DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Recuperatorio de Trabajo Práctico I

Indiana Jones en búsqueda de la complejidad esperada

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Arroyo, Luis Alberto	913/13	luis.arroyo.90@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) 4576-3300
http://www.exactas.uba.ar

Contents

1 Introducción

En este informe, vamos a analizar la resolución de un mismo problema, en este caso el de encontrar dos subsecuencias, una creciente y una decreciente dentro de una secuencia de numeros, de manera tal que ambas secuencias tentas pintadas la mayor cantidad de números posible. Para ello resolveremos este problema de tres formas distintas, la primera será resuelta por medio de Backtracking; En la segunda se le agregará una poda y por ultimo se explicará un algoritmo de programacion dinámica.

Cada ejercicio presentará En este informe explicaremos el desarrollo del código realizado para resolver los problemas pertenecientes al Trabajo Práctico 1 y la justificación de la complejidad obtenida de cada ejercicio.

Los problemas serán resueltos utilizando las técnicas algorítmicas conocidas como Backtracking, programación dinámica y algoritmos golosos.

El código está desarrollado en Java y los gráficos los generamos a partir de la pagina de internet para crear gráficos a partir de tablas de datos, https://plot.ly/ y el programa gnuplot.

Es importante destacar además que a medida que se resolvieron los problemas, las soluciones fueron probadas para verificar el funcionamiento del código que serán presentadas como tests.

Para justificar la complejidad de los algoritmos que resuelven los problemas se utiliza pseudocódigo y luego una justificación de la complejidad del algoritmo. Por último, se realizará una experimentación para comparar el funcionamiento del algoritmo con la complejidad estimada.

2 Problema I: Cruzando el puente

2.1 Introducción

En el problema se pide que los arqueólogos y la tribu local crucen un puente colgante. Se tiene una sola linterna para cruzar el puente y es de noche. El puente puede ser cruzado de hasta dos personas a la vez y se debe cruzar con la linterna para iluminar el camino. Además, la gente de la tribu es caníbal y se pide que la cantidad de arqueólogos que haya en cualquier lado del puente sea mayor o igual a la de caníbales para evitar ser devorados.

Cada persona tarda un tiempo fijo en cruzar el puente. Si cruzan dos personas, tardarán lo mismo que la persona más lenta.

En el ejercicio se busca minimizar el tiempo en el que todos cruzan el puente, si es que hay una forma de hacerlo sin que ningún caníbal se coma a ningún arqueólogo.

2.1.1 Ejemplos

En los siguientes ejemplos mostraremos las soluciones a dos situaciones sin caníbales, en las que los arqueólogos deben cruzar al otro lado. Los arqueólogos serán notados como a(t), en donde t indica el tiempo que tardan en cruzar el puente.

Origen	Movimiento	Destino	Tiempo
Ejemplo 1			
Estado 0 a(1),a(2),a(7),a(8)	-	-	0
Estado 1: ida a(7), a(8)	a(1), a(2) →	-	2
Estado 2: vuelta a(7), a(8)	← a(1)	a(2)	3
Estado 3: ida a(1)	a(7), a(8) →	a(2)	11
Estado 4: vuelta a(1)	← a(2)	a(7), a(8)	13
Estado 5: ida -	a(1), a(2) →	a(7), a(8)	15
Estado 6 -	-	a(1), a(2), a(7), a(8)	15
Ejemplo 2			
Estado 0 a(1),a(9),a(10),a(11)	-	-	0
Estado 1: ida a(10), a(11)	a(1), a(9) →	-	9
Estado 2: vuelta a(10), a(11)	← a(1)	a(9)	10
Estado 3: ida a(11)	a(1), a(10) →	a(9)	20
Estado 4: vuelta a(11)	← a(1)	a(9), a(10)	21
Estado 5: ida -	a(1), a(11) →	a(9), a(10)	32
Estado 6 -	-	a(1), a(9), a(10), a(11)	32

2.2 Resolución del problema

El problema se resolvió con backtracking. Este algoritmo se eligió pues para saber cuanto se tarda en llegar al estado final (todos en la orilla destino) necesitamos hacer todos los pasos válidos hasta llegar al mismo y sumar cuanto tiempo cuesta cada acción (enviar a alguien por el puente). Ésto es básicamente recorrer el arbol de backtracking.

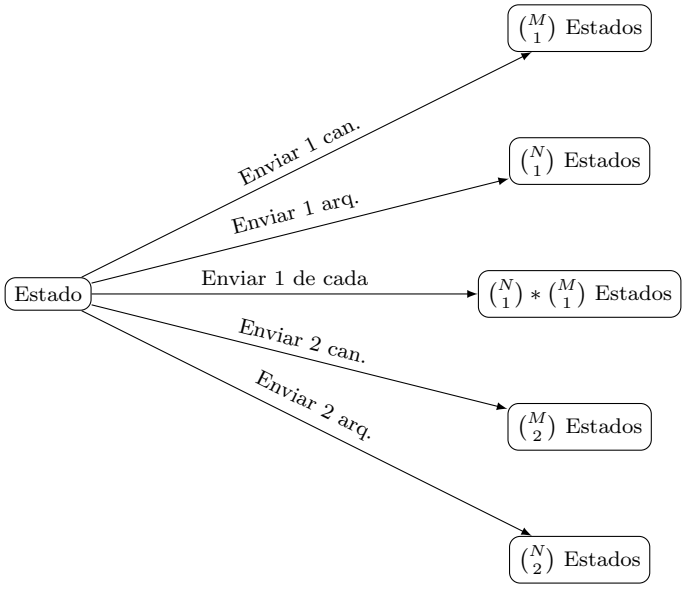
Podemos observar por medio de los ejemplos que no hay una forma clara de resolverlo sin utilizar este método: minimizar el tiempo de las idas sirve en el ejemplo 1 pero no en el ejemplo 2. Minimizar el tiempo de las vueltas funciona para el ejemplo 2 pero no para el 1. También podemos notar que no hay un principio de optimalidad claro: el estado 3 de menor tiempo se consigue siguiendo los pasos del ejemplo 1, pero el ejemplo 2 requiere otro estado 3 para dar el resultado óptimo.

Definición: Un estado describe la ubicación de cada arqueólogo y caníbal en el problema, y la ubicación de la linterna. En los estados los arqueólogos y caníbales están en una u otra orilla, pero nunca en el puente.

Nos movemos de un estado posible del problema a otro analizando quienes pueden cruzar el puente. De ésta forma podamos los estados no accesibles, y los que accederlos cuesta más que una solución previamente encontrada.

Tenemos en cuenta que los movimientos permitidos pueden ser estados ya visitados. Para evitar ello debemos guardar los estados accedidos y el tiempo que hemos tardado en accederlos, para podar las ramas que vuelvan a encontrarlos en un tiempo mayor.

Verificando ambas cosas, el algoritmo crea un backtracking donde cada nodo del árbol que representa es un estado, y sus nodos hijos son los estados a los que se puede acceder realizando movimientos válidos siempre y cuando no sean estados ya visitados en menor tiempo.



Notar que la cantidad de hijos de cada nodo es $\binom{N}{2}+\binom{N}{2}+\binom{N}{1}*\binom{M}{1}+\binom{N}{1}+\binom{M}{1} = \binom{N+M}{2}+\binom{N+M}{1} = \frac{(N+M)*(N+M-1)}{2} + N + M = \frac{(N+M)^2+N+M}{2}$ que es

$$\mathcal{O}((N+M)^2)$$

2.3 Algoritmos

Nuestra solución primero verifica que no haya más caníbales que arqueólogos, de haber algún arqueólogo. En ese caso, ya en el estado inicial los caníbales se los comerían.

Luego se arma el puente que cuenta con 2 objetos "orilla" y métodos para mover arqueólogos y caníbales de una orilla a la otra. Las orillas contienen una lista de arqueólogos y una de caníbales.

Algorithm 1 resolver

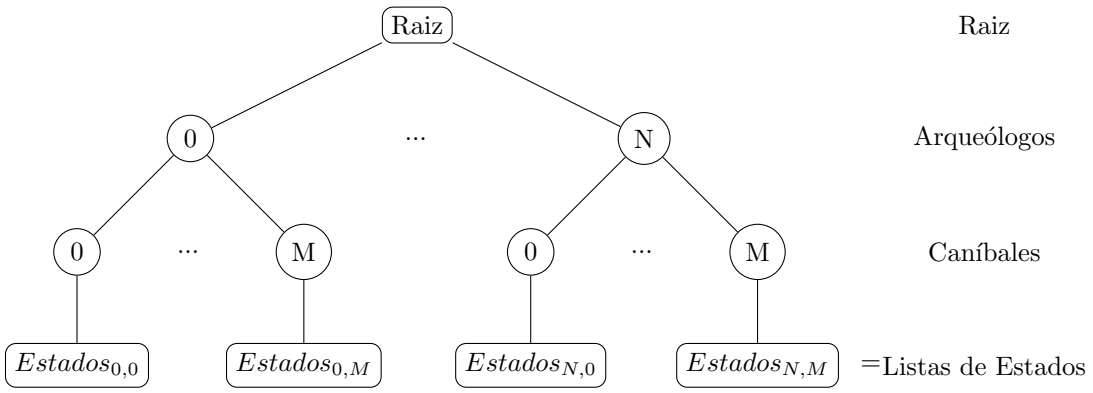
```
1: procedure RESOLVER(List<Long> arq, List<Long> can) → long
2:     //Verifico si no hay solución.
3:     if arq.size() < can.size() ∧ arq.size() > 0 then
4:         return -1;
5:     end if
6:     tTotal ← ∞;
7:     Orilla origen ← new Orilla(arq, can);
8:     //orilla con todos al comienzo
9:     Orilla destino ← new Orilla();
10:     //orilla destino
11:    Puente puente ← new Puente(origen, destino)
12:    estados ← new ArbolEstados(arq.size(),can.size());
13:    res ← Ida(puente, 0);
14:    return res
15: end procedure
```

Complejidad: $\mathcal{O}(N + M) + \mathcal{O}(ida)$

Como recorrer una rama del árbol de backtracking es equivalente a moverse de un estado a otro inmediato (enviando una sola vez gente por el puente), y siendo que se pueden tanto enviar como recibir hasta 2 personas, los estados pueden repetirse.

Para eso creamos un árbol de estados, el cual tiene N+1 nodos representando cada uno la cantidad de arqueólogos que tiene una orilla, y cada uno de estos nodos tiene M+1 nodos hijos, representando la cantidad de caníbales en la misma orilla. Cada nodo de caníbales posee una lista con todos los estados donde la orilla guardada tiene la cantidad de arqueólogos y caníbales correspondientes a la rama.

La estructura de árbol es muy útil en ésta situación ya que permite conseguir la lista de los estados con la cantidad de arqueólogos y caníbales del estado buscado, acotando la búsqueda significativamente.



Verificar el estado es equivalente a recorrer los N+1 nodos representando la cantidad de arqueólogos y luego los M+1 nodos de los caníbales hasta dar con la lista de estados con esas cantidades. Luego, hay que recorrer una lista de estados que potencialmente tiene todas las combinaciones de estados sin repetición, que a lo sumo son

$$\binom{N+M}{(N+M)/2}$$

Los estados se guardan y verifican antes de enviar personas por el puente, en las funciones *ida()* y *vuelta()*.

Para pasar de un estado a otro debemos verificar qué acciones se pueden tomar. Para ello armamos 5 funciones que prueban las posibles combinaciones para enviar gente por el puente (1 o 2 arqueólogos, 1 o 2 caníbales y uno de cada) llamadas **prueboEnviar*personas*** donde *personas* son las combinaciones antes descritas. Cada una de éstas utiliza el método correspondiente de la clase Puente.

Mostraremos como creamos todas las ramas donde enviamos 2 arqueólogos. Primero, el método de la clase Puente correspondiente es enviarArq(long,long):

Algorithm 2 enviarArq

```
1:  procedure ENVIARARQ(long i, long j) → long
2:      if orilla1.arqSize() < 2 ∨ (orilla1.dif() < 2 ∧ orilla1.arqSize() > 2) ∨ orilla2.dif() < −2 then
3:          return -1;
4:      end if
5:      tiempo ← orilla1.getArq(max(i,j));
6:      orilla2.addArq(tiempo);
7:      tiempo2 ← orilla1.getArq(min(i,j));
8:      orilla2.addArq(tiempo2);
9:
10:     return Math.max(tiempo, tiempo2)
11: end procedure
```

Complejidad: $\mathcal{O}(N) + \mathcal{O}(1) + \mathcal{O}(N) + \mathcal{O}(1) = \mathcal{O}(N)$

Justificación: Borramos los arqueólogos de la Orilla origen y los agregamos a la Orilla destino. getArq() cuesta $\mathcal{O}(N)$ y addArq() cuesta $\mathcal{O}(1)$. Luego devolvemos el máximo de los tiempos, que es lo que tardan ambos en cruzar el puente.
Observaciones: Se pueden enviar todas las combinaciones de 2 arqueólogos o ninguna. En el segundo caso, devuelve "-1" en $\mathcal{O}(1)$

La funcion *dif()* devuelve la cantidad de caníbales de una orilla menos la cantidad de arqueólogos de la misma. El **if** verifica si hay 2 arqueólogos para enviar, y que si éstos se envían no queden más caníbales que arqueólogos en alguna de las 2 orillas que tenga algún arqueólogo.

Algorithm 3 prueboEnviar2Arqueologos

```
1: procedure PRUEBOENVIAR2ARQUEOLOGOS(Puente p, long tParcial, long tVueltaMin, boolean ida)
   → long
2:     for i = 0; i < N−1; i++ do
3:         for j = i+1; j < N; j++ do
4:             q ← p
5:             t ← q.enviarArq(i, j);
6:             if t < 0 then
7:                 return -1;
8:             end if
9:             //Comparo el tiempo hasta este estado con el tiempo total, para podar
10:            if tParcial + t < tTotal then
11:                tVuelta ← vuelta(p,tParcial+t)
12:                if tVuelta ≥ 0 ∧ (t + tVuelta) < tVueltaMin then
13:                    //Me quedo con el mejor tiempo desde éste estado hasta el estado final
14:                    tVueltaMin ← t + tVuelta
15:                    tTotal ← (tParcial + tVueltaMin)
16:                end if
17:            end if
18:        end for
19:    end for
20:
21:    return tVueltaMin
22: end procedure
```

Complejidad: $\mathcal{O}(N^2) * (\mathcal{O}(enviarArq()) + \mathcal{O}(Ida())) = \mathcal{O}(N^3) + \mathcal{O}(N^2) * \mathcal{O}(Ida()) \Rightarrow \mathcal{O}(N^3)$+ recursión.

Justificación: Iteramos sobre todas las combinaciones de 2 arqueólogos posibles, y los movemos de una orilla a la otra. Moverlo cuesta lo mismo que recorrer la lista de arqueólogos y eliminar el buscado, que es $\mathcal{O}(N)$. Luego, llamamos a la función ida (o vuelta, que tiene la misma complejidad)

Observaciones:

- De aquí en más omitiremos el *+recursión* en los cálculos de complejidad dado que luego calcularemos la complejidad total del algoritmo en base al costo de los nodos sin la recursión.
- enviarArq(i,j) tiene la misma complejidad que enviarArq(i), vuelveArq(i,j) y vuelveArq(i). Idem para las funciones sobre caníbales, que todas son del orden $\mathcal{O}(M^3)$. enviarAmbos() y vuelveAmbos() son del orden $\mathcal{O}(N * M * (N + M))$. Todas las complejidades podrían escribirse como $\mathcal{O}(max(N, M)^3)$ o $\mathcal{O}((N + M)^3)$

tParcial: Siendo que ésta función se llama en el $Estado_i$, tParcial acumula el tiempo desde $Estado_0$ hasta $Estado_{i-1}$.

tVueltaMin: Con ésta variable se devolverá el tiempo desde $Estado_i$ hasta $Estado_{final}$.

tTotal: Es el tiempo de la mejor solución encontrada.

ida: Indica si el método es llamado desde *ida()* o *vuelta()*. En éste caso, escribimos el pseudocódigo como si fuera llamado desde *ida()*

Por último, los métodos principales que representan los nodos del backtracking son *ida()* y *vuelta()*. La diferencia principal en ambos es que *vuelta()* debe verificar al comenzar si se ha llegado al estado final, pues de ser así no hay que enviar a nadie de regreso a la orilla de origen.

Algorithm 4 Ida

```
1: procedure IDA(Puente p, long tParcial) → long
2:     Ordeno la orilla origen (para comparar correctamente los estados)
3:     Verifico que éste estado no haya sido alcanzado previamente en un tiempo menor
4: con checkEstado().
5:
6:     Pruebo todas las formas de enviar 2 arqueólogos.
7:     Pruebo todas las formas de enviar 2 caníbales.
8:     Pruebo todas las formas de enviar 1 arqueólogo y 1 caníbal.
9:     Pruebo todas las formas de enviar 1 arqueólogo.
10:    Pruebo todas las formas de enviar 1 caníbal.
11:
12:    Si ninguna de las anteriores fue posible o tardaron más que una solución previa, devuelvo -1.
13:    De otra forma, devuelvo el tiempo que cuesta la mejor solución a partir del estado actual.
14: end procedure
```

Complejidad: $\mathcal{O}(Sort()) + \mathcal{O}(checkEstado()) + \mathcal{O}(5 * prueba[...]()) = \mathcal{O}((N+M) * log_2(N+M)) + \mathcal{O}(\binom{N+M}{(N+M)/2}) + \mathcal{O}(5 * max(N,M)^3) = \mathcal{O}(\binom{N+M}{(N+M)/2}) = \mathcal{O}((N+M)!)$

Justificación: Ordenamos con Collections.sort() que está documentado como $\mathcal{O}((N+M) * log_2(N+M))$. El tiempo de checkEstado() se justificó en el dibujo del árbol de estados. Luego, si el estado no fue previamente visitado en un tiempo menor, ejecutamos 5 funciones con el mismo tiempo que prueboEnviar2Arqueologos().

En una versión anterior de nuestro código, el método *vuelta()* evaluaba los métodos para enviar personas a la orilla origen en el mismo orden que *ida()*. En un análisis posterior notamos que salvo un caso (3 arqueólogos y 3 caníbales) los únicos movimientos que se utilizaban en la solución óptima eran de enviar 2 personas a la orilla destino y retornar una.

Si bien no pudimos probar cuánto se utilizan las opciones de enviar una persona o de retornar 2, mostraremos en la sección de experimentación la diferencia de tiempos entre ése código y el de la versión final, donde *vuelta()* primero evalúa retornar una sola persona a la orilla origen.

2.4 Correctitud

Terminación: El algoritmo termina, dado que en el peor caso revisa todos los estados (cantidad finita). Un mismo estado sólo es analizado nuevamente siempre que se llegue en un tiempo menor que todas las veces anteriores, con lo que sólo pueden ser visitados una cantidad finita de veces al tener una cantidad finita de arqueólogos y caníbales, todos con tiempos positivos. Recorremos de a una rama a la vez hasta el final. Una vez que llegamos al final de la rama, retrocedemos un estado en la rama, llamémoslo E, y vemos si existe otra rama no analizada que se pueda bifurcar a partir de este estado. Si existe, la analizamos de la misma forma. Cuando terminamos de analizar la rama, al ir retrocediendo terminamos en el mismo estado E. Luego verificamos si quedan más ramas para repetir el proceso, o si no hay más ramas, retrocedemos en la rama hasta un estado E' y repetimos el proceso hasta analizar todos los estados, llegando hasta la raíz del árbol que se forma. Cuando desde la raíz no hay más bifurcaciones sin analizar, el algoritmo termina porque analizó todos los estados que podrían ser soluciones. Si hay estados que fueron podados, fue porque sabemos que no eran soluciones deseadas:

Las cotas de tiempos podan estados que se acceden en un tiempo mayor a una solución anteriormente encontrada. Por lo tanto, no evita que encontremos la solución óptima. Tampoco las podas que no nos permiten movernos a estados inválidos, pues la solución óptima requiere que todos los pasos sean válidos.

Reviso todas los estados alcanzables desde el estado inicial:
Quiero ver que puedo acceder a todo estado E_i que sea válido y alcanzable desde el estado inicial por una cantidad finita de pasos.

P(i): Se accede al estado E_i si éste es válido y existe al menos un estado E_{i-1} válido y alcanzable que difiera en una operación válida (teniendo en cuenta la ubicación de la linterna y la distribución de arqueólogos y caníbales). A partir de este estado E_{i-1} aplicamos un movimiento permitido para poder acceder a este estado E_i.

Caso base: El estado E_0 se accede dado que es el estado inicial, en el cual todos los arqueólogos y caníbales están del mismo lado y poseen la linterna. Dado que un estado inicial inválido no tiene sentido para el problema, asumiremos que siempre es válido.

Paso inductivo: P(i) → P(i+1): Tengo el conjunto de estados E_i. Son válidos y alcanzables por el algoritmo por la HI. A partir de aquí tenemos varias posibilidades:

- Si el conjunto de los estados E_i es vacío, entonces el estado E_{i+1} no es un estado válido, dado que no hay una combinación de movimientos válidos a partir del estado inicial con la cual llegar a este estado. Como no existe un estado anterior válidamente alcanzable, la propiedad P(i+1) resulta ser verdadera.
- Si el conjunto de estados no es vacío, entonces para todos los estados E_i puedo realizar alguna de las operaciones para mover algún arqueólogo o caníbal.
  - Si ningún estado puede realizar alguna operación que sea válida (es decir, que puedan trasladarse personas con la linterna y que los caníbales no devoren a algún arqueólogos), entonces el estado E_{i+1} no es un estado para el cual exista alguna secuencia de movimientos válidos de arqueólogos y/o caníbales para alcanzarlo. Como no se cumple esta propiedad, P(i+1) resulta ser verdadera.
  - Nombremos a la operación valida desde el estado E_i para llegar a E_{i+1} como Op_{i+1}. Llamemos también S_i a la secuencia de operaciones válidas que se utiliza para llegar al estado E_i, la cual existe por HI. Si agregamos Op_{i+1} al final de la secuencia S_i, obtenemos una secuencia de operaciones válidas desde el estado inicial hasta el estado E_{i+1} por la cual accederlo. De esta forma la propiedad P(i+1) resulta ser verdadera.

De esta forma probamos que alcanzamos a todos los estados. Por ende, si existe una solución válida la encontramos. Más aún, como podamos por tiempos quedándonos siempre la mejor solución que encontramos, siempre vamos a visitar el estado final que se consigue en el menor tiempo.

Para el caso que no exista solución, inicializamos ciertas variables en -1 para indicar que no hemos conseguido una solución. Si al final del algoritmo éstas variables no cambiaron, sabemos que no hay solución.

2.5 Complejidad del algoritmo

2.5.1 Complejidad demostrada

Cada estado tiene $\mathcal{O}(N+M)^2$ estados hijo (ignorando podas).

Como cada rama podría visitar cada uno de los estados, la altura máxima es $(N+M)!$, ignorando que siempre intentamos acercarnos a la solución en cada paso probando enviar 2 personas y devolver 1.

Podemos conseguir la cantidad de nodos de un árbol m-esimo de altura h con la fórmula

$$\sum_{i=0}^{h} m^i = \frac{1-m^{h+1}}{1-m}$$

La igualdad es válida por ser una serie geométrica.

Ésto hace que la cantidad de nodos del árbol de backtracking sea

$$\sum_{i=0}^{(N+M)!} (N+M)^{2i} \Rightarrow \mathcal{O}((N+M)^{2(N+M)!+2}) \Rightarrow \mathcal{O}((N+M)^{(N+M)!})$$

Cada nodo cuesta $\mathcal{O}((N+M)!)$ como fue demostrado en el pseudocódigo del algoritmo *ida()*

Resultado:

$$\mathcal{O}((N+M)^{(N+M)!} * (N+M)!)$$

2.5.2 Complejidad tomando en cuenta las podas

La complejidad teniendo en cuenta que la gran mayoría de los movimientos son de enviar 2 personas de ida y 1 de vuelta, con lo que la cantidad de estados que se recorrerán por rama (altura del árbol) es aproximadamente N+M:

$$\mathcal{O}((N+M)^{(N+M)} * (N+M)!)$$

2.6 Experimentación

Primero generamos una serie de tests para verificar el algoritmo. Éstos tests fueron creados a mano intentando cubrir la mayor cantidad de casos interesantes. El más destacable es el caso de 3 arqueólogos y 3 caníbales, que requiere que *vuelta()* envíe a la orilla origen 2 personas. También probamos quitar en *resolver()* la verificacion de if $(N < M \wedge N > 0)$ then return -1 y probar con esos casos. Como era de esperar éstos devolvieron "-1".

Luego armamos un generador de casos aleatorios para graficar los tiempos del algoritmo (en nanosegundos).

El generador recorre, para cada N+M, todas las combinaciones de N y M (N $\geq$ M). Para cada uno de éstas combinaciones se crean 10 problemas que se ejecutan 10000 veces para que la cache de la jvm no influya, y luego se toma el tiempo promedio de las siguientes 10000 ejecuciones. Para un mismo N+M se promedian todos los promedios y ése es el valor que se grafica.

En el primer gráfico se muestra el algoritmo donde *ida()* y *vuelta()* evalúan enviar primero 2 personas a la otra orilla. Ésta versión del algoritmo está claramente acotado por $\mathcal{O}((N+M)^{(N+M)} * (N+M)!)$. Aún más exacto es acotar el algoritmo por $\mathcal{O}((N+M)^{(N+M)})$. Ésta cota sale de los experimentos, mostrando que nuestras podas mejoran mucho el algoritmo respecto al peor caso demostrado (sin podas).

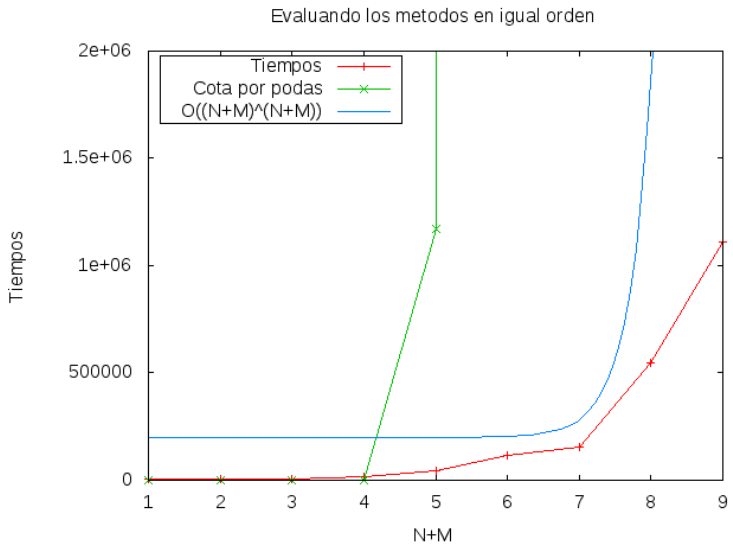


Figure 1: Tiempos evaluando en el mismo orden la forma de enviar personas a la otra orilla

En el segundo gráfico podemos observar la diferencia de desempeño entre el algoritmo utilizado en el gráfico anterior, y la versión final del mismo donde la única diferencia es que **vuelta()** primero evalúa enviar 1 persona a la orilla origen.

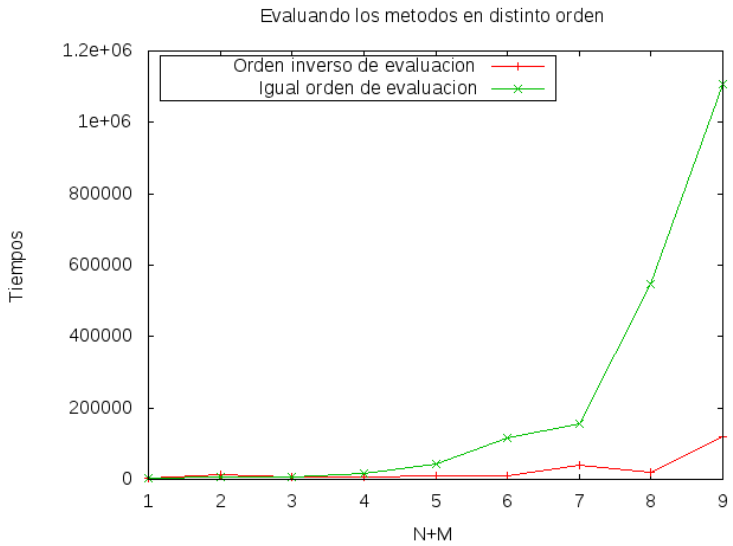


Figure 2: Diferencia de tiempos entre evaluar los métodos de la misma forma o discriminar entre ida() y vuelta()

Para mayor detalle, agregamos gráficos fijando N y variando M y viceversa, y los comparamos con los promedios antes tomados.

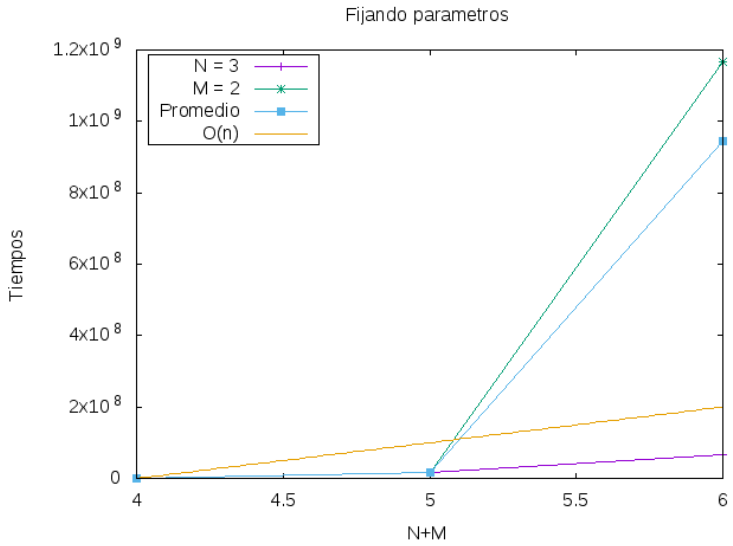


Figure 3: Diferencia para un mismo N y M respecto al promedio de todas las combinaciones válidas

Podemos ver como para un mismo N+M, se ve más influyente la cantidad de arqueólogos que la cantidad de caníbales. Entendemos que esto ocurre porque cuando N y M son iguales, las operaciones válidas para los estados son menos que las operaciones válidas cuando N es mayor a M. Y como devolvemos en $\mathcal{O}(1)$ que una operación no es válida, la curva de N=3 es mucho menor que la curva de M=2

3 Problema II: Problemas en el camino

3.1 Introducción

En este ejercicio, el grupo de arqueólogos necesita abrir una puerta cerrada con llave. En la misma habitación hay una balanza con dos platillos, y sobre uno de los platillos se encuentra la llave de la puerta. Al sacar la llave de la balanza, ésta cambia su equilibrio, la puerta se cierra y las paredes comienzan a moverse hacia ellos con el fin de aplastarlos. Para frenar las paredes y poder usar la llave, los arqueólogos tienen que volver la balanza a su equilibrio inicial utilizando un conjunto de pesas (ya que necesitan la llave como para volverla a colocar en la balanza), que tienen como peso una potencia de tres (no hay pesas del mismo peso). Para la suerte de los arqueólogos, uno de ellos tiene una balanza digital que les indica el peso de la llave que deben restablecer.

Dicho de forma más formal, dado el peso de la llave, se debe restablecer la balanza, de forma tal que la sumatoria de los pesos que se coloquen en el platillo donde estaba la llave, restada a la sumatoria de los pesos que se coloquen en el otro platillo, sea equivalente al peso de la llave.

3.2 Resolución del problema

Como los pesos en ambos platillos son sumatorias de potencias de 3, decidimos representar el peso de la llave como un número escrito en base 3. Este número nos dice que el valor a_i del número, siendo i la posición del valor comenzando a contar desde el valor menos significativo, es la constante que multiplica a la potencia 3^i, el cual puede ser 0, 1 *o* 2. Estas constantes nos van a servir para saber qué pesas tenemos que utilizar en qué balanza. Por cada dígito de la representación del peso de la llave en base 3 tenemos que tomar una decisión según su valor. Entonces tenemos 3 casos distintos:

- $a_i = 0$**:** Este caso indica que no tenemos que poner ninguna pesa en ningún platillo. Como esta potencia de 3 no esta incluida en la sumatoria de potencias de 3 que representa al peso de la llave, no tenemos que agregarla a la balanza.

- $a_i = 1$**:** Este caso indica que tenemos que poner la pesa de peso 3^i en el platillo donde se encontraba la llave. Esto se debe a que la potencia 3^i se encuentra en la sumatoria de potencias de 3 que representa al peso de la llave.

- $a_i = 2$**:** Este caso indica que deberíamos agregar peso para tratar de compensar la sumatoria, pero como solo tenemos un peso de cada potencia de 3, no podemos agregar 2 pesos en el platillo en que estaba la llave. Por eso es que agregamos el peso 3^i en el otro platillo y sumamos el valor 3^i representado en base 3, al peso de la llave representado también en base 3. Esto equivale a sumarle 1 a a_i, que quedaría en 0 y se le suma 1 al valor a_{i+1} (incluyendo también que todos los valores siguientes de la representación sean menores a 3).

  Sabemos que esto funciona por la siguiente igualdad: $3^{i+1} = 3 * 3^i = 2 * 3^i + 3^i$. En términos de nuestro problema, $2 * 3^i$ es el valor que nosotros tenemos y no podemos poner el el platillo de la llave. Entonces si le sumamos el término 3^i (es decir, ponemos el peso 3^i en el otro platillo para aumentar el peso que tenemos que poner en el platillo de la llave), necesitaríamos agregar el peso 3^{i+1} al platillo de la llave.

Como se puede observar en el último de nuestros casos, para evitar problemas por sumarle 1 al valor a_{i+1}, tenemos que analizar este valor luego de analizar todos los valores menos significativos antes para tener los valores posteriores que serán usados efectivamente.

Sabiendo como operar con cada dígito de la representación en base 3 del peso de la llave, procedemos a armar nuestro algoritmo. Éste consiste en iterar sobre cada dígito de la representación, del valor menos significativo hasta el más significativo, y para cada dígito realizar alguno de los 3 casos descriptos previamente.

Para facilitar los cálculos, cuando "nos llevamos una", establecemos una variable booleana como verdadera para indicarnos que en la siguiente iteración tenemos que sumarle 1 al valor.

A continuación incluimos el pseudocódigo del algoritmo:

3.3 Pseudocódigo

Algorithm 5 resolver

```
1: procedure RESOLVER(int peso_llave) → List<int> platillo_llave, List<int> platillo_otro
2:     //Obtengo la representación en base 3. De valores menos significativos a más significativos.
3:     List<Integer> key_weight_base_3 ← toBase3(peso_llave);
4:
5:     //Booleano para saber si me lleve una o no.
6:     boolean iTookOne = false;
7:
8:     for (int i = 0; i < key_weight_base_3.length; i++) do
9:         //Obtengo el valor i de la representación de la base.
10:        int value = key_weight_base_3.get(i);
11:
12:        //Si me lleve una antes incremento el valor.
13:        if (iTookOne) then
14:            value++;
15:            iTookOne = false;
16:        end if
17:
18:        //Agrego las pesas en los platillos correspondientes.
19:        if (value == 0) then
20:            //No tengo que poner ninguna pesa.
21:
22:        else if (value == 1) then
23:            //Tengo que poner la pesa 3^i en el lado de la llave.
24:            platillo_llave.add(3^i);
25:
26:        else if (value == 2) then
27:            //Tengo que poner la pesa 3^i en el otro lado y me llevo una para compensar los pesos.
28:            platillo_otro.add(3^i);
29:            iTookOne = true;
30:
31:        else
32:            //No tengo que poner ninguna pesa. Me llevo una para compensar.
33:            //(valor == 3, entonces la consideramos como 0 y me llevo una).
34:            iTookOne = true;
35:        end if
36:    end for
37: end procedure
```

Algorithm 6 toBase3

```
1: procedure toBase3(int peso_llave) → List<int> base3
2:     while (peso_llave > 0) do
3:         //Obtengo cociente y resto.
4:         int quotient = peso_llave / 3;
5:         int remainder = peso_llave % 3;
6:
7:         //Actualizo el peso de la llave.
8:         peso_llave = quotient;
9:
10:        //Agrego el valor a la lista resultado.
11:        base3.add( remainder );
12:    end while
13:
14:    return base3;
15: end procedure
```

3.4 Correctitud

En esta sección explicaremos por qué el algoritmo funciona. Sabemos que todo número puede representarse como una sumatoria de potencias de una misma base b, donde cada potencia puede ser multiplicada por un número menor a b. Es decir, para nuestro caso podemos expresar el número X como una sumatoria de potencias de 3 multiplicadas por un valor $0 \leq a_i \leq 2$:

$$\sum_{0}^{\infty} a_i * 3^i = X$$

Si consideramos al término izquierdo de la igualdad como el platillo de la balanza donde se encontraba la llave, y al término derecho de la igualdad como el otro platillo de la balanza, nuestro algoritmo lo que hace es transformar la sumatoria en otra sumatoria de potencias de 3 pero el valor que multiplica a cada potencia va a ser solamente el valor 0 o 1. El valor X de este lado se utiliza simplemente para representar el equilibrio inicial en la balanza y simplificar las cuentas y explicaciones.

Esto lo logramos transformando a las potencias 3^i multiplicadas por el valor 2, en potencias 3^{i+1}. La transformación se basa en la igualdad $3^{i+1} = 3 * 3^i = 2 * 3^i + 3^i$. Usando esta igualdad podemos, escribir al término $2 * 3^i$ como 3^{i+1}, siempre y cuando compensemos el término 3^i en el otro lado de la igualdad.

Obtenido el término 3^{i+1}, tenemos 3 casos que analizar:

- **En la sumatoria original,** $a_{i+1} = 0$ **o** $a_{i+1} = 1$**:** En estos casos sumamos el término obtenido en la transformación, y obtenemos que el valor que multiplica a la potencia 3^{i+1} ahora es 1 o 2. Siendo estos dos los valores que quedan multiplicando a la potencia, podemos proceder a continuar con el algoritmo, el cual manejara a estos valores cuando le toque analizarlos.

- **En la sumatoria original,** $a_{i+1} = 2$**:** En este caso, al sumarle la potencia 3^{i+1} obtenida en la transformación, se obtiene el término $3 * 3^{i+1}$, que es equivalente a 3^{i+2}. A este nuevo valor lo podemos sumar al término $i + 2$ de la sumatoria, teniendo que analizar nuevamente estos 3 casos para ese término.

Estas últimas operaciones terminan en algún momento, dado que no hay términos infinitos de la sumatoria que tengan $a_k = 2$ como valor que multiplica a la potencia 3^k.

Explicada la razón de porque transformar la sumatoria de la igualdad, nos falta justificar que al terminar de iterar los términos de la sumatoria desde el más chico hasta el más grande, no nos quedan potencias de 3 con el mismo exponente en ambos lados de la igualdad.

Esto se ve fácilmente de la siguiente forma: cuando estamos iterando el término i de la sumatoria, todos los términos anteriores (que van a ser potencias más chicas que la actual), ya fueron analizadas, y

los resultados de estas implicaron que no haya término 3^j $(j < i)$ en ningún lado de la igualdad, que el término 3^j esté del lado izquierdo de la igualdad con $a_j = 1$, o que 3^j esté del lado derecho de la igualdad también con $a_j = 1$. Esto nos asegura que al menos hasta la iteración i, tenemos a lo sumo solo una potencia 3^i entre ambos lados de la igualdad.

Con respecto a los término siguientes a i, como mantenemos los valores a_l $(l > i)$ iguales a 0, 1 *o* 2, siempre van a estar en condiciones de ser resueltos correctamente por iteraciones siguientes del algoritmo.

Una vez que el algoritmo termina, lo que obtenemos es una igualdad de la forma:

$$\sum_{0}^{\infty} a_i * 3^i = X + \sum_{0}^{\infty} b_i * 3^i$$

para las cuales $0 \leq a_i \leq 1$, $0 \leq b_i \leq 1$ y $a_i + b_i \leq 1$ (es decir que sólo esta la potencia $1 * 3^i$ en a lo sumo uno de los lados de la igualdad).

Como dijimos al comienzo, el lado izquierdo se corresponde con el platillo donde se encontraba la llave, y el lado derecho de la igualdad se corresponde con el otro platillo. Entonces, por cada término $1 * 3^i$ del lado izquierdo de la igualdad, tenemos que poner la pesa del mismo valor en el platillo. De forma análoga para el lado derecho, agregamos los valores $1 * 3^i$ de la sumatoria como pesas del mismo valor en el platillo.

3.5 Complejidad del algoritmo

Comenzaremos por la complejidad de la función toBase3(peso). Esta función devuelve la representación en base 3 del número pasado como parámetro y consiste en un ciclo donde las operaciones que se hacen tienen tiempo constante. Estas operaciones son de dividir el número por 3 hasta que el resultado sea 0 y obtener el resto, valor que será un elemento de la representación en base 3 del número. Entonces la complejidad de la función será la cantidad de veces que itere el ciclo. La cantidad de iteraciones que realiza será luego la parte entera superior del logaritmo en base 3 del peso. En la siguiente ecuación buscamos la cantidad de veces que debe iterarse la división para que peso sea menor a 1 (es decir, sea 0):

$$peso/(3^x) < 1$$

$$peso < 3^x$$

$$log_3(peso) < log_3(3^x)$$

$$log_3(peso) < x$$

Siendo X la cantidad de veces que dividimos el peso por 3, se ve que X tiene que ser mayor al log_3 del peso. Como X es un valor entero, nuestro X será el valor entero superior de $log_3(peso)$.

Como la cantidad de elementos de la lista retornada es uno por cada iteración, la cantidad va a ser también el valor entero superior inmediato a $log_3(peso)$.

Sabiendo esta complejidad, podemos explicar la complejidad de la función resolver(peso). Como se puede observar en el pseudocódigo, todas las operaciones que se realizan son operaciones de tiempo constante. Entonces lo que va a determinar la complejidad del algoritmo es la complejidad de la llamada a la función toBase3(peso) y la cantidad de iteraciones que realiza el ciclo. Este ciclo itera sobre cada uno de los elementos de la representación del número devuelta por la función toBase3(peso). Es decir que va a iterar $log_3(peso) + 1$ veces, una por cada dígito de la representación en base 3 de nuestro número, con todas operaciones constantes en cada iteración.

Sumando las complejidades importantes, tenemos por la llamada a toBase3 la complejidad $log_3(peso)$, y por el ciclo tenemos la misma complejidad.

Luego, la complejidad del algoritmo va a ser $O(log_3(peso))$. Como esta complejidad es menor a $\sqrt{peso}$, cumple con la condición pedida de que la complejidad debe ser menor a $O(\sqrt{peso})$.

Mostremos ahora que $O(log_3(x))$ es menor a $O(\sqrt{x})$.

$$log_3(x) < \sqrt{x}$$
$$0 < \sqrt{x} - log_3(x)$$

Derivando $f(x) = \sqrt{x} - log_3(x)$ obtengo :

$$f'(x) = 1/2\sqrt{x} \ - \ 1/x.ln(3)$$
$$f'(x) = (1 - 2/\sqrt{x}.ln(3)) \ /2\sqrt{x}$$

Como puede observarse la derivada es positiva cuando $1 - 2/\sqrt{x}.ln(3)$ lo es. Luego:

$$1 - 2/\sqrt{x}.ln(3) > 0$$
$$1 > 2/\sqrt{x}.ln(3)$$
$$\sqrt{x} > 2/ln(3) \approx 1.82$$

Y esta desigualdad se cumple cuando $4 \leq x$.
De esta forma obtenemos que $f(x) = \sqrt{x} - log_3(x)$ es creciente $\forall x \geq 4$. Calculando $f(4)$:

$$f(4) = \sqrt{4} - log_3(4) \approx 2 - 1.26185950714291 > 0 \ \forall \ x \geq 4$$

y como $f(x)$ es creciente, tenemos que

$$f(x) > 0 \ \forall x \geq 4$$
$$\sqrt{x} - log_3(x) > 0 \ \forall x \geq 4$$
$$\sqrt{x} > log_3(x) \ \forall x \geq 4$$

Analizando los valores $x = 1, 2, 3$:

$$\sqrt{1} = 1 > log_3(1) = 0$$
$$\sqrt{2} > 1.4 > 0.64 > log_3(2)$$
$$\sqrt{3} > 1.7 > log_3(3) = 1$$

Luego obtuvimos:

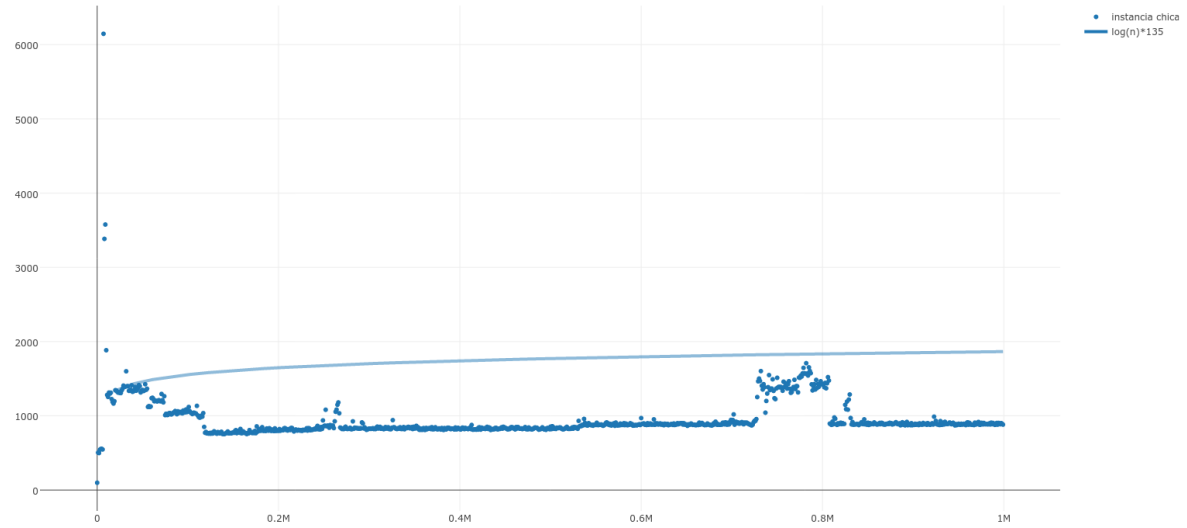
$$\sqrt{x} > log_3(x) \ \forall x > 0$$

Y por ende:

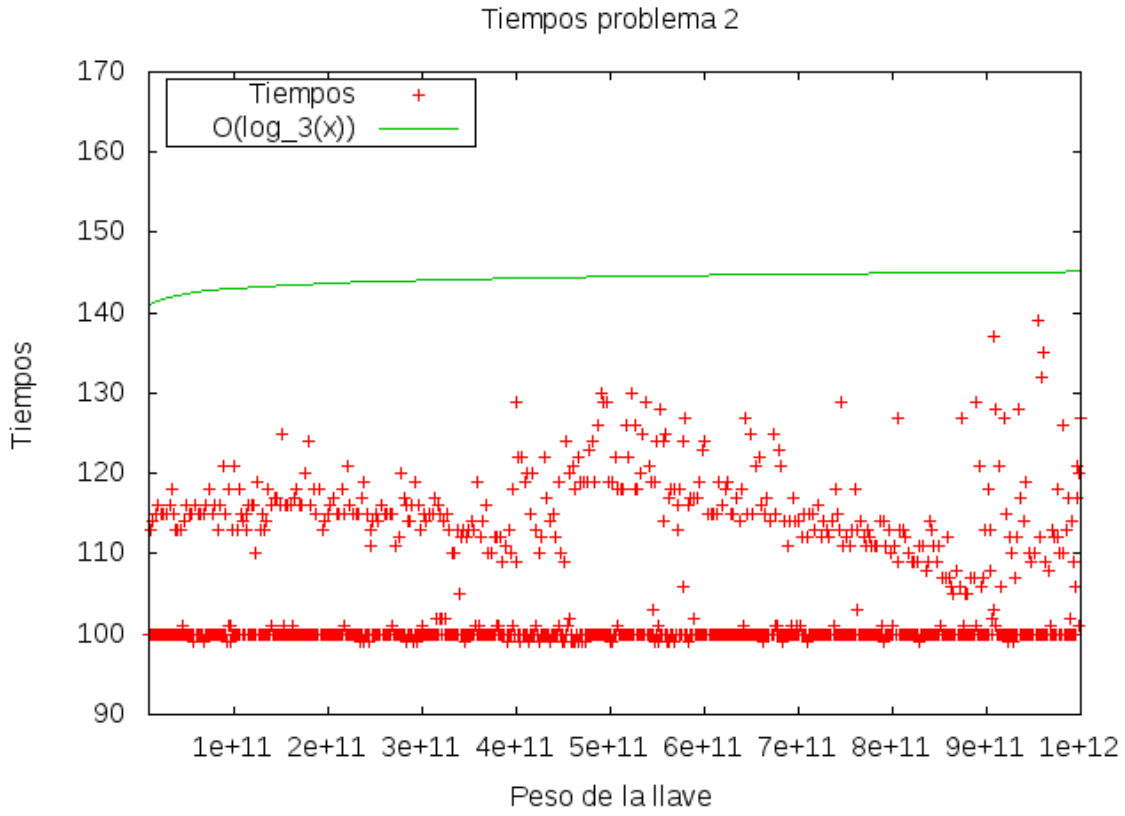
$$O(log_3(x)) esmenorque O(\sqrt{x})$$

3.6 Experimentación

Como el único parámetro de entrada del algoritmo es P, como experimentación correremos el programa variando el P en instancias pequeñas e instancias grandes, dado que P puede tomar valores de hasta 10^{15}.



Si bien en el gráfico los resultados estan por debado de la función log(P).350, se puede observar que el crecimiento del tiempo de la ejecución de las instancias es muy leve. Esto se debe a que en realidad el algoritmo tiene como complejidad al entero superior próximo de $log_3(x)$. Además, al ser un logaritmo, la instancia más grande da que $log_3(1.000.000) \approx 12.6$, y para una instancia no tan chica, digamos 100.000, tenemos que $log_3(100.000) \approx 10.5$. Por lo que al ser tan pequeña esta diferencia, ya que los valores enteros de los logaritmos varían entre 11 y 13 (por ser el entero superior mas próximo), la complejidad empírica obtenida pareciera tener un comportamiento constante.
Las instancias van de 0 a 1 millón.



En el siguiente gráfico se testearon pesos más grandes desde 10^{10} hasta 10^{12} y se ve que se cumple con la cota propuesta. Como puede observarse, la mayoría de los tiempos obtenidos en los experimentos tienen un valor similar a la experimentación anterior. Nuevamente esto se debe a que la diferencia entre tiempos al ser una complejidad logarítmica entre distintos tamaños de instancia es muy chica: en estas instancias con valores elevados la curva logarítmica es tan leve que al tomar sus valores enteros superiores se comporta como una recta constante en este rango de instancias.

4 Problema III: Guardando el tesoro

4.1 Introducción

Una vez pasadas todas las pruebas, Indy y el grupo de arqueólogos se encuentran en una sala llena con N tipos de tesoros de los que se sabe su valor y su peso. El grupo desea saquear el templo maximizando el beneficio del botín que pueden meter en sus M mochilas de capacidades diferentes. Para decidir que tipos de tesoros tomar y en que cantidad, se pidió desarrollar un algoritmo que devuelva el valor del mejor botín y como deben estar distribuidos los tesoros en las distintas mochilas.

En las próximas secciones se usarán las siguientes notaciones para referir a distintas variables del problema:

- M = cantidad de mochilas disponibles
- N = cantidad de tipos de tesoros
- K_i = capacidad de la mochila i $(1 \leq i \leq M)$
- C_i = cantidad de tesoros del tipo i $(1 \leq i \leq N)$
- P_i = peso de cada tesoro i $(1 \leq i \leq \sum C_i)$
- V_i = valor de cada tesoro del tipo i $(1 \leq i \leq \sum C_i)$

4.2 Resolución del problema

La primer propuesta tomada en cuenta fue diseñar un algoritmo que haga uso de la técnica de backtracking para resolver problema. Sin embargo, la complejidad del mismo resultó ser $O(M * (\sum_{i=0}^{M} C_i)!)$. Esto se debía a que, por las características de la técnica usada, se probaba poner toda las combinaciones posibles de tesoros en cada mochila.

Dicha complejidad superaba en demasía la complejidad pedida por el enunciado por lo que se estudiaron las propiedades del algoritmo para implementar la mayor cantidad de podas posibles e intentar, de esta forma reducir el tiempo de ejecución.

Durante esté análisis se vio que muchos casos, llegado a cierto punto, debían recalcular subproblemas que ya habían sido calculados en otro momento y se decidió cambiar el backtracking por el uso de programación dinámica para evitar este overhead innecesario.

4.2.1 La función recursiva

Suponiendo que los tesoros del templo están organizados en un array de longitud $\sum C_i$, en la que cada posición del mismo contiene un único tesoro (pudiendo haber repetidos si hay mas de un tesoro de un mismo tipo), se propuso la siguiente función recursiva para calcular cual es el mejor valor posible del botín:

$$F(i, x_1, \ldots, x_n) = \text{Máximo valor posible del botín con } i \text{ tesoros y una mochila de capacidad } x_1, \text{ una de capacidad } x_2, \ldots, \text{y una mochila de capacidad } x_n$$

Sea $F^j(i) = F(i, x_1, \ldots, x_{j-1}, x_j - P_{i+1}, x_{j+1} \ldots, x_n)$ entonces se puede escribir F como:

$$F(i, x_1, \ldots, x_n) = \begin{cases} 0 & \text{si } i = 0 \vee (\forall\ 1 \leq j \leq n)\ x_i = 0 \\ F(i-1, x_1, \ldots, x_n) & \text{si } (\forall\ 1 \leq j \leq n)\ P_i > x_j \\ -\infty & \text{si } (\exists\ 1 \leq j \leq n)\ x_i < 0 \\ max \begin{Bmatrix} F(i-1, x_1, \ldots, x_M), \\ V_i + F^1(i-1), \\ \vdots \\ V_i + F^j(i-1) \\ \vdots \\ V_i + F^n(i-1) \end{Bmatrix} & \text{sino} \end{cases}$$

La función está divida en 4 casos:

- **Primer caso:** No hay tesoros que poner o todas las mochilas tienen capacidad cero. En este caso no hay botín posible y el mejor valor es 0.
- **Segundo caso:** El i-ésimo tesoro no entra en ninguna de las mochilas disponibles, entonces se debe conseguir el mejor botín sin tenerlo en cuenta.
- **Tercer caso:** Este es un caso especial al cual solo se llega si se intenta meter un tesoro cuyo peso es mayor a la capacidad de la mochila. En la función es tomado en cuenta para que en el cuarto caso, si sucede que una mochila no puede contener el tesoro el valor sea lo mínimo posible y no afecte al resultado real de la instancia del problema
- **Cuarto caso:** El i-ésimo tesoro entra en una o más mochilas. Se busca maximizar el valor del botín, por lo tanto se calcula el máximo entre el valor del botín resultante de no poner el tesoro y el máximo botín conseguido al poner el tesoro en alguna mochila.

Se debe notar que en el último caso se calculan los valores del botín resultantes de poner el tesoro en cada mochila, cada uno de estos cálculos es necesario ya que las instancias más chicas generadas al meter el tesoro en mochilas distintas son distintas.

Por último, si se desea que la función devuelva la solución al problema propuesto se la debe llamar de la siguiente forma $F\left(\sum C_i, K_1, K_2, \ldots, K_M\right)$

4.2.2 Desarrollo del algoritmo

Al momento de implementar la función se optó por resolverla con un algoritmo bottom-up que opere sobre los valores de una matriz multidimensional.

Dicha matriz, es una matriz de M + 1 dimensiones cuyos tamaños son las capacidades de las mochilas y la cantidad de tesoros incluidos en el botín. La celda $matriz[i_1, i_2, \ldots, i_{M+1}]$ representa el valor $F(i_1, i_2, \ldots, i_{M+1})$.

Se ve que la matriz generada es una matriz de $(\sum C_i) \times K_1 \times \cdots \times K_M$ celdas

De esta forma, todos las celdas cuyos índices son de la forma $[i, 0, 0, \ldots, 0]$ ó $[0, x_1, x_2, \ldots, x_m]$ (que corresponden al caso base de la función) son inicializadas con ceros. A partir de aquí, la matriz se recorre de manera incremental calculando los valores de cada celda a partir de los valores de las celdas calculadas anteriormente. Para aclarar esto, el algoritmo resuelve el valor de la celda $[j, j_1, \ldots, j_m]$ (que corresponde al valor $F(j, j_1, \ldots, j_m)$) usando los valores de las celdas $[j-1, j_1, j_2 \ldots, j_m], [j-1, j_1, j_2 - 1, \ldots, j_m], \ldots, [j-1, j_1, j_2, \ldots, j_{m-1} - 1, j_m], [j-1, j_1, j_2 - 1, \ldots, j_{m-1}, j_m - 1]$ (es decir, de cada una de las celdas superiores respecto a cada una de sus dimensiones) de acuerdo a lo indicado por la función F.

Una vez calculados todos los valores, solo debemos encontrar el camino desde $matriz\left[\sum C_i, K_1, K_2, \ldots, K_M\right]$ hasta $matriz[0, K'_1, \ldots, K'_2]$ que hace posible el valor guardado en la primer posición mencionada (los valores K'_j son las capacidades finales de las mochilas, es posible que el mejor botín encontrado deje, en alguna de ellas, espacio que no pueda ser llenado por ninguno de los tesoros restantes).

Para encontrar este camino se usan los siguientes criterios:

- Si la mejor solución encontrada para el i-ésimo tesoro es igual a la mejor solución encontrada para el $(i-1)$-ésimo tesoro, entonces el i-ésimo tesoro no pertenece al botín.
- Si lo primero no sucede, el tesoro pertenece al botín y debe ser puesto en algún mochila. Para identificar dicha mochila se debe probar meterlo en cada una de ellas hasta encontrar la celda cuyo valor sumado al valor del i-ésimo tesoro sea el valor del botín completo.

4.3 Pseudocódigo

Todos los algoritmos tienen acceso a *tesoros* la lista de tesoros y *mochilas* la lista de mochilas disponibles.

Algorithm 7 Resolver

```
1:  procedure RESOLVER
2:      dimensions ← [∑ C_i, K_1, ..., K_M]                          ▷ O(M + 1)
3:      matrix ← MultidimensionalMatrix(dimensions);                 ▷ O(∏ dimensions[i])
4:      index ← [1,0,...,0]                                          ▷ Hay M ceros O(M+1)
5:      while index[0] ≤ ∑ C_i do                                    ▷ Se repite ∏ dimensions[i] veces
6:          matrix.set(index, maxValue(matrix, index))               ▷ Setear la celda index de la matriz al
7:                                                                   ▷ valor correspondiente O(M)
8:          index ← nextIndex(matrix, index)                         ▷ O(1)
9:      end while
10:     index ← previousIndex(matrix,index);
11:     lootValue ← matrix.get(index);
12:     while index[0] > 0 do                                        ▷ Se repite ∏ dimensions[i] veces
13:         m ← whereToPutTreasure(matrix, index);       ▷ Conseguir la mochila donde poner el tesoro
    O(m)
14:         if m < 0 then
15:             index[0] ← index[0] - 1;                             ▷ No poner el tesoro
16:         else
17:             agregar(tesoros[index[0]], mochilas[m])     ▷ Agregar el tesoro a la m-ésima mochila. O(1)
18:             index[0] ← index[0] - 1;          ▷ Con estas dos líneas se mueve a la celda que representa
19:             index[m] ← index[m] - peso(tesoros[index[0]])        ▷ la instancia generada al poner el
20:                                                                  ▷ tesoro en la mochila
21:         end if
22:     end while
23: end procedure
```

El paso en el que se rellenan con ceros las posiciones indicadas por el caso base de la función recursiva es omitido pues se asume que todas los valores de la matriz son inicializados en cero.

Algorithm 8 maxValue

```
procedure MAXVALUE(MultidimensionalMatrix matrix, Integer[] index) → Integer
    auxIndex ← index
    auxIndex[0] ← auxIndex[0] - 1     ▷ Esta es la celda que corresponde a no poner el tesoro index[0]
    maxValue ← matrix.get(auxIndex)                          ▷ El valor del botín sin el tesoro
    for m = 1 to dimensiones(matrix) do                      ▷ Se repite M veces
        if index[m] >= peso(tesoros[index[0]]) then     ▷ Si la capacidad de la mochila m es suficiente
            auxIndex[m] ← index[m] - peso(tesoros[index[0]]);  ▷ Poner el tesoro en la mochila m O(1)
            newValue ← valor(tesoro[index[0]]) + matrix.get(auxIndex);  ▷ Conseguir el valor del botín
            if newValue > maxValue then      ▷ Si el valor que se consigue es mejor que el de los otros
intentos
                maxValue ← newValue;                         ▷ se setea como maximo
            end if
            auxIndex[m] ← auxIndex[m] + peso(tesoros[index[0]])      ▷ Sacar el tesoro de la mochila
        end if
    end for
    Devuelvo maxValue
end procedure
```

Algorithm 9 whereToPutTreasure

```
1: procedure WHERETOPUTTREASURE(MultidimensionalMatrix matrix, Integer[] index) → Integer
2:     auxIndex ← index
3:     auxIndex[0] ← index[0]-1
4:     actualValue ← matrix.get(index)
5:     maxValue ← matrix.get(auxIndex);
6:     if maxValue == actualValue then          ▷ Si el botín con el tesoro y el botín sin el tesoro
7:         Devolver -1                          ▷ tienen el mismo valor entonces omitirlo
8:     end if
9:     maxValue ← 0                             ▷ Setear el valor maximo a cero
10:    for m = 1 to dimensiones(matrix) do      ▷ Se repite M veces
11:        if index[m] ≥ peso(tesoros[index[0]]) then
12:            auxIndex[m] ← index[m] - peso(tesoros[index[0]])
13:            newValue ← valor(tesoro[index[0]]) + matrix.get(auxIndex);
14:            if newValue == actualValue) then
15:                whereToPut ← m
16:                maxValue ← newValue
17:            end if
18:            auxIndex[m] ← index[m]
19:        end if
20:    end for
21:    Devolver whereToPut
22: end procedure
```

Se puede ver que las funciones whereToPutTreasure y maxValues son similares. En ambas funciones se checkean los valores de las soluciones anteriores para decidir, en una, cual es el máximo valor de botín posible y, en la otra, en que mochila debo ubicar cierto tesoro para conseguir el botín que corresponde al valor ubicado en la celda pasada por parámetro.

4.4 Correctitud

El algoritmo propuesto puede ser dividido en partes:

1. El cálculo del valor del mejor botín
2. Calcular los tesoros que pertenecen a dicho botín y en que mochila deben ser guardados.

Como se explicó en la sección 4.3.3, la primer parte del algoritmo corresponde al cálculo de la función F para sus distintos parámetros de entrada.

4.4.1 Primer parte

Por inducción, se puede ver que el primer paso es correcto:

Casos Base:

- $\sum C_i = 0$:

  Al no haber tesoros que poner en las mochilas, no hay botín posible, por lo tanto no hay valor positivo para el mismo.

  En la implementación del algoritmo, la matriz es inicializada con todos sus valores en 0 pero no es modificada por el while correspondiente a esta parte del algoritmo. Por lo que $lootValue = 0$.

- $K_i = 0 \; \forall \; 1 \leq i \leq M$

  Todas las mochilas tienen capacidad cero y como el peso de cualquier tesoro es mayor a cero, entonces no existe un tesoro que entre en ninguna de ellas. Esto quiere decir que no hay botín posible, entonces, no hay valor positivo para el mismo.

Otra vez, se ve que el algoritmo respeta este caso, las iteraciones son realizadas, pero al momento de calcular el máximo valor del botín, se toma el valor de la solución anterior (que es 0) y se itera sobre todas las mochilas. Sin embargo, al tener todas ellas capacidad cero, en ninguna iteración se accede al if que remplaza el valor máximo por uno distinto. Luego, el resultado de esta operación termina siendo cero.

En un primer paso se calcula el valor de poner cero tesoros en las mochilas (este es el caso anterior) entonces este valor es cero. Luego, se trata de poner un tesoro pero, por las operaciones que realiza el algoritmo este valor será el mismo que el de la solución anterior entonces el mejor botín con un tesoro tiene valor 0.

En el n-ésimo paso, la solución será la misma que en el (n-1)-ésimo que, como se explicó, es 0. Esto implicaría que en el n-ésimo paso la solución seguirá siendo 0.

Por último, habiendo demostrado esto, se concluye que en último valor calculado $(M\,[\sum C_i, K_1, \ldots, K_M])$ es 0.

Paso inductivo:

Sea la hipótesis inductiva que el algoritmo es correcto para todas las instancias tal que hay $i-1$ tesoros, se puede deducir la correctitud del i-ésimo:

$$\begin{aligned} F(i, x_1, \ldots, x_M) &= max\left\{F(i-1, x_1, \ldots, x_M), V_i + F^1(i-1), \ldots, V_i + F^j(i-1), \ldots, V_i + F^n(i-1)\right\} \\ &= max\left\{F(i-1, x_1, \ldots, x_M), V_i + max\left\{F^1(i-1), \ldots, F^j(i-1), \ldots, F^n(i-1)\right\}\right\} \end{aligned}$$

Por la hipótesis inductiva, sabemos que $F(i-1, x_1, \ldots, x_M)$, $F^1(i-1), \ldots, F^n(i-1)$ son correctas.

Se puede ver que la función toma el máximo entre todas las posibilidades dados i tesoros con la configuración de mochilas dadas.

La función devuelve el máximo valor de entre no agregar el tesoro y agregarlo a alguna mochila. Por *HI*, el primer valor es correcto, veamos que el segundo valor lo es:

El segundo valor toma el máximo botín que se puede obtener si fuese que se reserva un lugar para el i ésimo tesoro en alguna mochila. Si $F^j(i-1)$ corresponde al mejor botín posible con esta característica, entonces el segundo valor es de la forma $V_i + F^j(i-1)$. Y si este no fuera el mejor valor posible, entonces existiría otro botin con valor $V_i + F^k(i-1)$, en este caso resultaria que $F^k(i-1) > F^j(i-1)$ pero esto sería absurdo por que $F^j(i-1)$ era el máximo.

El algoritmo propuesto realiza esta operación de la manera clásica, asigna un valor inicial a la variable $maxValue$ $(F(i-1, x_1, \ldots, x_M))$ y luego itera sobre las posiciones de la matriz que corresponden a intentar meter el botín en las distintas mochilas disponibles. Cuando se ve que, al ubicar el tesoro en cierta mochila, se consigue un valor mayor a $maxValue$, este es reemplazado por el nuevo valor encontrado.

Al finalizar las iteraciones, $maxValue$ contendrá el máximo valor posible del botín.

4.4.2 Segunda parte

Este paso consiste en reproducir las selecciones de los máximos elegidos en la sección anterior. Para esto, se checkean uno a uno los valores de cada instancia más chica del problema llamada en F.

La primer celda checkeada corresponde a la solución en la que el tesoro no es agregado y si el valor de esta solución y la solución que se esta buscando es igual, entonces el tesoro es omitido. Al resto de los valores se le suma el valor del tesoro que se intenta agregar, si el resultado es igual al valor de la celda que se está analizando entonces el tesoro se agrega a la mochila que corresponde al índice desplazado. Este desplazamiento corresponde al valor máximo elegido por la función F y estas operaciones se realizan de forma iterativa hasta que la celda analizada represente alguno de los casos bases de F.

Cuando el algoritmo finaliza, las mochilas contienen la sucesión de tesoros necesaria para poder lograr un botín de valor óptimo.

4.5 Complejidad del algoritmo

Al momento de calcular la complejidad de los algoritmos propuestos se asumió que el acceso a una posición de la matriz multidimensional es $O(1)$.

4.5.1 maxValue y whereToPut

En los algoritmos $maxValue$ y $whereToPutTreasure$, los for realizan M iteraciones en las cuales se accede y/o modifica la matriz y distintas variables enteras, cada una de estas operaciones es $O(1)$. En ambos casos los condicionales de los if dentro del for hacen comparaciones de enteros por lo que estas comparaciones se realizan en $O(1)$. Las asignaciones realizadas fuera del for son realizadas en $O(1)$, salvo la variable $auxIndex$ que es inicializada en $O(M+1)$.

Luego ambos algoritmos son de complejidad $O(M+M+1) \subset O(M)$.

4.5.2 resolver

En el algoritmo $resolver$, se crean 2 arrays (`dimensions` e `index`) de $M+1$ elementos cada uno ($O(M+1)$), luego se crea la matriz multidimensional que es $O\left(\prod \texttt{dimensions}[i]\right)$ y luego se ejecutan los dos $while$.

Con el primer $while$ se rellena cada posición de la matriz con el valor correspondiente, esto es hacer $\prod \texttt{dimensions}[i]$ iteraciones. En cada iteración se debe setear la celda correspondiente usando la función $maxValue$ ($O(M)$) y luego se aumenta el indice en 1 ($O(1)$). Entonces el $while$ completo tiene complejidad $O\left(M * \prod \texttt{dimensions}[i]\right)$

En el segundo, en peor caso, debemos volver a recorrer cada celda de la matriz por lo que se realizan la misma cantidad de iteraciones. En cada iteración se realiza una llamada a $whereToPut$ ($O(M)$) y a luego se calcula el proximo indice en $O(1)$, además, si es necesario, se agrega a una mochila un tesoro.

Las mochilas pueden ser vistas como listas enlazadas, por lo que agregarles un elemento es $O(1)$.

Por lo tanto el segundo while tiene la misma complejidad que el primero.

El algoritmo completo tiene complejidad

$$O\left(M+1+M*\prod \texttt{dimensions}[i]+M*\prod \texttt{dimensions}[i]\right) \subset O\left(M*\prod \texttt{dimensions}[i]\right)$$

Ahora, `dimension[0]` es la cantidad de tesoros en la sala, es decir $\sum_{i=1}^{N} C_i$, entonces:

$$M*\prod_{i=0}^{M} \texttt{dimensions}[i] = M*\left(\sum_{i=1}^{N} C_i\right)*\prod_{i=1}^{M} \texttt{dimensions}[i]$$

Y cada uno de los indices siguientes, corresponden a los pesos de los mochilas, es decir $indice[i] = K_i$ para $1 \leq i \leq M$ y vale:

$$M*\left(\sum C_i\right)*\prod_{i=1}^{M} \texttt{dimensions}[i] = M*\left(\sum C_i\right)*\prod_{i=1}^{M} K_i$$

Luego el algoritmo completo es de complejidad $O(M*\left(\sum_{i=1}^{N} C_i\right)*\prod_{i=1}^{M} K_i)$, solo falta ver que dicha complejidad es menor a la requerida por el enunciado, es decir se necesita comprobar que:

$$O\left(M*\left(\prod_{i=1}^{M} K_i\right)*\left(\sum_{i=1}^{N} C_i\right)\right) \subset O\left(\left(\sum_{i=1}^{M} K_i\right)^{M}*\left(\sum_{i=1}^{N} C_i\right)\right)$$

Para ver que esto es cierto basta con probar la siguiente desigualdad:

$$M*\left(\prod_{i=1}^{M} K_i\right)*\left(\sum_{i=1}^{M} C_i\right) \leq \left(\sum_{i=1}^{M} K_i\right)^{M}*\left(\sum_{i=1}^{N} C_i\right)$$

Como $\sum_{i=1}^{N} C_i \neq 0$:

$$M*\prod_{i=1}^{M} K_i \leq \left(\sum_{i=1}^{N} K_i\right)^{M}$$

Por la fórmiula del coeficiente multinomial se sabe que:

$$\left(\sum_{i=1}^{M} K_i\right)^M = \sum_{J_1+\ldots+J_M=M} \binom{M}{J_1,\ldots,J_M} * \prod_{t=1}^{M} {K_i}^{J_t} =$$

$$M * \prod_{i=1}^{M} K_i + \sum_{J_1,\ldots,J_M : J_1+\cdots+J_M=M \wedge \neg(\forall 1 \leq i \leq M) J_i = 1} \binom{M}{J_1,\ldots,J_M} * \prod_{t=1}^{M} {K_i}^{J_t}$$

El primer término mostrado en la última igualdad surge de elegir todos los exponentes iguales a 1. Por esta razón se agrega la condición $\neg(\forall 1 \leq i \leq M) J_i = 1$ al segundo término.

En la inecuación, entonces se remplaza:

$$M * \left(\prod_{i=1}^{M} K_i\right) \leq M * \prod_{i=1}^{M} K_i + \sum_{J_1,\ldots,J_M : J_1+\cdots+J_M=M \wedge \neg(\forall 1 \leq i \leq M) J_i = 1} \binom{M}{J_1,\ldots,J_M} * \prod_{t=1}^{M} {K_i}^{J_t}$$

$$0 \leq \sum_{J_1,\ldots,J_M : J_1+\cdots+J_M=M \wedge \neg(\forall 1 \leq i \leq M) J_i = 1} \binom{M}{J_1,\ldots,J_M} * \prod_{t=1}^{M} {K_i}^{J_t}$$

Y como todos los elementos de esta sumatoria son enteros positivos (por ser todos los K_i positivos), queda demostrado que la complejidad que proponemos es menor a la pedida por el enunciado.

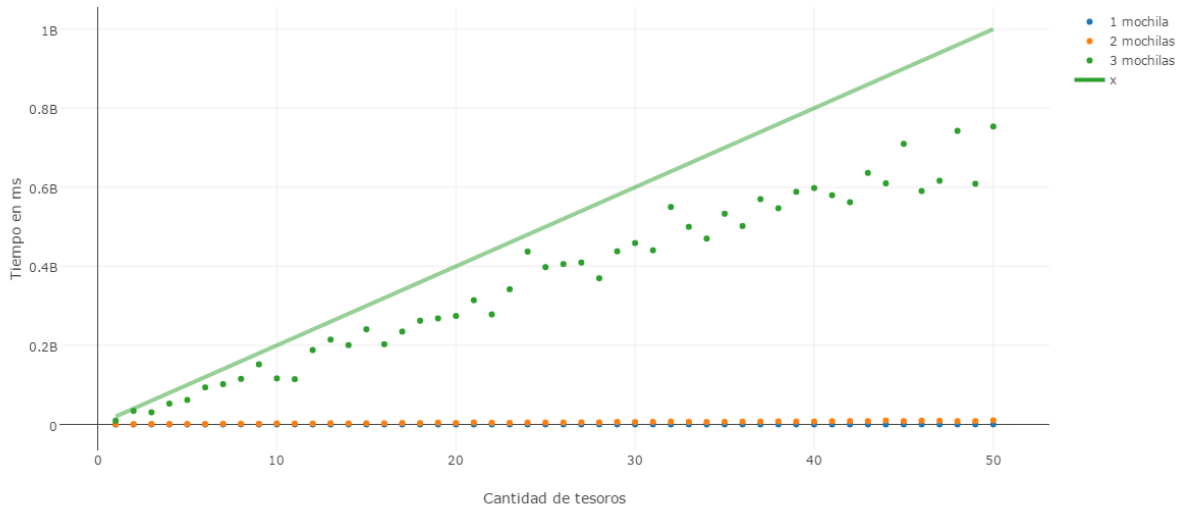
4.6 Experimentación

Para los experimentos se decidió analizar instancias se varían los tamaños de las mochilas y la cantidad de tesoros por separado. En ambos experimentos, las variaciones se prueban para 1, 2 y 3 mochilas.

En el primero de los casos, se fijó el tamaño de las mochilas en 50 unidades cada una y se generaron instancias de problemas con cantidades de tesoros variante entre 1 y 100. El precio y valor de cada uno de los tesoros fue generado aleatoriamente entre los valores 1 y 100.

Se debe notar que los valores y pesos de los tesoros no influyen en la complejidad del algoritmo por lo que no se consideró necesario evaluar instancias con valores demasiados granes.

Los resultados obtenidos se muestran en los siguiente gráficos:



Se puede ver que para los 3 casos probados el crecimiento temporal es lineal, lo que corresponde con la complejidad teórica calculada cuando la cantidad de mochilas y sus capacidades son tomadas como constantes. Y además, se nota que al aumentar la cantidad de mochilas los tiempos para instancias

con la misma cantidad de tesoros incrementan notablemente debido a que la constante se ve afectada exponencialmente por la cantidad de mochilas.

$$O\left(\left(\sum_{i=1}^{1} 50\right)^{1} * \left(\sum_{i=1}^{N} C_i\right)\right) = O\left(\sum_{i=1}^{N} C_i\right) = O(X)$$

$$O\left(\left(\sum_{i=1}^{2} 50\right)^{2} * \left(\sum_{i=1}^{N} C_i\right)\right) = O\left(\sum_{i=1}^{N} C_i\right) = O(X)$$

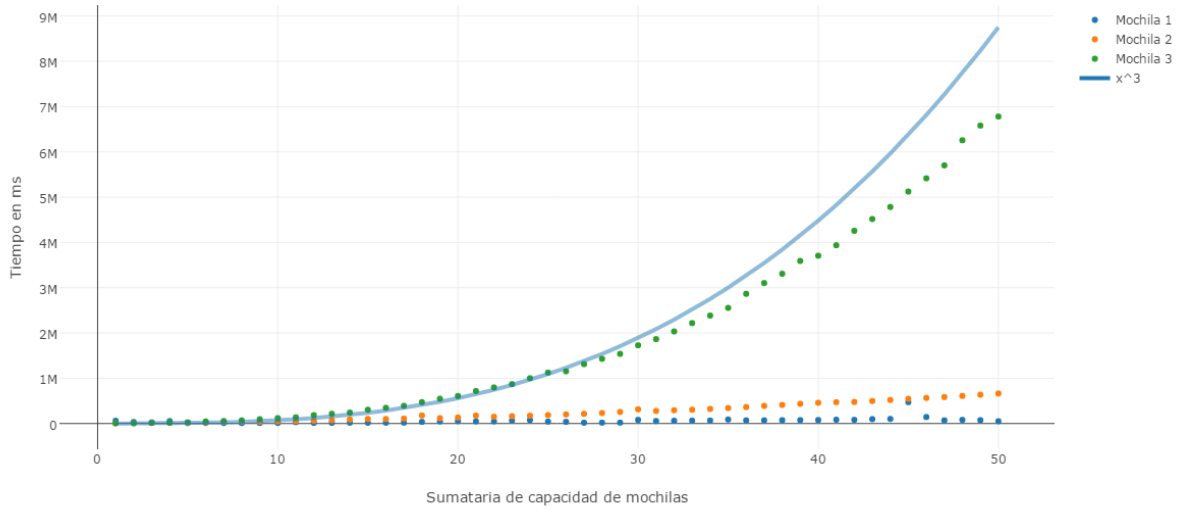
$$O\left(\left(\sum_{i=1}^{3} 50\right)^{3} * \left(\sum_{i=1}^{N} C_i\right)\right) = O\left(\sum_{i=1}^{N} C_i\right) = O(X)$$

En el siguiente, se ha realizado un experimento similar anterior diferenciándose de que todas las instancias los tesoros van a ser los mismos y se varían las capacidades de las mochilas.

Los tesoros elegidos fueron:

Cantidad	Peso	Valor
1	30	14
3	35	10
4	40	30
2	45	4
1	50	20
5	50	60

En este caso, el polinomio que acota temporalmente a los tiempos calculados está dado por la cantidad de mochilas. En el siguiente gráfico se observa que para 1 mochila, los tiempos se incrementan de forma lineal, mientras que en las instancias de 2 mochilas su aumento es cuadrático y en las de 3 mochilas es cúbico.



De esta forma se corrobora que la complejidad se cumpla, ya que la complejidad experimental de cada caso termina siendo: Sea K_i al peso de la i-ésima mochila:

$$O\left(\left(\sum_{i=1}^{1} K_i\right)^{1} * \left(\sum_{i=1}^{6} C_i\right)\right) = O\left(\left(\sum_{i=1}^{1} K_i\right)^{1} * (16)\right) = O(K_1)$$

$$O\left(\left(\sum_{i=1}^{2} K_i\right)^{2} * \left(\sum_{i=1}^{6} C_i\right)\right) = O\left(\left(\sum_{i=1}^{2} K_i\right)^{2} * 16\right) = O\left(\left(\sum_{i=1}^{2} K_i\right)^{2}\right)$$

$$O\left(\left(\sum_{i=1}^{3} K_i\right)^3 * \left(\sum_{i=1}^{6} C_i\right)\right) = O\left(\left(\sum_{i=1}^{3} K_i\right)^3 * 16\right) = O\left(\left(\sum_{i=1}^{3} K_i\right)^3\right)$$

5 Hoja de cambios:

5.1 Problema 1

- En la tabla de ejemplos de la sección 2.1.1 se cambió el nombre de Puente por Movimiento, para diferenciar más el puente del problema original con el objeto Puente
- Se mejoraron los textos de Resolución del problema, Algoritmos y Complejidad.
- Se agregó el algoritmo *enviarArq* y se expandió el pseudocódigo de *prueboEnviar*2*Arqueologos* para mayor claridad.
- Se explicó el cambio en la complejidad de algunos algoritmos, que en el cálculo se tiene en cuenta la recursión pero para otras operaciones no.
- Se dio una mejor explicación de porqué el programa termina y una demostración de porqué el algoritmo recorre todos los estados válidos.
- Se explicó porqué la altura máxima del árbol es de $\mathcal{O}((N+M)!)$
- Se agregó la descripción del generador de casos aleatorios para graficar los tiempos del algoritmo. Además, se agregó un gráfico fijando N en un caso y M en el otro.

5.2 Problema 2

- Se corrigió la demostración de la complejidad.
- Se explicó de mejor manera los resultados obtenidos en la experimentación realizada.

5.3 Problema 3

- Se explicó de mejor manera función recursiva
- Se cambió el paso inductivo de la demostración de correctitud para que sea más entendible.