DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº 1

Indiana Jones en búsqueda de la complejidad esperada

Segundo cuatrimestre de 2016 — Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Langberg, Andrés	249/14	andreslangberg@gmail.com
Luce, Nicolás	294/14	nicolas.p.luce@gmail.com
Rodriguez, Lucas	593/14	rodriguez.lucas.e@gmail.com
Walter, Nicolás	272/14	nicowalter25@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Problema 1: Cruzando el puente

1.1. Introducción

En uno de sus viajes en el medio oriente, Indy y su grupo de arqueólogos recorren el bosque con la ayuda de miembros de una tribu caníbal (donde la cantidad total del contingente no supera a 6 personas).

En el medio del trayecto, se encuentran con un puente colgante cuya estructura dudosa no permite que crucen más de dos personas al mismo tiempo. Al contar con una sola linterna, siempre que quede por lo menos una persona del grupo en el punto de partida, alguien que se encuentre del otro lado debe volver con la linterna a buscarlo. Además, dado que parte del grupo practica el canibalismo, los arqueólogos deciden de manera unánime establecer la restricción de que en ningún momento puede haber mayoría de caníbales en alguno de los lados del puente.

El tiempo apremia, por lo que se busca que el grupo cruce el puente de la manera mas rápida posible. Cada individuo tiene su propia velocidad, y y si dos personas cruzan el puente lo cruzarán a la velocidad del más lento.

En cuanto a un planteo abstracto del problema, el grupo esta conformado con N arqueólogos y M caníbales, donde podemos identificar a los integrantes del grupo por su tipo y velocidad (expresada como un entero entre 1 y 10^6).

El objetivo es cruzar a todos del lado A al lado B del puente (representados con conjuntos) en el menor tiempo posible, teniendo en cuenta las restricciones previamente descritas.

Definimos como movimientos, a las posibles manera de cruzar el puente. Por ejemplo, 'AC' corresponde a un arqueólogo y un caníbal cruzando el puente.

1.2. Desarrollo

Los individuos son representados con el tipo `unsigned int` ya que el rango de éste alcanza para abarcar las cotas requeridas por el problema. En principio no hay manera de diferenciar a los individuos por tipo. Para especificar ésta característica se utilizaron distintos vectores que representan el conjunto de arqueólogos o caníbales de cada lado del puente. Es decir, tenemos las siguiente estructuras para la representación de cada individuo:

- **indysA**: El conjunto de arqueólogos de lado A del puente representado por un `vector<unsigned int>`.
- **indysB**: El conjunto de arqueólogos de lado B del puente representado por un `vector<unsigned int>`.
- **canibA**: El conjunto de caníbales de lado A del puente representado por un `vector<unsigned int>`.
- **canibB**: El conjunto de caníbales de lado B del puente representado por un `vector<unsigned int>`.

El acercamiento utilizado para la resolución del problema es un algoritmo del tipo backtracking, en otras palabras, el objetivo es generar todos los estados posibles a partir de uno inicial, filtrando aquellos que rompen el invariante de representación estipulado por el enunciado del problema.

Cada estado es representado por un nodo dentro del árbol generado por el algoritmo, siendo la solución final un camino particular de ésta estructura. Cada nodo contiene un vector de hijos para cada posible movimiento que permita el estado, siendo los elementos de éste las distintas combinaciones de arqueólogos y/o caníbales posibles, sumado a la velocidad acumulada hasta ese punto.

La construcción de la estructura se realiza de forma recursiva mediante dos funciones, `ida()` y `vuelta()`. Se escribe solo `ida()` ya que la lógica de `vuelta()` es similar.

Algoritmo 1 ida

Input:
Output:

```
if estado invalido then
    nodo.valido ← false
    return nodo
end if
if estado ganador then
    nodo.valido ← true
    return nodo
end if
velocidad mínima ← ∞
if #indysA ≥ 2 then
    for each combinación de arqueólogos do
        iA ← indysA − {A_i, A_j}, iB ← indysB + {A_i, A_j}
        nueva velocidad ← velocidad acumulada + max(A_i, A_j)
        res ← vuelta(iA, iB, canibA, canibB, nueva velocidad)
        if res.valido then
            nodo.AA • res
            velocidad mínima ← min(velocidad mínima, res.velocidad)
        end if
    end for
end if
if #canibA ≥ 2 then
    //misma idea que el anterior...
end if

if #indysA ≥ 1 and #canibA ≥ 1 and movimiento anterior ≠ AC then
    for each combinación de arqueólogos y caníbales do
        iA ← indysA − {A_i}, iB ← indysB + {A_i}
        cA ← canibA − {C_j}, cB ← canibB + {C_j}
        nueva velocidad ← velocidad acumulada + max(A_i, C_j)
        res ← vuelta(iA, iB, cA, cB, nueva velocidad)
        if res.valido then
            nodo.AC • res
            velocidad mínima ← min(velocidad mínima, res.velocidad)
        end if
    end for
end if

if #hijos(nodo) = 0 then
    nodo.valido ← false
    return nodo
else
    nodo.valido ← true
    nodo.velocidad ← velocidad mínima
    return nodo
end if
```

Estas toman como parámetros los cuatro vectores mencionados anteriormente y la velocidad del estado anterior (la suma del tiempo utilizado por los individuos al cruzar el puente).

1.3. Correctitud

A modo de demostrar que el algoritmo propuesto efectivamente brinda soluciones óptimas, se enlistarán una serie de proposiciones junto con sus respectivas demostraciones que ayudarán a realizar esta tarea.
Primero definimos la siguiente función partida.

$$f(x) = \begin{cases} -2 & \text{si x es par} \\ 1 & \text{si x es impar} \end{cases}$$

Luego, sea S la siguiente sucesión,

$$S(K) = K + \sum_{i=0}^{2K-4} f(i), \forall K \in \mathbb{N}/K \geq 2$$

Realizando un breve análisis, esta sucesión es igual a la constante 0. Una vez definida la sucesión, podemos demostrar las proposiciones.

Proposición 1.1 *Dada una entrada E válida, el algoritmo evaluado en E termina.*

Como se explicó anteriormente los movimientos considerados al cruzar el puente del lado A al lado B son de dos individuos y en el caso contrario son de 1 individuo. Por lo tanto, podemos remarcar que en los nodos de nivel par se realizaran movimientos que involucren dos individuos y en los nodos de nivel impar se realizaran movimientos que involucren un individuo.

Sea $K(N_i)$ la cantidad de individuos en el lado A en un nodo de nivel i, podemos decir que $K(N_{i+1}) = K(N_i) - 2$, si i es par y $K(N_{i+1}) = K(N_i) + 1$ si i es impar. En el algoritmo se busca que $K = 0$ y realizando movimientos de esta manera, esto se logrará en $(2K-4)+1$ pasos como puede ser visto en un análisis de la sucesión mencionada anteriormente.

De no encontrar una solución al problema de cruzar el puente, como por ejemplo el caso $N = 4$ y $M = 4$, todos los nodos del árbol serán no válidos y por lo tanto serán filtrados por el nodo padre y este a su vez por el suyo, dejando únicamente la raíz con una velocidad asociada de -1. Otro punto a demostrar es que el algoritmo no genera ciclos, que será demostrado más adelante.

Proposición 1.2 *Si $K(N_i) \geq 2$ con i impar, realizar movimientos de dos individuos siempre resultarán en una solución no óptima.*

Sea N_i con i impar, un nodo de nivel i y $K(N_i) \geq 2$. Los movimientos disponibles son AA o CC o AC, y A y/o C. Supongo que al realizar cualquiera de estos movimientos el estado resultante es válido. Luego, existen dos posibles hijos de este nodo, haciendo un movimiento de dos y otro de uno. Tomo por ejemplo AC y A, donde vel(A) = vel(C).

- Movimiento AC: $K(N_{i+1}) = K(N_i) + 2$
- Movimiento A: $K(N_{i+1}) = K(N_i) + 1$

Luego, la velocidad utilizada en ambos casos es la misma. Como fue mencionado anteriormente la cantidad de movimientos restantes a realizar es

$$S(K) = K + \sum_{i=0}^{2K-4} f(i), \forall K \in \mathbb{N}/K \geq 2$$

Y dado que, $K(N_i) + 2 < K(N_i) + 1$ se necesitará una mayor cantidad de movimientos para finalizar el algoritmo. Como las velocidades deben ser positivas y un camino del árbol tiene al menos dos niveles más que el otro, resulta trivial que aquel camino de mayor altura tendrá una solución mayor con respecto al otro. Análogamente se puede demostrar que si $K(N_i) \geq 2$ con i par, realizar movimientos de un individuo siempre resultarán en una solución no óptima.

Existe un escenario donde no es posible realizar ningún movimiento del lado B al lado A de un individuo ya que generan estados inválidos. Este es el siguiente,

Lado A	**Lado B**
$N_A = M_A$	$N_B = M_B$

Luego, $2 \leq N_A + M_A \leq 4 \leq N_B + M_B$, queda claro que el único movimiento posible realizable es AC, si $N_B = 2$ también es posible AA pero no existen soluciones luego de este movimiento. En el caso donde $N_A + M_A = 2$ y $N_B + M_B = 4$ al realizar el movimiento AC del lado B al A, el resultante es el estado espejado donde la única solución surge al tener la posibilidad de mover los dos arqueólogos del lado A al B, situación que no es posible con valores mayores y por esto no poseen solución.

El algoritmo identifica el escenario donde ningún movimiento de un individuo es válido y realiza un movimiento AC de ser posible, notificando a la función siguiente para que ésta no realice la misma acción ya que generaría un ciclo entre estados.

Proposición 1.3 *Dada una entrada E válida, el algoritmo evaluado en E genera todos los estados válidos óptimos.*

Sea N_i un nodo de nivel i con $K(N_i) \geq 2$, separo en dos casos:

- $i \equiv 0 \bmod(2)$. En este nodo sabemos que los movimientos disponibles son AA, CC y AC. Los movimientos A y C no son considerados ya que nunca proporcionaran soluciones óptimas, como se demostró en la proposición 2.
  Luego, si al realizar alguno de estos movimientos se detecta un estado invalido, el algoritmo no lo considerará como un estado hijo.

- $i \equiv 1 \bmod(2)$. En este nodo sabemos que los movimientos disponibles son A, C y AC si ninguno de los anteriores brinda una solución.
  Luego, si al realizar alguno de estos movimientos se detecta un estado invalido, el algoritmo no lo considerará como un estado hijo.

Proposición 1.4 *Sea N_i y N_j un nodo de nivel i, j tal que $i \leq j$ y N_j pertenece a una rama de N_i. $N_i \neq N_j$, es decir, que dado un estado no puedo llegar de nuevo a si mismo.*

Como fue demostrado antes, $K(N_i) \neq K(N_j), \forall(i, j \in \mathbb{N})i\neq j$ y por lo tanto nunca podrán ser iguales.

1.4. Complejidad del algoritmo

Como se mencionó anteriormente, para obtener el resultado esperado el algoritmo construye un árbol con los posibles estados del problema. En particular, cada nodo de nivel par tendrá a lo sumo $\binom{N}{2} + \binom{M}{2} + N \cdot M$ hijos y aquellos de nivel impar tendrán a lo sumo $N + M + N \cdot M$ hijos.

La altura de este árbol en el peor caso será igual a la cantidad de posibles estados. Sea K el cardinal del conjunto de individuos, ignorando las podas realizadas por el algoritmo, la cantidad de estados posibles es igual al cardinal del conjunto de partes de este conjunto. Es

decir 2^K, dado que los estados pueden diferenciarse por el lado en el que está la linterna se pueden obtener el doble de estados. En conclusión, la altura del árbol es acotado por 2^{K+1}.

Por último, la cantidad de operaciones que cada nodo realiza puede ser acotada por $5N + 5M$ operaciones en el peor caso. Con ésta última consideración podemos concluir que la complejidad del algoritmo en el peor caso es igual a:

$$\mathcal{O}((5N + 5M) \cdot (\tbinom{N}{2} + \tbinom{M}{2} + N \cdot M)^{2^{N+M+1}}).$$

De todas maneras, esto puede simplificarse considerablemente debido que el peor caso es aquel donde se realicen la menor cantidad de podas. Esto sucede cuando todos los individuos son del mismo tipo, arqueólogos o caníbales, ya que todo movimiento que se haga resultará en una posible solución al problema de cruzar el puente. Por lo tanto, la complejidad resultante es:

$$\mathcal{O}(5N \cdot \tbinom{N}{2}^{2^{N+1}}) = \mathcal{O}(N \cdot \tbinom{N}{2}^{2^{N+1}}) \ .$$

Análogamente, si todos los individuos son caníbales, la complejidad es: $\mathcal{O}(M \cdot \tbinom{M}{2}^{2^{M+1}})$

1.5. Experimentación

Una vez obtenida la complejidad teórica correspondiente al algoritmo propuesto para este problema, el siguiente paso es observar los comportamientos reales del mismo frente a los distintos escenarios sugeridos. Recordemos,

"[...] el grupo esta conformado por N arqueólogos y M caníbales [...]"

Sea $V_i, W_j \in \mathbb{N}$, tal que la velocidad del i-ésimo arqueólogo es igual a V_i y el j-ésimo caníbal es igual a W_j. Luego, las restricciones de entrada son listadas a continuación:

- $1 \leq N + M \leq 6$
- $M \leq N$
- $1 \leq V_i \leq 10^6$
- $1 \leq W_j \leq 10^6$

Dado que el algoritmo genera estados distintos para velocidades repetidas de individuos del mismo tipo y ya que 10^6 es representable por el tipo `unsigned int`, experimentar con los valores de las velocidades no afectaría en absoluto la performance.

La hipótesis que se busca confirmar es, como fue mencionado anteriormente, que el peor caso que el algoritmo debe resolver es aquel donde no se realicen podas en el árbol generado. Dicho escenario surge al no haber caníbales o arqueólogos en la entrada ($M = 0 \vee N = 0$). En estos casos, no existen estados inválidos, y por lo tanto todo movimiento resulta en una posible solución para cruzar el puente.

Las siguientes mediciones temporales son un resultado promedio de 50 repeticiones. Respecto al gráfico se utilizó como eje de ordenadas el tiempo en nanosegundos, y el eje de abscisas la cantidad de arqueólogos, siendo constante para cada función la cantidad de caníbales. [1]

Seguido de los resultados obtenidos podemos notar ciertas particularidades. Para las entradas $M = 0$ y $M = 1$ los tiempos son similares con respecto a la cantidad total de individuos, esto podría ser debido a que aún con un caníbal no es posible generar estados inválidos y por lo tanto no es posible podar el árbol, algo que si es notable en las mediciones de valores mayores de M.

[1] Dado que dejar constante la cantidad de arqueólogos y variar la cantidad de caníbales es análogo, solo graficamos estos resultados.

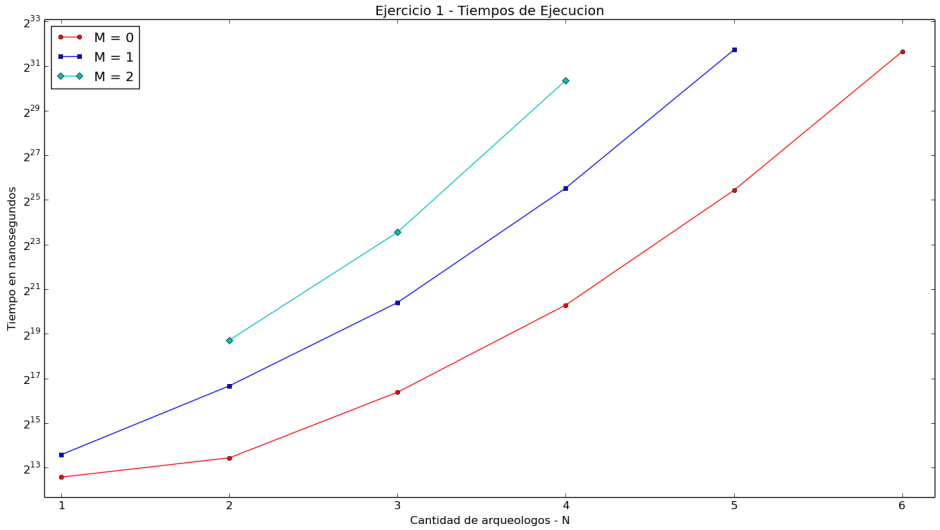

Figura 1: Tiempos de Ejecución

Cantidad de Caníbales	Cantidad de Arqueólogos: 1	2	3	4	5	6
0	6.084,22	11.085,20	85.324,40	1,2817E+06	4,5291E+07	3,3769E+09
1	12.221,30	103.823,00	1,3825E+06	4,8594E+07	3,5644E+09	-
2	-	429.454,00	1,2317E+07	1,3801E+09	-	-
3	-	-	1,1328E+08	-	-	-

Cuadro 1: Tiempos de Ejecución en nanosegundos

A modo de confirmar esta equivalencia, se realizó un segundo experimento, esta vez midiendo la cantidad de nodos generados para cada valor de entrada y de esta manera tener un dato preciso de la cantidad de podas realizadas en los distintos escenarios.

Cantidad de Caníbales	Cantidad de Arqueólogos: 1	2	3	4	5	6
0	2	2	16	271	10231	741046
1	2	16	271	10231	741046	-
2	-	54	1990	273106	-	-
3	-	-	19663	-	-	-

Cuadro 2: Cantidad de Nodos Generados

Como podemos ver, $M = 0$ y $M = 1$ son efectivamente el mismo caso, y en particular el peor caso para $K = 6$, siendo K la cantidad de individuos.

Por último y no por eso menos importante, analicemos la cota teórica propuesta anteriormente con respecto a las mediciones tomadas. Dado que la función mencionada mide cantidad de operaciones y del experimento realizado se obtuvo tiempo de ejecución, se define una constante $c = 100000$ nanoseg. y luego $f(N) = c \cdot (3N + 2N) \cdot \binom{N}{2}^{2^{N+1}}$

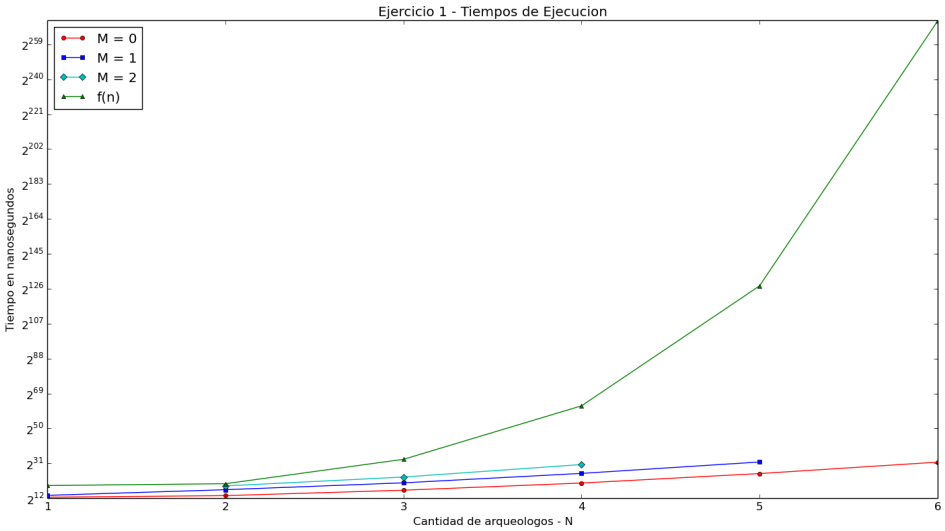

Figura 2: Tiempos de Ejecución y Cota Teórica

Si bien $f(N)$ es una cota superior para el algoritmo, es notable que $2^{(K+1)}$ como la altura del árbol en el peor caso es ampliamente superior al resultado visto en los experimentos realizados (ver Cuadro 3), incluso sin haber podas. Esto se debe a que dos nodos serán adyacentes si y solo si es posible generar uno en base al otro con un movimiento válido.

Cantidad de Caníbales \ Cantidad de Arqueólogos	1	2	3	4	5	6
0	2	2	4	6	8	10
1	2	4	6	8	10	
2	-	6	8	10	-	-
3	-	-	12	-	-	-

Cuadro 3: Altura de los Árboles Generados

2. Problema 2: Problemas en el camino

2.1. Introducción

El equipo logra cruzar el puente y entra a la fortaleza antigua, donde llegan a la puerta de la sala donde están los tesoros. Esta puerta sólo puede ser abierta con la ayuda de una llave, la cual se encuentra en una balanza de dos platillos. Al tomar la llave, la balanza cambia su desequilibrio inicial, lo cual acciona una trampa que no deja salir a nadie de la habitación.

La única escapatoria que tienen, es emular el peso de la llave con la ayuda de unas pesas cuyo valor esta determinado por las distintas potencias de 3, teniendo en cuenta que solo tienen un ejemplar de cada una.

Formalmente, podemos decir que el objetivo del problema es lograr alcanzar el peso de la llave (expresado como un entero positivo menor a 10^{15}), sumando o restando distintas potencias de 3, identificando cuales se colocaron en cada platillo de la balanza.

Se propone resolver este problema con una complejidad temporal de $O(\sqrt{P})$, siendo P el peso de la llave, el cual se pasa como parámetro de la función que lo resuelva.

- Ejemplo:

  Sea la entrada P = 16, la salida del algoritmo propuesto debe retornar en la primer línea de la salida la cantidad de pesas que se colocaron en cada balanza, para luego detallar cuáles se utilizaron de cada lado mediante 2 arreglos que representan ambos platillos.

  La salida correcta en este caso sería:

```
2 2  # pesas en cada platillo
1 27 -> pesas del platillo izquierdo
3 9  -> pesas del platillo derecho
```

  Donde claramente $16 = 1 + 27 - 3 - 9$

2.2. Desarrollo

En primer lugar, se calcula el desarrollo en base 3 del número P mediante la función *toBase3*, la cual retorna en un vector que potencias de 3 es necesario sumar para alcanzar dicho valor, expresada con su dígito correspondiente (entre 0 y 2).

Algoritmo 2 toBase3

Input: $P \in \mathbb{N}$

Output: $res \in \mathbb{Z}^n$, el número P escrito en base 3.

```
while P ≥ 3 do
    res • (P mod 3)
    P ← P div 3
end while
if P < 3 then
    res • P
    return res
end if
```

Dado que por restricción del problema sólo se cuenta con una pesa por potencia, formalmente esto es equivalente a que el dígito de cada potencia sea 0 ó 1. En los casos donde

la misma es 2, podemos expresar esos valores como el resultado de la resta entre la siguiente potencia y actual.

Demostramos esta proposición por inducción:
Sea $P(n) : 3^{n+1} - 3^n = 2 * 3^n$, queremos probar que $P(n)$ es Verdadera para todo n $\in \mathbb{N}$.

- Caso base:
  $P(0) : 3^{0+1} - 3^0 = 2 * 3^0 \Leftrightarrow 2 = 2\surd$

- Paso inductivo:
  Dado h $\in \mathbb{N}$, suponemos Verdadera P(h). Queremos ver si vale P(h+1).

  P(h+1): $3^{h+2} - 3^h = 2 * 3^{h+1}$
  $3 * (3^{h+1} - 3^h) = 2 * 3^{h+1}$
  $3^{h+1} - 3^h = 2 * 3^h$

  Ahora, aplicando la hipótesis inductiva, podemos llegar a la siguiente igualdad:
  $2 * 3^h = 2 * 3^h \ \surd$

  Es decir hemos probado tanto el caso base como el paso inductivo. Se concluye que $P(n)$ es Verdadero, $\forall$ n $\in \mathbb{N}$

■

Con el uso de esta proposición, se logra encontrar la manera de expresar el peso de la llave mediante sumas y restas de distintas potencias de 3(donde los términos positivos serían pesas en el platillo izquierdo y las restas en el derecho). Ahora, pasamos a explicar en que parte de nuestra solución se utiliza dicha proposición ya demostrada.

Este procedimiento, esta dado por la función *balancear*, cuyo funcionamiento es detallado a continuación:

Algoritmo 3 balancear

Input: $base3 \in \mathbb{Z}^n$
Output: $res \in \mathbb{Z}^{n+1}$.

```
for i = 0 to n-1 do
    if base3[i] = 2 then
        res[i] ← res[i] - 1
        res[i+1] ← res[i+1] + 1
    else if base3[i] = 1 then
        if res[i] = 1 then
            res[i] ← -1
            res[i+1] ← res[i+1] + 1
        else
            res[i] ← 1
        end if
    end if
    i ← i+1
end for
return res
```

El algoritmo, consiste en recorrer linealmente el desarrollo en base 3 de P, y paralelamente coloca en el vector *res* la valuación final de cada potencia (0 si no es usada, 1 si la pesa se encuentra en el platillo derecho y -1 en caso contrario), teniendo en cuenta la restricción del problema.

Para entender mejor que decisión se toma según el dígito encontrado, se divide en casos:

- $base3[i] = 2$: En este caso, se debe utilizar la proposición previamente demostrada, por lo cual se resta una vez la potencia actual y se suma la siguiente.

- $base3[i] = 1$: Si hasta el momento no se había sumado ninguna pesa del valor 3^i, simplemente se establece su valor en 1. Si ya había una, se la resta y se suma la siguiente potencia.
- $base3[i] = 0$: No se hace nada y se pasa a verificar el siguiente valor de $base3$.

2.3. Complejidad del algoritmo

El algoritmo para escribir un número en base 3 hace como máximo t iteraciones.

número de iteración	**valor P**
1	P
2	P/3
3	$(P/3)/3 = P/3^2$
4	$(P/3^2)/3 = P/3^3$
.	.
.	.
.	.
t	$P/3^{t-1}$

Luego de la última iteración, $P/3^{t-1}$ puede ser igual a 1 ó 2.

- Caso $P/3^{t-1} = 1$:
  $1 = \frac{P}{3^{t-1}} \Leftrightarrow 3^{t-1} = P \Leftrightarrow t = 1 + log_3(P)$
- Caso $P/3^{t-1} = 2$:
  $2 = \frac{P}{3^{t-1}} \Leftrightarrow \frac{2}{3}3^t = P \Leftrightarrow t = log_3(P\frac{3}{2}) = log_3(P) + log_3(\frac{3}{2})$

Entonces, la complejidad de este algoritmo es $\mathcal{O}(\log P)$. Como en cada iteración, se agrega un elemento al vector que retorna, la longitud del mismo también es log_3(P).

El algoritmo encargado de decidir donde colocar cada pesa, consta de un ciclo que realiza $|v|$ iteraciones, siendo v el vector que describe el desarrollo en base 3 de P. Siempre se aplica este algoritmo al vector retornado por la función *toBase3*, por lo que $|v|$ es igual a log_3(P). Como el resto de las operaciones de este algoritmo son $\mathcal{O}(1)$, la complejidad temporal total es $\mathcal{O}(\log P) \subseteq O(\sqrt{P})$.

2.4. Experimentación

El algoritmo implementado para resolver el problema tiene una complejidad teóricamente demostrada de $log_3(P)$ para el peor caso.

Dado que la única entrada del algoritmo consiste en el peso de la llave, los experimentos van a estar basados en los distintos valores que puede tomar el mismo.

Por la restricción impuesta en el enunciado, donde se indica que el peso de la llave debe ser menor a 10^{15}, podemos observar que la longitud del vector que representa el peso en base 3, será a lo sumo 32 (3^{32} es mayor que 10^{15}). Hecha esta observación, suponemos que no habrá diferencias en la performance en distintos casos, ya que la función *balancear* recorrerá un vector pequeño y solo efectúa comparaciones y asignaciones. Además, por la rapidez de los algoritmos, consideramos adecuado medir la performance en nanosegundos.

En primer lugar, realizamos las mediciones necesarias para confirmar que nuestra complejidad teórica es acertada, por lo que se procede a comparar el tiempo de ejecución del algoritmo en su mejor y peor caso (medido en nanosegundos) con la función $c * log(P)$, donde P es el peso de la llave y c una constante.

Para lograr un acercamiento lo más acertado posible, se tomó un total de 10000 mediciones por cada peso evaluado y se calculó un promedio de esos valores.

Para comparar la performance del algoritmo, elegimos medir el tiempo basándonos en P $= 3^i$ y P $= 3^i - 1$, ya que son casos muy distintos. En el primero, el desarrollo en base 3, tiene un 1 y luego, 0's. En cambio, el desarrollo en base 3 del segundo caso, estará conformado únicamente por 2's

Luego de realizar las mediciones pertinentes, los resultados fueron los siguientes:

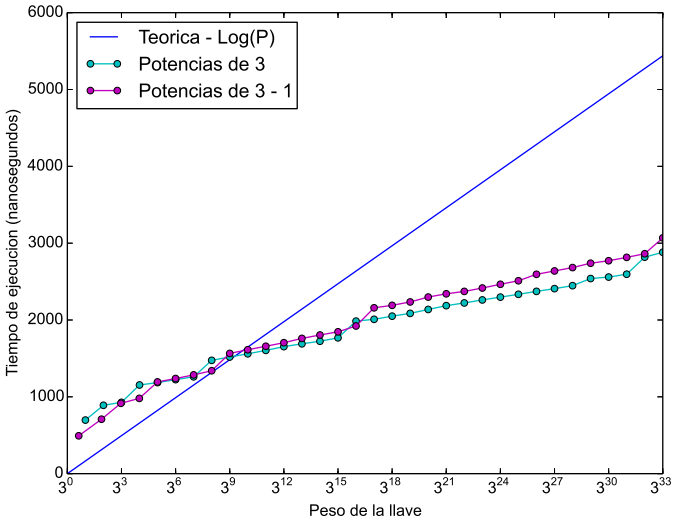

Figura 3: Tiempos de Ejecución en nanosegundos

Analizando los resultados obtenidos, pudimos sacar las siguientes conclusiones:

- Tal como habíamos planteado, en la figura se puede observar como la curva generada por el tiempo de ejecución puede ser acotada en ambos casos por una función $c*log(P)$ para un c fijo , por lo que podemos confirmar que nuestra complejidad teórica es acertada.
- En cuanto a la diferencia con respecto a la ejecución con las distintas entradas propuestas, los tiempos son prácticamente iguales, teniendo en cuenta que las mediciones fueron realizadas en nanosegundos, por lo que son sensibles a perturbaciones del sistema en el que están corriendo. Esto confirma nuestra hipótesis.
- Por último, al analizar detenidamente la forma de la curva en ambos casos, reconocimos la existencia de ciertos valores en los que el tiempo aumentaba de manera anómala en los puntos donde 3 es elevado a alguna potencia de dos. Al analizar el código de las funciones, atribuimos estos saltos al momento en el que el vector utilizado para almacenar el desarrollo de P en base 3 es extendido.

Para fundamentar las ultimas 2 observaciones, decidimos realizar más mediciones con respecto al comportamiento de los tiempos de ejecución en relación al posible valor de la entrada, en este caso con 100 valores al azar en el rango de posible peso de la llave. La toma de mediciones se realizó con los mismos recaudos que en el experimento anterior.

Finalmente, se tiene que $\frac{P}{log(n)}$ converge a una constante que puede ser acotada por c $=$ 1215. Es así que llegamos a la conclusión de que efectivamente la complejidad temporal de la solución desarrollada coincide con la complejidad teórica estipulada.

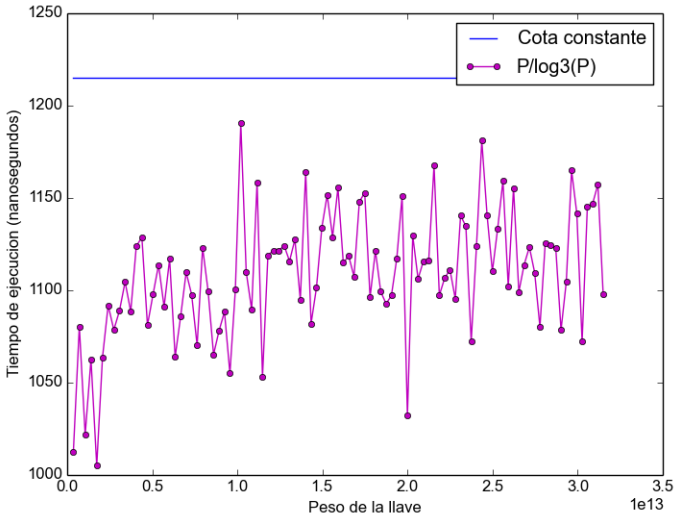

Figura 4: Tiempos de Ejecución - Entrada aleatoria

Como último punto en la experimentación, quedó pendiente confirmar o refutar nuestra hipótesis con respecto a esos saltos en la figura 3. Como mencionamos previamente, creemos que este comportamiento se debe a la extensión del vector, ya que el mismo se redimensiona al momento de completar su capacidad. Vamos a comparar sus tiempos de ejecución con una implementación de las mismas funciones pero con un vector de longitud estática (en particular de longitud 33, ya que esta determinada por las restricciones del problema).

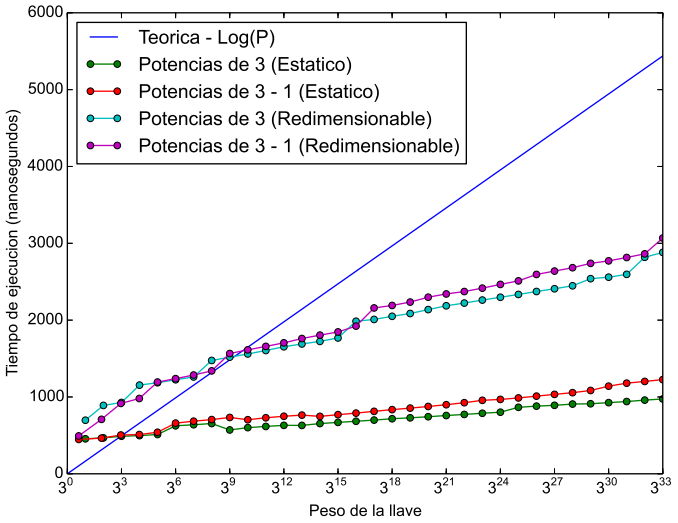

Figura 5: Tiempos de Ejecución - Distintas implementaciones

Confirmando nuestra hipótesis, se puede ver claramente la mejora en términos de performance con respecto al tiempo al utilizar un vector de longitud estática.

3. Problema 3: Guardando el tesoro

3.1. Introducción

Indy y sus arqueólogos logran entrar a una habitación donde se encuentran con tesoros. Hay exactamente N tipos de tesoros. Para guardarlos, el grupo cuenta con M mochilas disponibles para llevarse algunos recuerdos. Estas mochilas, tienen capacidad limitada, por lo que deben que elegir los artículos inteligentemente, para llevarse la mayor ganancia posible.

La resolución del problema consiste en elaborar un programa que reciba como entrada el valor de N y M, y para cada tipo de tesoro, su cantidad disponible, su peso y valor por unidad, y devuelva la ganancia máxima que se puede obtener a partir de poner tesoros en las mochilas sin superar su capacidad. También el programa deberá informar que tesoro va en cada mochila.

3.2. Desarrollo

Formalmente se tiene una lista de tesoros y M mochilas. Cada tesoro tiene un peso y un valor asociado. Queremos definir una función que aplicada a ciertos argumentos nos devuelva la solución al problema. Definimos a $g =$ "ganancia óptima considerando n tesoros para M mochilas de capacidad $k_1, \ldots, k_M$ respectivamente".

Queremos ver si a partir esta definición de g, vale que $g \equiv f$. Donde f es lo siguiente:

$$(\forall k_1, \ldots, k_M \in \mathbb{N})$$
$$f(0, k_1, \ldots, k_M) = f(n, 0, 0, \ldots, 0) = 0$$

$$(\forall k_1, \ldots, k_M \in \mathbb{N}, 0 < n)$$
$$f(n, k_1, \ldots, k_M) = \text{máx}\{f(n-1, k_1, \ldots, k_M), \max^*_{\substack{1 \le t \le M \\ peso(n) \le k_t}} \{f(n-1, k_1, \ldots, k_t - peso(n), \ldots, k_M) + valor(n)\}\}$$

$$\text{con} \quad \max^* S = \begin{cases} \text{máx}\, S & \text{si } S \neq \emptyset \\ 0 & \text{si } S = \emptyset \end{cases} \quad \text{(Sea S un conjunto cualquiera)}$$

Como inicialmente tenemos N tipos de tesoros con cantidades $c_1, \ldots, c_N$ de cada uno y M mochilas de capacidad $m_1, \ldots m_M$ respectivamente, si la función f devuelve la ganancia óptima (o sea $f \equiv g$), entonces la solución a nuestro problema es $f(\sum_{i=1}^{N} c_i, m_1, \ldots, m_M)$.

Estamos cubriendo todos los casos posibles para cada tesoro, y sobre todos esos nos quedamos con el que nos da la mayor ganancia. Para cada paso chequeamos a lo sumo $M+1$ situaciones, el tesoro no lo usamos, o lo usamos en la mochila 1, o en la 2, y así hasta la M, y sobre todos esos tomamos el máximo. Esta es la idea detrás de la prueba de correctitud del algoritmo que haremos más adelante.

Proposición: La función f devuelve la ganancia óptima.
Demostración: Lo haremos por inducción en la cantidad de tesoros:
Sea $P(n) : (\forall k_1, \ldots, k_M \in \mathbb{N}), \quad f(n, k_1, \ldots, k_M) \equiv$ "La ganancia óptima considerando n tesoros para M mochilas de capacidad $k_1, \ldots, k_M$".
Queremos probar que $P(n)$ es Verdadera para todo $n \in \mathbb{N}$.

- Caso base:
  $P(0) : f(0, k_1, \ldots, k_m) = 0$. Si no consideramos ningún elemento, la ganancia que se obtiene es nula, por lo tanto la mejor solución es 0 (no agregar ningún elemento). Se cumple $P(0)$.

- Paso inductivo:
  Si $n > 0$, queremos ver que $P(n-1) \Rightarrow P(n)$ es Verdadera.
  Hay dos casos: se logra la ganancia óptima incluyendo el tesoro n, o no haciendolo.

  - El tesoro n no está en la solución óptima:
    Como el tesoro n no está en la solución óptima entonces la ganancia máxima que podemos obtener con n tesoros sin considerar el último, es la ganancia máxima que podemos formar con los primeros $n-1$ tesoros. En términos de la función sería que $f(n, k_1, \ldots, k_M) = f(n-1, k_1, \ldots, k_M)$, que por **HI** sabemos que es la ganancia óptima.
    Entonces queremos ver que para este caso la función $f(n, k_1, \ldots, k_M)$ nos devuelve $\underbrace{f(n-1, k_1, \ldots, k_M)}_{\alpha}$.
    Suponemos que no:
    $$\Rightarrow f(n, k_1, \ldots, k_M) \neq \alpha$$
    Nuevamente tenemos dos casos para analizar:

    $$\text{Sea} \quad \gamma = \max^{*}_{\substack{1 \leq t \leq M \\ peso(n) \leq k_t}} \{f(n-1, k_1, \ldots, k_t - peso(n), \ldots, k_M) + valor(n)\}$$

    - $\alpha < \gamma$:

      $$\Rightarrow (\exists t \in \mathbb{N}, 1 \leq t \leq M) / \underbrace{f(n-1, k_1, \ldots, k_t - peso(n), \ldots, k_M)}_{\beta} + valor(n) > \alpha$$

      Hay una solución de valor β que usa los primeros $n-1$ tesoros y mochilas de capacidad $k_1, \ldots, k_t - peso(n), \ldots, k_M$. Pero como teníamos mochilas de capacidades $k_1, \ldots, k_t, \ldots, k_M$ podemos poner el tesoro n en la mochila k_t, y eso nos da una solución de valor $\beta + valor(n)$ para el caso de tener n tesoros, y como $\beta + valor(n) > \alpha$ entonces la mejor solución utiliza al tesoro n. Pero esto no puede pasar, porque estamos en el caso en que n no pertenece a la solución óptima, por lo tanto **ABS**!

    - $\alpha \geq \gamma$:

      $$\Rightarrow f(n, k_1, \ldots, k_M) = \text{máx}\{\alpha, \gamma\} = \alpha$$

      Pero habíamos supuesto que $f(n, k_1, \ldots, k_M) \neq \alpha$, **ABS**!

    Por los dos caminos llegamos a un absurdo, que vino de suponer que la función devolvía un valor distinto a α. Si la solución óptima no incluye al tesoro n, entonces lo mejor posible es la mejor opción hasta el tesoro anterior.

  - El tesoro n está incluido en la solución óptima:
    Como el tesoro está incluido en la solución óptima, tiene que estar en alguna mochila; entonces, sin pérdida de generalidad, podemos suponer que está en la mochila i-ésima.
    Veamos que $f(n, k_1, \ldots, k_M) = \underbrace{f(n-1, k_1, \ldots, k_i - peso(n), \ldots, k_M)}_{\delta} + valor(n)$.
    $\delta + valor(n)$ es la ganancia óptima. Por **HI**, δ es la ganancia máxima para los primeros $n-1$ tesoros y las mochilas de capacidad $k_1, \ldots, k_i - peso(n), \ldots, k_M$. Luego, como habíamos supuesto, en la mochila i-ésima ponemos el tesoro n.
    Para probar esto basta con ver que:

    $$\delta + valor(n) = \max_{\substack{1 \leq t \leq M \\ peso(n) \leq k_t}} \{f(n-1, k_1, \ldots, k_t - peso(n), \ldots, k_M) + valor(n)\}$$

y que:

$$\delta + valor(n) \geq \alpha$$

Si probamos eso y luego reemplazamos en la definición de la función llegamos a lo que queríamos probar:

$$f(n, k_1, \ldots, k_M) = \text{máx}\{\alpha, \delta + valor(n)\} \Rightarrow f(n, k_1, \ldots, k_M) = \delta + valor(n)$$

Veamos estas dos cosas por separado:

- $\delta + valor(n)$ es el máximo del conjunto:
  Basta con ver que el siguiente máximo se alcanza para la mochila i-ésima:

  $$\max_{\substack{1 \leq t \leq M \\ peso(n) \leq k_t}} \{f(n-1, k_1, \ldots, k_t - peso(n), \ldots, k_M) + valor(n)\}$$

  Suponemos que no, entonces:

  $$\begin{gathered} (\exists j \in \mathbb{N}, 1 \leq j \leq M) \quad / \\ f(n-1, k_1, \ldots, k_i - peso(n), \ldots, k_M) + valor(n) \\ < \\ f(n-1, k_1, \ldots, k_j - peso(n), \ldots, k_M) + valor(n) \\ \equiv \end{gathered}$$

  $$\equiv \underbrace{f(n-1, k_1, \ldots, k_i - peso(n), \ldots, k_M)}_{\text{Solución A}} < \underbrace{f(n-1, k_1, \ldots, k_j - peso(n), \ldots, k_M)}_{\text{Solución B}}$$

  Tenemos dos soluciones óptimas para los primeros $n-1$ tesoros y las respectivas capacidades de las mochilas. Como la Solución B es mejor que la A, conviene poner el tesoro n en la mochila j-ésima, pero eso no puede pasar porque la solución óptima tenía al tesoro n en la mochila i-ésima. Esto es absurdo, que vino de suponer que índice i no maximizaba la expresión.

- $\delta + valor(n) \geq \alpha$:
  Suponemos que no: Entonces vale que $\delta + valor(n) < \alpha$.
  α y δ son soluciones que usan los primeros $n-1$ tesoros y son óptimas por **HI**.
  Luego, agregamos el tesoro n a la solución δ y por lo que supusimos tenemos que es menor a otra solución que usa únicamente los a los primeros $n-1$ tesoros y no al n-ésimo. No puede pasar que haya una solución mejor que no utilice al tesoro n porque el mismo estaba incluido en la solución óptima. Nuevamente llegamos a un absurdo, que vino de suponer que $\delta + valor(n) < \alpha$.

Entonces f devuelve efectivamente "la ganancia óptima considerando n tesoros para M mochilas de capacidad $k_1, \ldots, k_M$". Por lo que $f(\sum_{i=1}^{N} c_i, m_1, \ldots, m_M)$ es la solución a nuestro problema.
Si encontramos un algoritmo que compute a f, tenemos resuelta la parte de la ganancia máxima y después faltaría ver en que mochila va cada tesoro. Se puede ver fácil un algoritmo recursivo que implemente a la función f para un M fijo.

Algoritmo 4 gananciaMaxima - exponencial

Input: $n, m_1, \ldots, m_M \in \mathbb{Z}$
Output: $res \in \mathbb{Z}$

```
if n = 0 or (m_1 = 0 and ... and m_M = 0) then
    return 0
else
    lista = [ ]
    if peso(n) ≤ m_1 then
        lista.append(valor(n) + gananciaMaxima(n − 1, m_1 − peso(n), ..., m_M))
    end if
    ⋮
    if peso(n) ≤ m_i then
        lista.append(valor(n) + gananciaMaxima(n − 1, m_1, ..., m_i − peso(n), ..., m_M))
    end if
    ⋮
    if peso(n) ≤ m_M then
        lista.append(valor(n) + gananciaMaxima(n − 1, m_1, ..., m_M − peso(n)))
    end if
    return máx(gananciaMaxima(n − 1, m_1, ..., m_M), lista)
end if
```

A partir del código se puede ver que la complejidad del algoritmo es exponencial. Pero se pedía resolver el problema con un algoritmo que a lo sumo realice $\mathcal{O}(T \prod_{i=1}^{M} m_i)$ operaciones. Donde T es la cantidad total de tesoros (incluyendo los repetidos de cada tipo) y M la cantidad de mochilas con m_i la capacidad de cada una.

Para mejorar la complejidad temporal, utilizamos programación dinámica como técnica algorítmica. Como ya tenemos una función recursiva que calcula la ganancia óptima (f), vamos a plantear los subproblemas en función de ella. La implementación utilizada es del tipo *bottom-up*, lo que quiere decir que comenzamos resolviendo subproblemas más simples y vamos avanzando hacia los más complejos. Guardamos las respuestas a cada subproblema en una matriz M-dimensional. Cada celda de la matriz representa la ganancia óptima conjunta para las mochilas de capacidades $m_1, \ldots, m_M$ habiendo recorrido hasta el tesoro n. En términos de nuestra función, tenemos la siguiente igualdad:

$$f(n, m_1, \ldots, m_M) = matriz[m_1] \ldots [m_M]$$

Para calcular el óptimo hasta el tesoro n vamos a necesitar el óptimo para el tesoro anterior, para cualquier capacidad menor o igual que las mochilas actuales. Esto nos da una idea de como ir calculando los valores. Como siempre se usan capacidades menores o iguales a las actuales, se puede ver que una opción válida es: para cada tesoro recorrer la matriz empezando desde los índices más grandes y luego ir decrementándolos en orden de a uno por vez.

Veámoslo con un ejemplo que luego puede ser generalizado: si $M = 2$ tenemos una matriz de dos dimensiones. Una forma posible de recorrerla es: empezar en la última fila y en la última columna e ir avanzando por columnas de derecha a izquierda y luego ir subiendo por las filas. Entonces vamos a ir recorriendo desde el primer tesoro hasta el n-ésimo y cada matriz la vamos completando de la forma mencionada anteriormente. Empezamos desde el primer tesoro y vamos aumentando porque como hicimos una implementación *bottom-up* para calcular el n-ésimo necesitamos ya tener calculado el anterior, y para el anterior el anterior y así sucesivamente.

3.2.1. Correctitud

A continuación presentamos el pseudocódigo propuesto para la función f. Como el algoritmo general, además de la ganancia máxima debe devolver que tesoro va en cada mochila, guardamos una matriz por tesoro para luego poder reconstruir la solución. Esto no haría falta si tuviéramos que decir solo el valor máximo. Como ya se conoce una cota para el valor $1 \leq M \leq 3$ (cantidad de mochilas), por cuestiones de implementación vamos a utilizar esta información.

Algoritmo 5 gananciaMaxima

Input: n, M1, M2, M3 $\in \mathbb{Z}$
Output: res $\in \mathbb{Z}$

```
// creamos un arreglo de n+1 posiciones
// con matrices de (M1+1) x (M2+1) x (M3+1) inicializadas con 0
matrices = crearArreglo(n+1, crearMatriz(M1+1, M2+1, M3+1))
for i = 0,...,n-1 do
    //copiamos la matriz anterior, i+1 es la actual
    //van defasadas +1 respecto a los tesoros
    matrices[i+1] = matrices[i]
    for m1 = M1,...,0 do
        for m2 = M2,...,0 do
            for m3 = M3,...,0 do
                opciones = [ ];
                opciones.push-back(matrices[i+1][m1][m2][m3]);
                if peso(i) ≤ m1 then
                    opciones.push-back(matrices[i+1][m1-peso(i)][m2][m3]+valor(i));
                end if
                if peso(i) ≤ m2 then
                    opciones.push-back(matrices[i+1][m1][m2-peso(i)][m3]+valor(i));
                end if
                if peso(i) ≤ m3 then
                    opciones.push-back(matrices[i+1][m1][m2][m3-peso(i)]+valor(i));
                end if
                matrices[i+1][m1][m2][m3] = opciones[indiceDelElementoMaximo(opciones)];
            end for
        end for
    end for
end for
return matrices[n][M1][M2][M3]
```

Veamos que la función *gananciaMaxima* computa a la función f:
Bastaría con mostrar que entre cada iteración del ciclo principal vale que $f(n, k_1, k_2, k_3) \equiv matrices(n, k_1, k_2, k_3)$.
Caso base:
$P(0) : f(0, k_1, k_2, k_3) = 0$ por definición. Si $n = 0$, entonces la función *gananciaMaxima* crea un arreglo de tamaño 1 con una matriz inicializada en 0 de tamaño $(k_1+1) \times (k_2+1) \times (k_3+1)$. Al primer ciclo no entra porque no se cumple la guarda. Luego la función devuelve una posición de la matriz que estaba inicializada en 0, por lo tanto el resultado es 0. Se cumple $P(0)$.
Paso Inductivo:
Si $n \geq 0$, queremos ver que $P(n) \Rightarrow P(n+1)$.

Al inicio de cada iteración se copia la $matriz(n)$ a la $matriz(n+1)$. Entonces $matriz(n) = matriz(n+1)$. Se puede ver que por la forma en que se recorre la matriz (desde las capacidades máximas hasta 0), se actualiza una única vez cada celda y un valor que ya fue actualizado no se vuelve a visitar. Esto sucede porque se comienzan desde las capacidades máximas, y cuando un objeto entra en alguna mochila, para consultar el valor que genera esa acción, se le resta al peso actual de la mochila [2] el peso del tesoro. Como siempre se resta en esos casos, nunca va a suceder que se visiten filas y/o columnas más grandes (ya visitadas).

Vimos que al finalizar una iteración, se actualizan todos los valores de la matriz en forma ordenada. Veamos que este nuevo valor corresponde a $f(n+1, k_1, k_2, k_3)$. Para cada k_i, el algoritmo calcula:

$$\text{máx}\{matrices[n+1][k1][k2][k3], \text{máx}\{opciones\}\} =$$

$$= \text{máx}\{matrices[n+1][k1][k2][k3], \max^{*}_{\substack{1\leq t\leq 3 \\ peso(n+1)\leq k_t}} \{valor(n+1)+matrices[n+1][k1][ki-peso(n+1)][k3]\}\} =$$

Pero como al inicio de cada iteración se hace una copia, vale que: $matriz(n) = matriz(n+1)$:

$$= \text{máx}\{matrices[n][k1][k2][k3], \max^{*}_{\substack{1\leq t\leq 3 \\ peso(n+1)\leq k_t}} \{valor(n+1)+matrices[n][k1][ki-peso(n+1)][k3]\}\} =$$

Sabemos por **HI** que $f(n, k_1, k_2, k_3) \equiv matrices(n, k_1, k_2, k_3)$ para todo k_i:

$$= \text{máx}\{f(n, k_1, k_2, k_3), \max^{*}_{\substack{1\leq t\leq 3 \\ peso(n+1)\leq k_t}} \{valor(n+1) + f(n, k_1, ...k_i - peso(n+1), ...k_3)\}\} =$$

$$f(n+1, k_1, k_2, k_3)$$

Ya tenemos el máximo valor, ahora nos falta decir que tesoro va en cada mochila. Para eso armamos un algoritmo que recibe como entrada un arreglo de matrices. La matriz i-ésima del arreglo tiene la solución del subproblema i, donde cada una corresponde al mayor valor de ganancia para cada capacidad posible, hasta el tesoro i. Además el algoritmo recibe por parámetro 3 listas para agregar los tesoros, una para cada mochila. A continuación el algoritmo:

[2] En este caso es el número de fila o columna de la matriz, dependiendo de que mochila sea.

Algoritmo 6 guardarTesoros

Input: matrices, lista1, lista2, lista3
Output: modifica lista1, lista2 y lista3

```
// Recorremos las matrices de cada paso para ver que tesoro pusimos en cada mochila
m1 = capacidad de la m1
m2 = capacidad de la m2
m3 = capacidad de la m3
n = cantidad de tesoros
for i = n-1,..., 0 do
    if matrices[i][m1][m2][m3] != matrices[i+1][m1][m2][m3] then
        // el tesoro i-esimo lo pusimos,
        // ahora nos fijamos en que mochila
        opciones = [-1, -1, -1]
        if peso(i) ≤ m1 then
            opciones[MOCHILA-1] = matrices[i][m1-peso(i)][m2][m3]
        end if
        if peso(i) ≤ m2 then
            opciones[MOCHILA-2] = matrices[i][m1][m2-peso(i)][m3]
        end if
        if peso(i) ≤ m3 then
            opciones[MOCHILA-3] = matrices[i][m1][m2][m3-peso(i)]
        end if
        switch (indiceDelElementoMaximo(opciones))
            case MOCHILA-1:
                // agrego el el tesoro actual a los elementos de la mochila 1
                lista1.append(tesoro(i))
                m1 -= peso(i)
            case MOCHILA-2:
                // agrego el el tesoro actual a los elementos de la mochila 2
                lista2.append(tesoro(i))
                m2 -= peso(i)
            case MOCHILA-3:
                // agrego el el tesoro actual a los elementos de la mochila 3
                lista3.append(tesoro(i))
                m3 -= peso(i)
        end switch
    end if
end for
```

Para armar la solución recorremos las matrices desde el último paso hasta el primero. Nos fijamos si paso a paso (de a pares) la ganancia máxima cambió:

- Si cambió:

  Si el valor no es el mismo quiere decir que para alcanzar esa ganancia máxima tuvimos que haber usado ese tesoro. Entonces nos fijamos en que mochila se puede ubicar de forma que al seguir recorriendo las nuevas mochilas con capacidades actualizadas, obtengamos el máximo valor posible. Entonces agregamos el tesoro a esa mochila, y luego actualizamos el peso de la misma restándole el peso del tesoro. Seguimos recorriendo.

- Si no cambió:

Entonces quiere decir que para este paso el valor del objeto no fue tenido en cuenta, es decir, este objeto no está en la solución óptima. No lo agregamos y seguimos recorriendo.

3.2.2. Complejidad

Luego partiendo de que los algoritmos anteriores ($gananciaMaxima$ y $guardarTesoros$) son correctos, podemos combinarlos de forma sencilla y generar un único algoritmo más compacto que resuelva todo el problema. Entonces por último nos queda ver que el algoritmo final cumple con la cota de complejidad pedida.

Algoritmo 7 gananciaMaximaConTesoros - Parte 1

Input: n, M1, M2, M3, listaM1, listaM3, listaM3
Output: res $\in \mathbb{Z}$, modifica listaM1, listaM3, listaM3

```
1:  res = 0
2:  matrices = crearArreglo(n+1, crearMatriz(M1+1, M2+1, M3+1))
3:  for i = 0,...,n-1 do
4:      //copiamos la matriz anterior, i+1 es la actual
5:      //van defasadas +1 respecto a los tesoros
6:      matrices[i+1] = matrices[i]
7:      for m1 = M1,...,0 do
8:          for m2 = M2,...,0 do
9:              for m3 = M3,...,0 do
10:                 opciones = [ ];
11:                 opciones.push-back(matrices[i+1][m1][m2][m3]);
12:                 if peso(i) ≤ m1 then
13:                     opciones.push-back(matrices[i+1][m1-peso(i)][m2][m3]+valor(i));
14:                 end if
15:                 if peso(i) ≤ m2 then
16:                     opciones.push-back(matrices[i+1][m1][m2-peso(i)][m3]+valor(i));
17:                 end if
18:                 if peso(i) ≤ m3 then
19:                     opciones.push-back(matrices[i+1][m1][m2][m3-peso(i)]+valor(i));
20:                 end if
21:                 matrices[i+1][m1][m2][m3] = opciones[indiceDelElementoMaximo(opciones)];
22:             end for
23:         end for
24:     end for
25: end for
26: res = matrices[n][M1][M2][M3]
```

Algoritmo 8 gananciaMaximaConTesoros - Parte 2

```
27: // Recorremos las matrices de cada paso para ver que tesoro pusimos en cada mochila
28: m1 = M1
29: m2 = M2
30: m3 = M3
31: n = cantidad de tesoros
32: for i = n-1,..., 0 do
33:     if matrices[i][m1][m2][m3] ≠ matrices[i+1][m1][m2][m3] then
34:         // el tesoro i-esimo lo pusimos,
35:         // ahora nos fijamos en que mochila
36:         opciones = [-1, -1, -1]
37:         if peso(i) ≤ m1 then
38:             opciones[MOCHILA-1] = matrices[i][m1-peso(i)][m2][m3]
39:         end if
40:         if peso(i) ≤ m2 then
41:             opciones[MOCHILA-2] = matrices[i][m1][m2-peso(i)][m3]
42:         end if
43:         if peso(i) ≤ m3 then
44:             opciones[MOCHILA-3] = matrices[i][m1][m2][m3-peso(i)]
45:         end if
46:         switch (indiceDelElementoMaximo(opciones))
47:             case MOCHILA-1:
48:                 // agrego el el tesoro actual a los elementos de la mochila 1
49:                 lista1.append(tesoro(i))
50:                 m1 -= peso(i)
51:             case MOCHILA-2:
52:                 // agrego el el tesoro actual a los elementos de la mochila 2
53:                 lista2.append(tesoro(i))
54:                 m2 -= peso(i)
55:             case MOCHILA-3:
56:                 // agrego el el tesoro actual a los elementos de la mochila 3
57:                 lista3.append(tesoro(i))
58:                 m3 -= peso(i)
59:         end switch
60:     end if
61: end for
62: return res
```

Queremos ver que $gananciaMaximaConTesoros$ es $\mathcal{O}(T \prod_{i=1}^{M} m_i)$. Donde T es la cantidad total de tesoros (incluyendo los repetidos de cada tipo) y M la cantidad de mochilas con m_i la capacidad de cada una.

La instrucción en la línea 2 tiene un costo de $\mathcal{O}((n+1)(M1+1)(M2+1)(M3+1))$ porque se crea un arreglo de $n+1$ posiciones donde cada elemento es una matriz de $(M1+1) \times (M2+1) \times (M3+1)$.

Analicemos el ciclo que comienza en la tercera línea, se ejecuta n veces. Luego dentro del cuerpo del ciclo tenemos en la línea 6 una copia de una matriz que tiene un costo de $\mathcal{O}((M1+1)(M2+1)(M3+1))$ que se va a sumar al costo que tenga el ciclo que empieza en la línea 7. Este último ejecuta $((M1+1)(M2+1)(M3+1))$ veces instrucciones constantes: crear un vector vacío es $\mathcal{O}(1)$, agregar un elemento al mismo es, en el peor caso, $\mathcal{O}(cantidad-de-elementos)$ y como a lo sumo vamos a tener 4 elementos eso es $\mathcal{O}(1)$. El resto de las

instrucciones son accesos a vectores o vectores de vectores, comparaciones y asignaciones, o sea que son constantes. La función $indiceDelElementoMaximo()$ es lineal en la cantidad de elementos, el peor caso es $\mathcal{O}(4) = \mathcal{O}(1)$. Por lo tanto el ciclo que comienza en la línea 7 y termina en la 24 realiza $(M1+1)(M2+1)(M3+1)\mathcal{O}(1) = \mathcal{O}((M1+1)(M2+1)(M3+1))$. Entonces el costo total del ciclo de la tercera línea cuesta $n \cdot (\mathcal{O}((M1+1)(M2+1)(M3+1)) + \mathcal{O}((M1+1)(M2+1)(M3+1))) = \mathcal{O}(n(M1+1)(M2+1)(M3+1))$. Hasta ahora tenemos el costo de la segunda línea $\mathcal{O}((n+1)(M1+1)(M2+1)(M3+1))$ más el costo de la tercera $\mathcal{O}(n(M1+1)(M2+1)(M3+1))$, eso es $\mathcal{O}(n \cdot M1 \cdot M2 \cdot M3)$. Notar que los "+1" no afectan a la complejidad.

Hasta la línea 25 tenemos un costo total de $\mathcal{O}(n \cdot M1 \cdot M2 \cdot M3)$. Desde la 26 hasta la 31 son tadas asignaciones que consumen tiempo constante. Nos queda por analizar el ciclo que empieza en la 32: el mismo se ejecuta n veces. En el cuerpo del ciclo tenemos todas operaciones constantes, veámoslas: los accesos a vectores los consideramos constantes, las comparaciones también, en la línea 36 creamos un vector de 3 posiciones lo cual se hace en $\mathcal{O}(1)$. Por último tenemos nuevamente un llamado a la función $indiceDelElementoMaximo()$ que es lineal en la cantidad de elementos, en este caso tenemos $3 \implies \mathcal{O}(1)$. Agregar un elemento a una lista también se realiza en tiempo constante; podemos concluir que el ciclo que comienza en la línea 32 es $\mathcal{O}(n)$. Por lo tanto el costo final de la función es $\mathcal{O}(n \cdot M1 \cdot M2 \cdot M3) + \mathcal{O}(n) = \mathcal{O}(n \cdot M1 \cdot M2 \cdot M3)$. Como n es la cantidad total de tesoros y $M1, M2, M3$ las capacidades de cada mochila, se cumple que la función es $\mathcal{O}(T \prod_{i=1}^{M} m_i)$ como queríamos ver.

3.3. Experimentación

Vimos que el algoritmo propuesto resuelve el problema en a lo sumo $\mathcal{O}(T \prod_{i=1}^{M} m_i)$. Donde T es la cantidad total de tesoros (incluyendo los repetidos de cada tipo) y M la cantidad de mochilas con m_i la capacidad de cada una.
Queremos constatar empíricamente que esta familia de funciones es una cota superior para el tiempo de ejecución del algoritmo. Dado que la función depende de más de un parámetro, analizaremos de uno.

3.3.1. Cantidad de tesoros

Queremos ver experimentalmente que variando la cantidad de tesoros y dejando las capacidades de las mochilas fijas, obtenemos variaciones lineales en cuanto al tiempo de ejecución. Para realizar este experimento utilizamos dos mochilas de capacidades 10 y 20 respectivamente. Luego fuimos tomando mediciones de cuánto tardaba el algoritmo en resolver una instancia dada para 1 tesoro, luego para 2 y así sucesivamente hasta 50 (el máximo número de tesoros que pedía el enunciado). Para cada medición realizamos 100 repeticiones y luego el promedio de las ejecuciones son los distintos puntos que generan el gráfico. En el pseudocódigo se ve que cuando un tesoro entra en alguna mochila, este se tiene en cuenta para calcular posteriormente el máximo entre todas las posibilidades. Entonces si un objeto no entra en ninguna mochila, el algoritmo realiza una menor cantidad de instrucciones que si entrara en alguna. Gracias a este comportamiento del algoritmo, hay instancias que tienen una misma cota de complejidad pero que sus constantes son menores. La constante va a ser menor cuando todos los objetos no entren en ninguna mochila, y mayor cuando todos los objetos tengan un peso menor o igual al mínimo de las capacidades de las mochilas. También podemos ver que si los pesos de los tesoros son aleatorios entre 1 y un valor mayor a la capacidad de la mochila más grande, va haber ciertos tesoros que no van a entrar en ninguna mochila, pero también habrá otros que entran en todas. En esta última situación, deberíamos observar que el tiempo de ejecución es mayor que el caso en que ningún tesoro entra y menor comparado al caso en

que todos entran. Solo nos preocupa generar el peso de forma aleatoria y no así su valor o cantidad del tipo de tesoro. Su valor esta acotado y no tiene influencia en la complejidad y si hay una cantidad mayor a uno de algún tipo de tesoro, estos se pueden pensar como esa cantidad de tipos de tesoros distintos pero con mismo peso y valor.

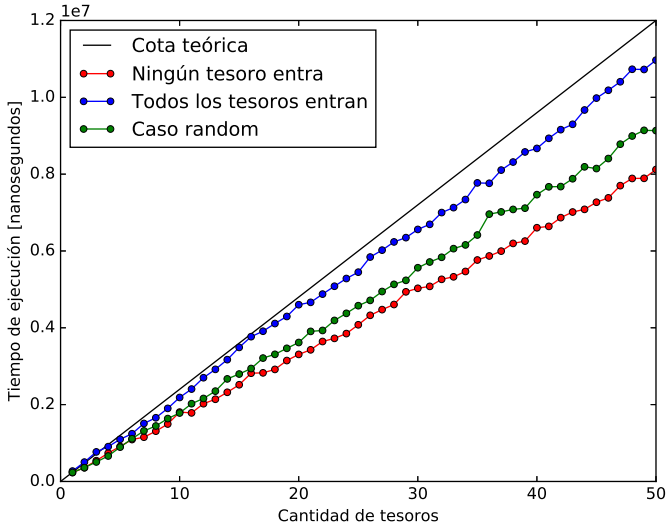

Figura 6: Comparación entre tiempos de ejecución para mochilas de tamaño fijo y variando la cantidad de tesoros. Se promediaron 100 corridas para cada punto del gráfico.

En la figura 6 se puede ver que, como supusimos, variar la cantidad de tesoros dejando las capacidades fijas tiene un comportamiento lineal en el tiempo de ejecución. También como era de esperar el caso que menos tarda es cuando ningún tesoro entra en las mochilas debido a que se hacen menos comparaciones y accesos a memoria. Las instancias generadas con pesos random demoran más que el caso en que no entra ningún tesoro.

3.3.2. Capacidad de la mochila

De la misma forma que para el experimento anterior, si dejamos todas las variables fijas y vamos calculando el tiempo de ejecución para las distintas capacidades de una mochila, nuevamente se obtiene que su variación es lineal. Queremos ver que sucede si variamos al mismo tiempo la capacidad de dos mochilas para una cierta cantidad de tesoros fija. Entonces vamos a considerar dos mochilas que comienzan con capacidad cero y van aumentando al mismo tiempo de a uno. De la función de complejidad se puede ver que al aumentar las capacidades simultáneamente el tiempo de ejecución aumenta cuadráticamente. Vamos a realizar un experimento para corroborar este comportamiento. Para esto vamos a tomar una cantidad fija de 30 tesoros con valores de pesos aleatorios entre 0 y 50. También para cada medición tomaremos 50 repeticiones. El total de mochilas será de dos y ambas comenzarán con un capacidad de 0, luego las iremos aumentando simultáneamente hasta que tengan una capacidad de 50.

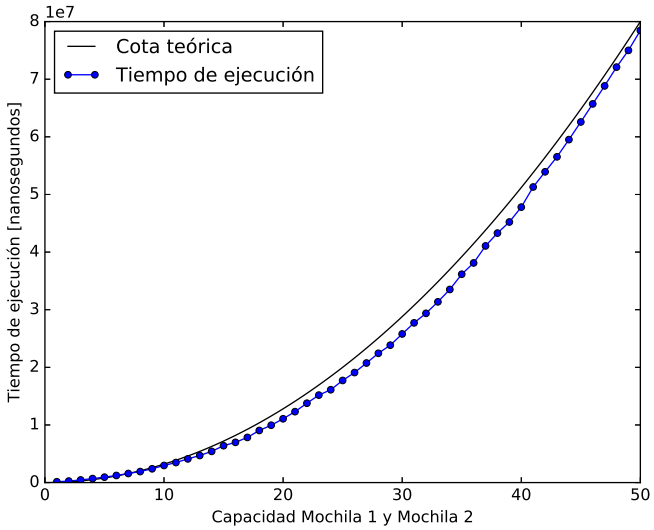

Figura 7: Tiempos de ejecución para 30 tesoros variando las capacidades de dos mochilas. El eje de las abscisas representa la capacidad de cada mochila. Se promediaron 50 corridas para cada punto del gráfico.

De la figura 7 se puede ver que la suposición era correcta. Pudimos ver empíricamente que la cota encontrada analíticamente a partir del algoritmo coincide con la experimental.