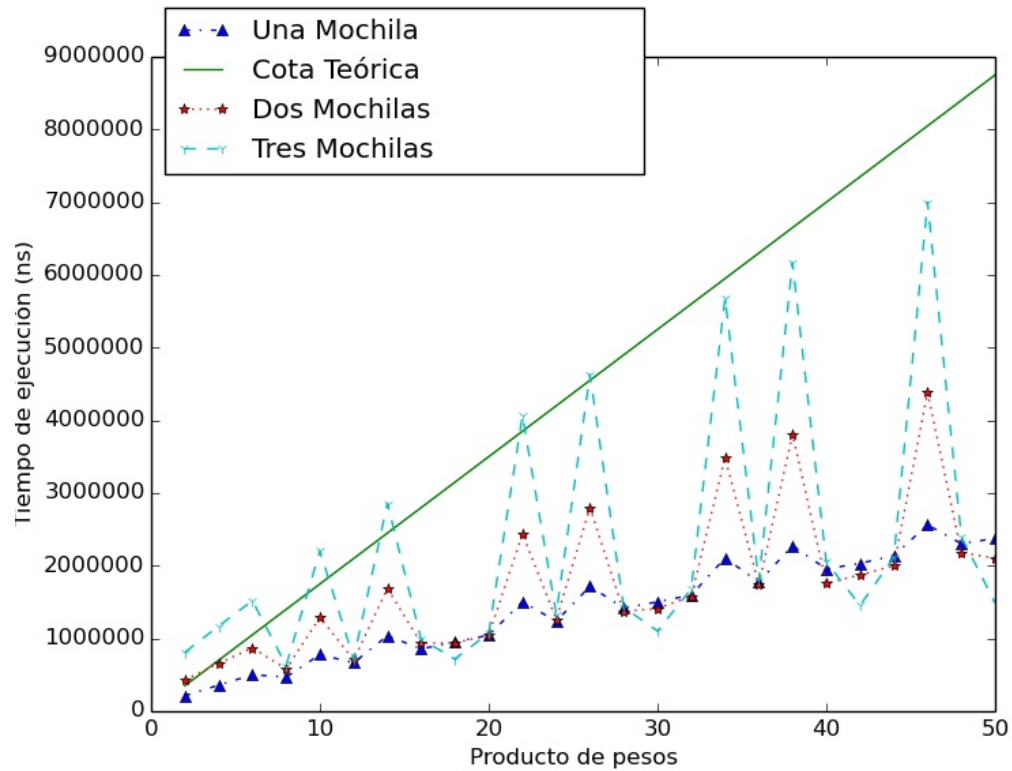


Figura 7: variación de la Productaria de Pesos.

En el gráfico separaremos las mediciones en 3 partes, una los picos altos, desde el seis y cada cuatro números con la excepción de cuatro casos. Dos los picos bajos desde el ocho y cada cuatro números y tres, las anomalías que rompen el patrón de los picos altos. Debajo pondremos cuales son las capacidades, en cada tabla especificamos si ese peso responde a Picos Altos, Bajos o anomalías.

Picos Altos	Dos mochilas	Tres mochilas	Picos Altos	Dos mochilas	Tres mochilas
2	1*2	1*1*2	8	2*4	2*2*2
4	2*2	1*2*2	12	3*4	2*2*3
6	2*3	1*2*3	16	4*4	2*2*4
10	2*5	1*2*5	20	4*5	2*2*5
14	2*7	1*2*7	24	4*6	2*2*6
22	2*11	1*2*11	28	4*7	2*2*7
26	2*13	1*2*13	32	4*8	2*2*8
34	2*17	1*2*17	36	6*6	2*2*9
38	2*19	1*2*19	40	8*5	2*2*10
46	2*23	1*2*23	44	11*4	2*2*11
-	-	-	48	8*6	2*2*12

Anomalías	Dos mochilas	Tres mochilas
18	$3*6$	$2*3*3$
30	$5*6$	$2*3*5$
42	$6*7$	$2*3*7$
50	$5*10$	$2*5*5$

En los picos altos la cantidad de mochilas importa, en el mismo producto de pesos 3 mochilas tardan más que 2 y las dos tardan más que 1 sola. Lo cual por la complejidad teórica resulta incoherente, aún así, esto se puede explicar por como creamos las pruebas. En los picos altos los pesos de los objetos es uno, esto quiere decir que no solo todos los objetos entran, sino que los pesos de las mochilas se reducen de a uno, lo cual genera mayores llamadas a las recursiones. si es una sola mochila hace dos llamadas, si son tres hace cuatro y esto depende directamente de los pesos, si los pesos fueran mas grande, algunas llamadas recusivas no las haría por que se irían de rango.

En cambio en los picos bajos se puede ver como se reducen los pesos de una forma más rápida ya que los pesos son dos, haciendo menos llamadas recursivas.

Ahora nos falta hablar de los cuatro elementos que no solo rompen con el patrón de los picos altos, sino que tambien a diferencia de los picos bajos hace que los tiempo de ejecución de tres mochilas sean menos que el de dos o el de una. Esto se explica tambien con los pesos que se eligieron para llenar las mochilas, sigue siendo dos, pero en este caso entra menos veces a las llamadas en compuesto por tres ya que los pesos de cada mochila es menor.

En conclusión podemos decir que los tiempos de ejecución, si decidimos que los objetos tengan como peso mínimo la capacidad mínima del conjunto de mochilas, tardan menos que cuanto más homogénea sea la descomposición, hasta favoreciendo el caso de más mochilas, y cuanto más desbalanceada sea la descomposición más va a tardar, favoreciendo el caso de menos mochilas. Entonces podemos decir que tenemos números que sus productos son más valiosos en tiempo de ejecución que otros, por ejemplo sabemos que los primos van a tardar notablemente más si son más mochilas y tambien podemos suponer con cierto grado de certeza que los cubos perfectos son más eficientes en tiempo si decidimos dividirlo en tres mochilas.

:-"Y todos estos tesoros van a ir para algun museo no?"

:-"Sí,Indi... lo que digas..."