DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Nicolás Bukovits	546/14	nicobuk@gmail.com
Kevin Frachtenberg	247/14	kevinfra94@gmail.com
Julián Len	467/14	julianlen@gmail.com
Nicolás Len	819/11	nicolaslen@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. Introducción

En este trabajo se tienen tres problemas que se resolvieron aplicando las técnicas algorítmicas estudiadas en la materia. Los mismos presentaron cada uno un desafío distinto, teniendo que aplicar métodos diferentes para resolverlos y cumplir con los requisitos exigidos.

Además de la resolución de los mismos, se procedió a demostrar la correctitud de cada implementación. Esto fue acompañado a su vez de una justificación de la complejidad temporal.

Cada ejercicio contó con su respectiva experimentación para corroborar que la complejidad temporal teórica se cumpliera y en los casos donde el algoritmo podía comportarse mejor, verlo reflejado de alguna manera.

Los experimentos contaron con diversas medidas para asegurar su efectividad de las cuales las siguientes fueron iguales para los tres problemas:

- Sólo se midió el costo temporal de generar la solución, no de lectura y escritura del problema.
- Para la medición del tiempo se utilizó la biblioteca `chronos` con unidad de tiempo en microsegundos.

2. Ejercicio 1: Vamos a buscar la respuesta a P=NP!

2.1. Descripción del problema

Figura 1: Indiana Jones junto a un Canibal del grupo de exploración

Indiana Jones debe seguir un mapa que posiblemente lo lleve a encontrar la solución a P=NP. Para esto, lleva a un grupo de arquéologos compuesto por A personas y le pide ayuda a un grupo de gente local de tamaño C para poder llegar hasta su destino sin grandes problemas. Sin embargo, durante el camino se encuentran con un puente en mal estado en el que no podrán pasar más de dos personas a la vez. Sumado a eso, hay solo una linterna para todo el equipo, por lo que en cada cruce alguien debe volver con la linterna. Como si el problema del puente fuera poco, el grupo local es conocido por su canibalismo, así que no pueden quedar más caníbales que arqueólogos de alguno de los lados del puente.

La resolución del problema consiste en elaborar un programa que recibe como entrada el valor de A y C, y luego las velocidades de cada arqueólogo $(a_0, ..., a_A)$ y la de cada canibal $(c_0, ..., c_C)$ y devuelve la velocidad mínima con la que se puede cruzar el puente.

Por ejemplo, si el programa recibe lo siguiente como entrada:

```
2 1
1 2
1
```

La salida correcta sería:

```
4
```

2.2. Solución propuesta

La solución de este problema fue lograda considerando todos las formas posibles de cruzar el puente. Para esto, el algoritmo propuesto chequea todas las posibilidades que se tienen para ir de un lado al otro del puente. Es decir, sea $A1$, $A2$ y $C1$, $C2$ dos personas de cada grupo, se elige una de las formas que hay de cruzar el puente, las cuales consisten en que cruce alguno de los elementos {A1, C1, (A1,C1), (A1,A2), (C1,C2)}. Para cada una de esas opciones, se intentara realizar el cruce $\forall a \in$ *arqueólogos* y $\forall c \in$ *caníbales*. En otras palabras, se realiza el intento para todas las combinaciones de personas que hay.

Además, lo que tomamos como velocidad en cada cruce es el tiempo que tarda pasar un grupo o persona desde un lado del puente a otro. Es decir, sean $C = \{c1, c2, c3\}$, conjunto de caníbales y sea $A = \{a1, a2, a3\}$ de arqueólogos, cada uno con su respectiva velocidad. Sean $x, y \in$C$\cup$A. Entonces la velocidad de pasar x e y es del max(V(x), V(y)) con V la funcion que da la velocidad de una persona.

Solamente van a tener la chance de cruzar quienes estén del lado en que está en la linterna y que su cruce no lleve a un *estado inválido*. Un estado válido es aquél en el que no se estuvo anteriormente (con respecto a la cantidad de caníbales y arqueólogos de cada lado y a la ubicación de la linterna) y que no deje a más caníbales que arqueólogos de alguno de los dos lados.

Una solución es válida cuando lograron cruzar todos las personas (arqueólogos y caníbales) de un lado al otro del puente. Esto es posible si no hubo estados inválidos en el camino para que todos crucen. Dado que se prueban todas las combinaciones posibles, vamos a obtener múltiples soluciones de distintas velocidades. Es por ello que una vez que se tengan todos las soluciones posibles, se compararán las velocidades de cada solución y se tomará la mínima.

Teniendo en cuenta lo planteado en este informe sobre el problema, podemos marcar que el algoritmo realizado fue construido en base a la técnica de *backtracking* que al igual que en este caso, consiste en probar todas las posibilidades descartando la mayor cantidad de soluciones incorrectas posibles al mismo tiempo y dejando como resultado una lista con las soluciones válidas. Así, se prueban todos los caminos posibles para cruzar el puente y al final se obtiene una lista con todos los tiempos que puede tomar cruzar el puente (excepto el caso donde no haya ningún caso posible, que devolvemos -1).

Más específicamente, la aplicación de *backtracking* en este problema consiste en probar todos los idas y vueltas de todas las combinaciones de personas, pero sumado a que la poda utilizada es la de no repetir estados, cuya correctitud será explicada posteriormente. El algoritmo tiene forma recursiva que depende de la cantidad de arqueólogos y caníbales de cada lado y la unicación de la linterna (al igual que la validez de los estados).

```
BTCruzarPuentes(Parámetros)
si estadoActual tiene linterna a la derecha
  lado de origen  = lado derecho
  lado de destino = lado izquierdo
si no
  lado de origen  = lado izquierdo
  lado de destino = lado derecho
linternaEnDeracha = !linternaEnDerecha

esSolucion = canibales_izq + arquielogos_izq == 0;
Si es solucion
    encolar tiempo en soluciones
} sino


    for #mandarCanibales = 0 ... minimo(2, #canibales del lado de origen):
      for #mandarArqueologos = 0 ... (2 - #CanibalesQueCruzan):
        if esEstadoValido(#CanibalesEnOrigen - #canibalesQueCruzan,
                          #ArqueologosEnOrigen - #arqueologosQueCruzan,
                          #CanibalesEnDestino + #CanibalesQueCruzan,
                          #ArqueologosEnDestino + #ArqueologosQueCruzan,
                          linternaEnDerecha, estadosAnteriores):



          switch((#mandarCanibales,#mandarArqueologos)
              moverEsaCantidad
              cambiarDeLadoLinterna
              guardarEstadoNuevo
              BTCruzarPuente
          end switch

        endif
      end for
    end for
  end if
```

Esta es una porción como pseudocódigo del algoritmo completo, en la cual se muestra cómo funciona la parte recursiva del mismo. Esta función, toma como parámetros de entrada *canibalesOrigen, arqueologosOrigen, canibalesDestino, arqueologosDestino,linternaDer, estadosAnteriores, tiempo, soluciones.*

El algoritmo, en primer lugar chequea si dentro del estado en el cual entró a la recursión, cruzaron todos los exploradores (arqueólogos y caníbales). En tal caso, se agrega el *tiempo* que tomó cruzar el puente a la variable *soluciones.* Luego, regresa de la recursión para volver hacia arriba en un nivel en el árbol de ejecución y se continúa probando las otras posibilidades de caminos.

Caso contrario, el algoritmo prueba todos los casos de mandar caníbales y/o arqueólogos por el puente (tomando en cuenta de mandar uno solo o dos). Por cada caso, si el movimiento es válido cada vez que un explorador es elegido, se crea un nuevo estado y es agregado a un vector de **estados**. La idea de agregar nuevoEstado a EstadosAnteriores consiste en poder eliminar los caminos que lleven a una situación que ya se haya estado con anterioridad (por ejemplo, que cruce un canibal y luego vuelva), para evitar loops infinitos. Además, al quitarla luego de la recursión hace que para cada altura del árbol de ejecución tengamos la misma cantidad de estados y que éstos sean todos distintos. La razón por la cual serán distintos proviene de que cada estado se arma basándose en la cantidad de arqueólogos y caníbales que hay de cada lado y luego sus velocidades, según de qué lado está la linterna. Entonces si cada vez que haya que cruzar el puente se elige una cantidad distinta de exploradores, cada nuevo estado tendrá como máximo 5 formas distintas (que cruce un solo canibal, un solo arqueólogo, dos canibales, dos arqueólogos o un arqueólogo y un canibal). Y por cada tipo de explorador elegido, se entra a la recursión con cada uno de los que se encuentran del lado de la linterna.

Finalmente, se devuelve el vector de *soluciones*, el cual contiene cada uno de los tiempos, y se devuelve el mínimo.

2.2.1. Detalles implementativos

El algoritmo fue implementado en lenguaje C++. Para almacenar la solución, se recurre a la clase `vector`, proporcionada por la librería estándar del lenguaje.

Para manejar los estados en el árbol de ejecución, se van almacenando los nuevos estados en un vector de `Estados` antes de entrar en una recursión y se quita al retornar de la misma. Esto es para que, si del estado S_i, se agrega un estado y se llega al S_{i+1}, si este es inválido, pueda regresar nuevamente a S_i, e intentar con otros estados. Además, esto sirve para que para el mismo nivel dentro del árbol de recursión, la cantidad de estados dentro de cada nodo interno u hoja, sea la misma.

Un `Estado` es una clase la cual consiste de 4 `Int`, 2 para la cantidad de arqueólogos y 2 para la cantidad de caníbales de cada uno de los lados en ese momento, y de un `Bool` para indicar si la linterna se encuentra a la derecha o no.

Llamamos *árbol de ejecución* al árbol que se va generando de acuerdo a las desiciones tomadas en cuanto a qué explorador(es) cruzará(n) el puente.

La forma en que se elige quiénes cruzarán de un lado a otro luego de haber decidido a qué grupo pertenecen, es iterar sobre el vector del grupo y lado correspondiente y probar todas las combinaciones de una o dos personas. Si bien se considera un *estado* a la cantidad de caníbales y arqueólogos que haya de cada lado y la ubicación de la linterna, es importante considerar que cualquier par de personas pueda cruzar (si van a cruzar dos o una si cruza una sola). Es por esto que se realizan ciclos que recorran a los vectores y se entra en una nueva recursión por cada persona que se decide enviar al otro lado.

2.3. Demostración de la correctitud

Para demostrar correctitud del algoritmo debemos demostrar por un lado, que utilizar la técnica backtracking cumple con recorrer todos los casos y devuelve el más rápido. Y por otro lado que la poda de evitar que se repitan casos es válida.

Esto último se puede ver en el algoritmo. Pues al generarse un estado nuevo, el mismo es guardado en un struct *Estados* o variable *nuevoEstado*, luego éste es usado como parámetro para la función *estadoValido*, la cual chequea en un vector de estadosAnteriores, si ese estado ya había sido alcanzado. Si esto pasa, el algoritmo no hace nada para ese caso y prueba mandar otro/s arqueólogo/s o caníbal/es según toque en el ciclo principal para encontrar un nuevo estado.

La función principal del algoritmo depende de la técnica de backtracking. En esta, intentamos probar cada uno de los casos posibles, descartando la mayor cantidad de casos inválidos posibles a la vez. Para esto último utilizamos la poda anteriormente mencionada. Esto es para, una vez recorrido todas las maneras de cruzar el puente, se llegue a (si hay o no) una solución, y en caso de que haya, se llegue a la más rápida.

Lo primero que se realiza es chequear que el estado no sea inválido. Es decir, que no haya más caníbales que arqueólogos o que haya 0 arqueólogos. Si esto se cumple pasa a la etapa de recursión. Caso contrario,

termina inmediatamente la ejecución devolviendo -1.

En la parte recursiva, se chequea si todas las personas pasaron al otro lado del puente. Si no, se realiza lo siguiente: Lo primero que realiza el algoritmo dado el caso donde no esté encontrada la solución, es fijarse los 5 casos posibles de mandar por el puente:

- 2 Arqueólogos
- 1 Arqueólogo y 1 Caníbal
- 2 Caníbales
- 1 Arqueólogo
- 1 Caníbal

Por cada uno de estos casos, si es viable mover esa cantidad sin entrar en un caso inválido (un caso donde ya se encontraba la misma cantidad de caníbales y arqueólogos de un lado del puente y del otro, en un estado anterior), continúa a su función de *mover* correspondiente, que es aquella que realiza la recursión. Caso contrario, prueba entrar a la recursión con algún otro caso.

Es decir sea E_i el i-ésimo estado válido. Al probar con el estado E_{i+1}, existen dos casos:

- E_{i+1} es válido. Esto significa que accede a su función correspondiente y es agregado a **estadosAnteriores**.
- E_{i+1} no es válido. Esto significa que $E_{i+1} \in$estadosAnteriores. Lo que realiza el algoritmo es volver al E_{i+1} y probar con el estado E_{i+2}.. Hasta agotar las 5 posibilidades.

Luego, esta función de *mover* mencionada, que realiza la recursión, es llamada una vez por cada uno de los 5 casos que sean validos. Cada una sirve para, dado con cual de los 5 casos posibles se entró a esta función específica, entrar nuevamente a la función recursiva con todas las combinaciones posibles de mandar a cada caníbal o arqueólogo disponible en el destino. De esta manera, no sólo se estaría abarcando todos los movimientos por el puente posibles, sino que también, la combinación de los integrantes (cada uno con su velocidad) de cada grupo de exploradores, en cada uno de los movimientos.

Por ejemplo si el movimiento es mandar 2 caníbales y este estado es válido. Entonces sean C1, C2 y C3 caníbales, las posibles combinaciones para envíar del otro lado del puente serían (C1,C2), (C2,C3) y (C3, C1). Hacemos esto sin importar el órden (C1, C2) o (C2, C1). Pues dado que el objetivo del problema era dar la solución más rapida, la velocidad de elegir primero a C1 y luego a C2, será la misma que hacer lo contrario. Esto se realiza para los 5 casos, independientemente de que tipo de explorador sea.

Entonces, por cada uno de estas situaciones, al entrar nuevamente al árbol de recursión, se vuelve a chequear si es un estado válido y se repite el procedimiento.

Para este entonces, el algoritmo recorrió cada uno de los estados E_j. Quedandose en el vector **soluciones** con conjuntos de soluciones, donde cada uno tendrá una velocidad diferente y todos serán válidos o serán -1. Finalmente el algoritmo recorre este vector y devuelve el más rápido (que es recorrer el vector de soluciones y buscar el mínimo). De esta manera tenemos garantizado que, en el vector **soluciones**, estarán todas las soluciones posibles de combinar cada uno de los movimientos; a quién mover y su velocidad. También garantizamos que dado el caso que no haya solución se devolverá -1 y dado el caso que sí haya, devolverá quien cruzó en menor tiempo.

En cuanto a la poda, podemos decir que es correcta utilizando el absurdo: Sea un Estado $E \in estadosAnteriores$. Supongamos que un estado $K \notin estadosAnteriores$. Si $E = K$ entonces podríamos volver a pasar por todos los estados que hubo entre E y K. Entonces podríamos pasar infinitas veces por todos esos estados y sumar tiempos infinitas veces, lo cual es absurdo. Por otro lado, el hecho de chequear solamente las cantidades de arqueólogos y caníbales de cada lado y la ubicación de la linterna, podemos afirmar que es correcto porque la misma razón que antes. Si bien podemos tener distintas personas, los movimientos que habrá entre cada estado será de un número de personas y no de cuáles personas. Lo que puede variar es la velocidad con la que se llega a cada estado, pero es por eso que se prueban todas las combinaciones de personas para que crucen el puente en cada estado.

2.4. Complejidad teórica

El algoritmo comienza tomando 2 vectores (*arqs* y *cani*), siendo uno para los arqueólogos y otro para los caníbales. En cada posición de *arqs* y *cani* se encontrará una velocidad correspondiente a algún arqueólogo o caníbal. El tamaño de cada vector será igual a la cantidad de arqueólogos/caníbales que se tomaron como entrada. Llamaremos n a la cantidad de arqueólogos. Además, dada la lógica del algoritmo, si hay más caníbales que arqueólogos al comenzar, la ejecución termina sin llamar a la función principal y devuelve -1 inmediatamente, ya que no hay forma de que al comienzo haya más arqueólogos que caníbales en el lado izquierdo. En cambio, si hay 0 aqrueólogos o más arqueólogos que caníbales, sí se llama a la función que realiza *Backtracking*. En ella, se utilizan 2 vectores más para poder distinguir el lado del que se encuentra cada arqueólogo y cada caníbal. La inserción y eliminación de cada elemento en cada vector será de $O(1)$ amortizado ya que en caso de que el vector deba redimencionarse, se copian todos los elementos del vector a uno más grande dejando como complejidad $O(n)$. En cada llamada a la función principal del algoritmo, se prueban 5 casos: que cruce un arqueólogo solo, un canibal solo, dos arqueólogos, dos caníbales o un caníbal y un arqueólogo. Luego, cada nodo del árbol tendrá 5 hijos. Cada uno de ellos, será un posible estado válido, que se chequea revisando un vector de *estados válidos* y que a lo sumo tendrá el tamaño de la altura del árbol a causa de mantener todos los estados válidos anteriores a cada nodo, y por lo tanto, revisarlo tomará $O(n^2)$ (probado más adelante). Si es un estado válido, se realizarán las operaciones necesarias para decidir quién cruzará el puente, pero como se desea probar con todas las combinaciones, se recorren los vectores correspondientes a los grupos que vayan a cruzar, dando en peor caso una cantidad de posibilidades igual a $\binom{n}{2} = \frac{n*(n-1)}{2} \in O(n^2)$. Esto se realizaría en los casos que se quieren cruzar a dos personas, mientras que en la que cruza una sola es recorrer solamente un arreglo en $O(n)$. Luego, es llamar recursivamente a la función principal. Hasta ahora nos quedaría que la complejidad en cada nodo es de $O(n^2 * n^2) \in O(n^4)$.

La altura del arbol puede ser acotada por la cantidad de nodos, que equivale a la cantidad de estados válidos. Este número se calcula de la siguiente manera:

Dado que la cantidad de caníbales está acotada por la cantidad de arqueólogos, la cantidad de formas válidas que hay para repartir a todas las personas en ambos lados del puente manteniendo el invariante de que no haya más caníbales que arqueólogos de ninguno de los dos lados se calcula utilizando combinatoria. La cantidad de caníbales posibles en cada lado del puente es menor o igual a la cantidad de arqueólogos de ese lado (o sea entre 0 y n), pero en caso de que no haya arqueólogos en alguno de los lados, la cantidad de caníbales posibles es igual a la cantidad total de arqueólogos, salvo que no haya ninguno y en ese caso n pasará a ser el número de caníbales totales.

$$\sum_{i=1}^{n}(i+1) + (n+1)$$

$$\frac{(n+2)(n+1)}{2} + (n+1) - 1$$

$$\frac{(n+2)(n+1)}{2} + n$$

$$\frac{(n+2)(n+1) + 2n}{2}$$

$$\frac{n^2 + 3n + 2}{2}$$

Y de este tipo de funcion sabemos

$$\frac{n^2 + 3n + 2}{2} \in O(n^2)$$

Entonces, la altura del árbol va a estar acotada por $O(n^2)$. Retomando, sabemos que la cantidad de hojas de un arbol es $O(b^h)$ y sabemos también que en cada hoja habrá una solución posible. Ergo, llegamos a que la complejidad de encontrar las soluciones está acotada por $O(b^h)$ donde b es la cantidad de ramas que se abren en cada nodo, h es la altura del árbol y todo esto es el tamaño del árbol. Como mencionamos anteriormente, b es exactamente 5 y la altura del árbol está acotada por $O(n^2)$. Luego, la

complejidad temporal para encontrar las soluciones sería $O(5^{n^2})$. Pero hasta aquí no tenemos en cuenta que cada nodo cuesta $O(n^4)$ y que a cada hoja del árbol llegaremos probando todas las combinaciones de personas, que como dijimos con anterioridad es $O(n^2)$. Incorporando esto a la complejidad anterior en la cual suponíamos que cada nodo tenía costo $O(1)$ y que se llegaba una sola vez a cada hoja, nos queda que la complejidad temporal en peor caso es $O(n^4)$ por un lado y $O(n^2 * (5^{n^2}))$ ya que es la cantidad de veces que se llega a las hojas. Teniendo ambas cosas, concluímos que la complejidad temporal es $O(n^4 * (n^2 * (5^{n^2}))) \in O(n^6 * 5^{n^2})$.

En cuanto a la complejidad espacial, se utiliza un historial de estados anteriores en los que se van guardando los estados válidos por los que se pasó hasta cierto punto en cada nodo del árbol. Se usa como si fuera una pila y cada vez que se accede a un nivel inferior en el árbol de ejecución, se guarda el estado actual de los caníbales, los arqueólogos y la linterna; mientras que cuando se sube en el árbol, se elimina el último estado actual. Debido a esto, la complejidad espacial en es $O(n^2)$ ya que la pila tendrá a lo sumo el mismo tamaño que la cantidad de estados en la rama más larga, y como probamos antes, este valor está acotado por esa complejidad. Además, lo que se guarda en cada estado son 4 valores enteros que indican la cantidad de arqueólogos y caníbales de cada lado, y un booleano (representado con 1 o 0) que indica de qué lado está la linterna.

2.5. Experimentación

Para poder visualizar que la cota propuesta en la complejidad temporal funciona para el algoritmo que resuelve este problema, realizamos tres experimentos. Por un lado, un experimento que fijó la cantidad de arqueólogos en 0, y varió la cantidad de caníbales de 0 a 6. Por cada cantidad de caníbales, se corrió 30 veces el algoritmo, y en base a este resultado se sacó un promedio el cual será graficado. Por otro lado, el segundo experimento, se iteró la cantidad de arqueólogos de 1 a 4. Luego por cada valor, se iteró la cantidad de caníbales de 0 a cantidad de arqueólogos. Luego al igual que el primer experimento, se corrió 30 veces el algoritmo con cada uno de los casos y se graficó el tiempo promedio. Luego el último experimento, consistió en la distintas combinaciones de n arqueólogos, con m caníbales, tal que $n+m = 6$. Y nuevamente, se corrió 30 veces con cada cantidad el algoritmo y se calculó el promedio. No está demás aclarar, que el tiempo de corrida para casos mayores a siete exploradores, el tiempo de ejecución era muy alto. Es por esto en todos los casos

Los casos de prueba pueden observarse en la tabla que se encuentra en el anexo de este informe. En la experimentación nos independizamos de las distintas velocidades, dado que, independientemente del valor de cada una, el algoritmo se fija en la cantidad de arqueólogos/caníbales y no en sus velocidades.

Los resultados obtenidos fueron plasmados en el siguiente gráfico. El mismo es la representación del tiempo en funcion de la cantidad de arqueólogos. También se muestra la funcion propuesta como cota de complejidad temporal.

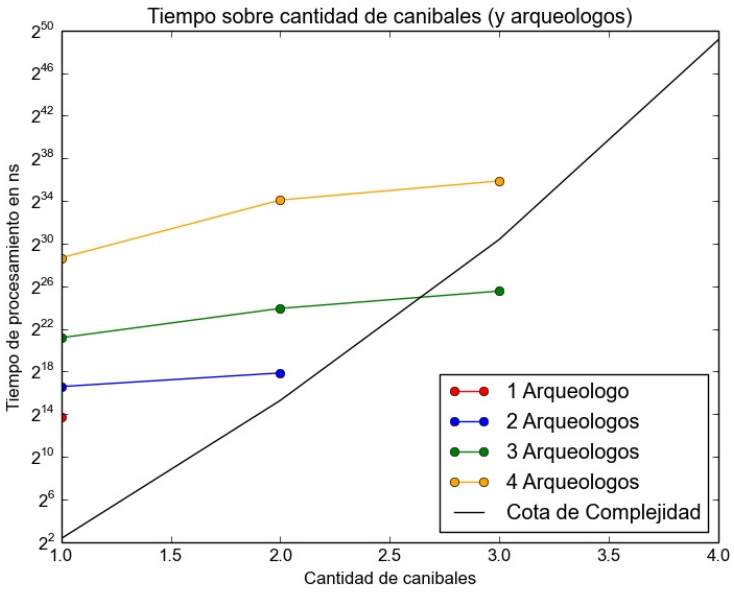

Figura 2

Pudimos notar como la cota de complejidad efectivamente se cumple con respecto al del algoritmo. Los peores casos dados, se mantuvieron entre 2 y 3 caníbales para los 4 casos de arqueólogos. Otra cosa que se puede observar, que la diferencia de tiempo de ejeución depende directamente de la cantidad total de exploradores. Entre otros motivos, la cantidad de ramas del árbol de ejecución es mayor dado la cantidad de combinaciones que debe abarcar por cada opción de cruzar el puente.

Para el caso en que no haya arqueólogos y solo haya caníbales, esperábamos que la resolución del problema sea más rápida que en los casos que hay más arqueólogos que caníbales. Probamos con cantidades de caníbales entre 1 y 6 y en el próximo grafico se ilustran los resultados de tiempo en función del número de personas.

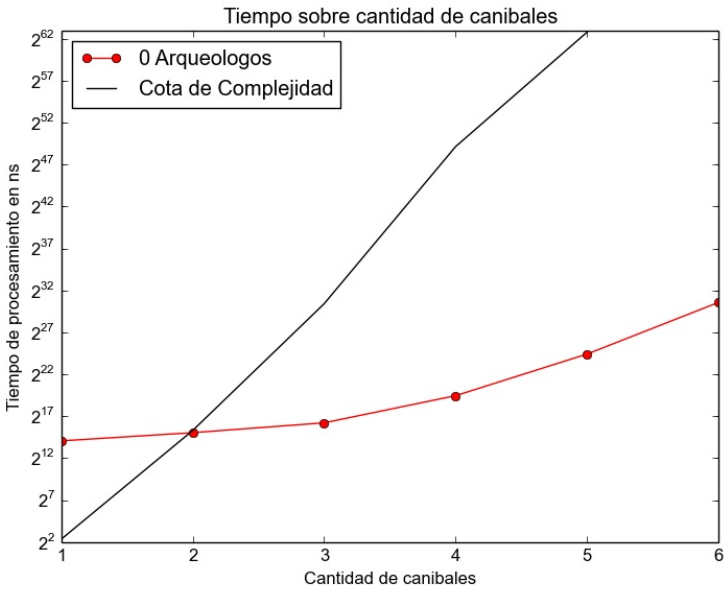

Figura 3

Notamos como el tiempo que toma, a mayor cantidad de caníbales sin arqueólogos, crece mucho más lento que para los casos con arqueólogos. Esto se debe a que el algoritmo, si bien va a probar manda 1 o 2 caníbales, únicamente debe encontrar la combinacion entre 2 caníbales. Sin probar mandar también a arqueólogos.

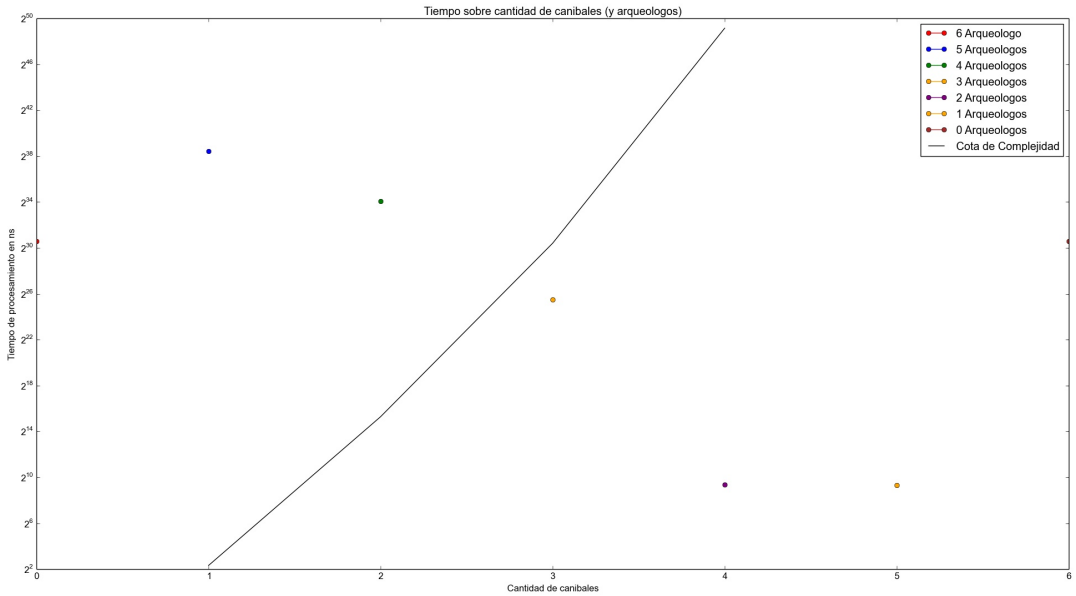

Figura 4

Por último, podemos observar en un mismo gráfico, el caso promedio además del mejor y peor. En primer lugar hablaremos del mejor caso. En donde habrá muchos caníbales y pocos arqueólogos. Este caso es directo, es decir, el algoritmo va a chequear que ningún caso es válido y no va a entrar en la recursión. Es por esto que para 4 y 5 caníbales los tiempos son menores. Por otro lado el caso promedio. Tanto para 6 caníbales como para 6 arqueólogos, además de para 3 caníbales y arqueólogos. Para el caso donde hay de un solo tipo (arqueólogos o caníbales), en el árbol de ejecución, los casos de mandar uno y uno, o dos del otro tipo, o uno del otro tipo, son salteados. Y de aquí es que el árbol de ejecución decrece y el tiempo de ejecución también. Por otro lado el caso de 3 y 3, también pertenece al tiempo promedio, pues, la cantidad de casos donde cae en los estados inválidos, son mayores a por ejemplo, donde hay más arqueólogos que caníbales, y esto evita en menor medida que se ramifique el árbol de ejecución. Por último, el peor caso, donde hay 4 arqueólogos y 2 caníbales o 5 arqueólogos y 1 caníbal. Al igual que antes, la cantidad de veces que cae en casos inválidos es mucho menor que en otros casos. Luego, las ramificaciones serán siempre válidad, y estas tendrán todos los movimientos posibles, además de muchas combinaciones de a quién enviar de cada grupo.

3. Ejercicio 2: Problemas en el camino

3.1. Descripción del problema

Figura 5: Equilibro

Luego de cruzar el puente, el equipo se encuentra en una habitación amplia con una puerta, la cual está cerrada. En el medio de la habitación hay una balanza de dos platillos, la cual contiene sobre su platillo izquierdo una llave que abre la puerta de la habitación. Al retirar la llave, se cambia el equilibro de la balanza, lo cual hace que se cierre la entrada de la habitación. La llave no sirve para abrir la puerta y lo único que puede realizar el equipo es recompener el equilibro de la balanza utilizando un conjunto de pesas de potencia de 3 distintas. Es decir, sólo disponen de una sóla pesa de cada potencia de 3. Los datos de entrada y de salida para el problema son los siguientes:

Formato de entrada: El único dato de entrada es un entero P que representa el peso de la llave.

```
P
```

Salida: La salida consiste en dos enteros S y T que representan la cantidad de pesas que hay que colocar en el platillo izquierdo y el derecho respectivamente, seguidas de otra línea con los valores de las pesas que se deben colocar del platillo izquierdo ordenadas de menor a mayor y finalmente otra línea con los valores de las pesas del platillo derecho, también con el mismo orden.

```
S       T
I1 ...  ...IS
D1 ...  ...DT
```

Se tiene como restricción que el peso P está en el siguiente rango: 1<= P <= 10^{15}. La complejidad temporal debe ser a lo sumo O($\sqrt{P}$).

3.2. Solución propuesta

La idea propuesta para la resolución del problema consiste en buscar cómo obtener una representación del número p. Si se consigue una representación del mismo se puede equilibrar la balanza. Primero nos abtraemos de las restricciones del problema y pensamos que cualquier número $a \in$ N admite un desarrollo en base d, utilizando sólo símbolos comprendidos en el rango de 0 a $d-1$. Como se dispone de sólo pesas que son potencias de 3, se decide utilizar la base 3 para representar el número. Si el problema no tuviera la restricción de que sólo se dispone de una pesa de cada potencia de 3, la resolución sería trivial ya que sólo consiste en obtener el desarrollo en base 3 del número p (que es un dato de entrada y que se asume que su desarrollo es en base 10) y poner tantas pesas de cada potencia de 3 (hasta 2 por potencia ya que en base 3, sólo se pueden utilizar los símbolos 0, 1 y 2) como indique el desarrollo, en el plato de la izquierda de la balanza para obtener el mismo peso p. Pero la restricción de que sólo se dispone de una sóla pesa de cada potencia de 3 hace que dicha solución no funcione. No obstante, teniendo en cuenta que en la balanza, las pesas que se colocan en el lado derecho 'restan' el peso colocado en el plato izquierdo se puede pensar en una solución en la cual cuando se tengan que utilizar dos pesas de la misma potencia, se puedan reemplazar por el uso de una pesa mayor colocada en el plato izquierdo (la cual se interpretaría como suma de dicha potencia de 3) y el uso de otra pesa colocada en el plato derecho (la

cual se interpretaría como resta de dicha potencia de 3). Para lograr que la suma de dicha pesa menos la resta de la otra pesa, den como resultado el mismo valor que las dos pesas originales, se debe encontrar alguna relación entre las potencias de 3. La solución propuesta consiste en lo siguiente:

Caso 1: El desarrollo en base 3 del número p no contiene el símbolo 2.

Este caso es trivial. Significa que a lo sumo se usa una vez cada potencia de 3 en el desarrollo del número. Por lo tanto, para obtener la solución al problema sólo bastar tomar las pesas correspondientes (las potencias de 3 que se utilizan en el desarrollo) y colocarlas en el plato izquierdo de la balanza para obtener el mismo peso original p y equilibrarla.

Caso 2: El desarrollo en base 3 del número p contiene al menos un símbolo 2.

Este es el caso interesante ya que no se dispone de dos pesas de la misma potencia. Sin embargo, se pueden reemplazar dos pesas de la misma potencia de la siguiente manera:

$$\begin{gathered} 2.3^n = 3^{n+1} - 3^n \\ 2.3^n = 3.3^n - 3^n \\ 2.3^n = (3-1).3^n \\ 2.3^n = 2.3^n \end{gathered}$$

Es decir, se pueden reemplazar dos pesas de una potencia de 3 por la inmediata potencia de 3 superior menos una sóla pesa de las dos potencias de 3. Igualmente esto no siempre se puede realizar ya que puede darse el caso de que ya se estén usando dos pesas de la potencia de 3 inmediata superior como en el siguiente ejemplo:

$$p = 1.3^2 + 2.3^1 + 2.3^0$$

Entonces se debe encontrar una solución general que resuelva el problema antes mencionado. Se propone la siguiente solución:

$$\sum_{i=0}^{n} 2.3^i = 3^{n+1} - 3^i$$

Se realiza la demostración de la anterior fórmula usando inducción en n:

Caso base: $n = 0$

$$\begin{gathered} \sum_{i=0}^{0} 2.3^i = 3^1 - 3^0 \\ 2.3^0 = 3 - 1 \\ 2 = 2 \end{gathered}$$

Paso inductivo: Usamos como hipótesis inductiva (HI) que la fórmula vale para n y queremos probar que vale para $n+1$.

$$\begin{gathered} \sum_{i=0}^{n+1} 2.3^i = 3^{n+2} - 3^i \\ \sum_{i=0}^{n} 2.3^i + 2.3^{n+1} = 3^{n+2} - 3^i \\ \text{Por HI:} \\ 3^{n+1} - 3^i + 2.3^{n+1} = 3^{n+2} - 3^i \\ 3.3^{n+1} - 3^i = 3^{n+2} - 3^i \\ 3.3.3^n - 3^i = 3^2.3^n - 3^i \\ 9.3^n - 3^i = 9.3^n - 3^i \end{gathered}$$

□

Como conclusión se puede observar que cuando se tenga el caso de que no sea posible tomar una pesa de una potencia de 3 porque ya se usan dos pesas de dicha potencia y tampoco sea posible reemplazarla por la siguiente ya que también se usan dos pesas de dicha potencia (es decir, en el desarrollo del número hay varios símbolos 2 consecutivos) se debe buscar la primer potencia de 3 más grande de la cual no se tenga que utlizar dos pesas y reemplazar las pesas originales por la suma del valor de la potencia 3 de la pesa encontrada menos el valor de la potencia de 3 más chica de la cual se utilizan dos pesas. Esta fórmula se puede aplicar a cada uno de los símbolos en el desarrollo del número.

3.2.1. Implementación

Habiendo introducido la idea, se detalla el comportamiento del algoritmo para una entrada con P. Se presenta un pseudocódigo para tener de referencia al seguir la explicación detallada a continuación:

```
izquierda = verdadero
divisiones = 0
izquierdas = []
derechas = []
Mientras p != 0
    cociente = p / 3
    resto = p - (cociente *3)
    Si resto es 0
      Si no lo puse en izquierda al anterior
        Agrego a izquierdas la pesa con valor (3 ^ divisiones)
      izquierda = verdadero
    Si resto es 1
      Si anterior lo puse en izquierda
        Agrego a izquierdas la pesa con valor (3 ^ divisiones)
      Si no
        Agrego a derechas la pesa con valor (3 ^ divisiones)
    Si resto es 2
      Si anterio lo puse en izquierda
        Agrego a derechas la pesa con valor (3 ^ divisiones)
      izquierda = falso
    p = cociente
    divisiones += 1
Fin Mientras
Si el último no lo puse en la izquierda
   Agrego a izquierdas la pesa con valor (3 ^ divisiones)
```

1. La primera parte consiste en la inicialización de valores para el algoritmo. Se tiene una variable booleana que indica en qué platillo fue colocada la última pesa para cada iteración. Esta variable se inicializa en verdadera y es utilizada para saber en qué platillos hay que colocar las pesas que siguen. Esta variable es muy importante ya que determina en qué casos se van a reemplazar dos pesas de una potencia de 3, por otras pesas y cómo hacerlo. La variable divisiones es simplemente un índice que representa la cantidad de divisiones realizadas al número P. Por último se tienen dos listas inicializadas vacías, que van a almacenar las pesas que hay que colocar en los platillos izquierdo y derecho.

2. El ciclo principal se ejecuta mientras P sea distinto de cero. En el cuerpo del ciclo se realiza la división del número P por 3 y se almacenan en dos variables distintas el cociente y el resto de dicha operación. Para el caso del valor del resto se tienen 3 posibilidades:

    2.1. EL resto es igual a 0: En este caso hay que preguntar si la anterior pesa fue colocada o no en el platillo derecho. Si fue colacada en el platillo derecho, se agrega a la lista de izquierdas la pesa que corresponde al valor ($3^{divisiones}$). La variable izquierda se actualiza con el valor verdadero, indicando que la última pesa fue colacada del lado izquierdo.

    2.2. El resto es igual a 1. En este caso tenemos dos posibilidades distintas:

        2.2.1. La anterior pesa fue colocada del lado izquierdo, por lo tanto esta pesa también tiene que colocarse del mismo lado. Se agrega entonces a la lista de izquierdas el valor ($3^{divisiones}$).

        2.2.2. La anterior pesa fue colocada del lado derecho, por lo tanto esta pesa también tiene que colocarse del mismo lado. Se agrega entonces a la lista de derechas el valor ($3^{divisiones}$).

    2.3. El resto es igual a 2: Sólo basta con fijarse si la anterior pesa fue colacada en el platillo izquierdo. De ser así, se debe agregar a derechas la pesa con el valor ($3^{divisiones}$).

3. Lo último que se realiza en el ciclo es actualizar el valor de P, el cual va a contener el valor del cociente calculado al principio. De esta manera el valor de P se irá decrementando en cada ciclo hasta llegar a cero. También se suma uno al índice de divisiones. Finalmente, se pregunta si la última pesa fue colocada en la lista de derechas. De ser así, se agrega a la lista de izquierdas la pesa con valor ($3^{divisiones}$).

3.2.2. Demostración de correctitud

Habiendo visto cómo funciona el algoritmo desarrollado, se procede a justificar por qué devuelve una solución. Para esto se explicará el algoritmo en base a las justificaciones matemáticas realizadas en la solución propuesta.

Correctitud de ciclo

Lo que se va a demostrar a continuación es que el ciclo principal termina y calcula la solución al problema.

El ciclo se ejecuta mientras P sea distinto de cero. Es decir termina cuando P es 0 y esto ocurre ya que lo que se hace en el ciclo es dividir el número P por 3 y reemplazar P por el cociente de esta división en cada iteración. Eventualmente el cociente va a ser cero, por los teoremas de división entera y de desarrollo en base d. Lo que se está haciendo es utilizar el algoritmo de división para obtener el desarrollo en base 3 del número P. Los restos en cada iteración son los que indican que símbolos hay que utilizar para la representación del número en esa base. En este caso no interesa obtener la representación del número en base 3, sino trabajar con los símbolos en su desarrollo (restos). El resto de la división sólo puede ser 0, 1 o 2 (por el teorema de la división entera).

A continuación se analiza la parte fundamental del algoritmo que consiste en la identificación de qué pesas utilizar y en qué platillos colocarlas dependiendo del resto. Se tienen las siguientes posibilidades:

1. Resto == 0. En este caso, hay que analizar qué fue lo que se realizó antes. Si el dígito anterior en la descomposición del número era un 0 o un 1 (hecho que se va a representar como que la última pesa fue colocada del lado izquierdo) no hay que realizar nada. En cambio si la anterior pesa fue colocada del lado derecho es porque el anterior dígito en el desarrollo del número era un 2. En este caso, hay que utilizar esta pesa (por lo expuesto en solución propuesta), por lo que se agrega a la lista de izquierdas el valor de la potencia de 3 correspondiente. Independientemente de si se utilizó esta pesa o no, la variable izquierda se actualiza con el valor verdadero, ya que el hecho de que no se use una pesa de una determinada potencia también se toma como que se 'usó" la pesa de la izquierda, por el mismo motivo explicado recién. Es decir, el valor de la variable izquierda es la que permite identificar en cada iteración, cual fue la última operación que se realizó.

2. Resto == 1. En este caso siempre hay que agregar una pesa. Si el valor de la variable izquierda es verdadero (esto es porque el anterior dígito fue un 0 o un 1) hay que agregar esta potencia de 3 a la lista de izquierda. En el caso contrario significa que el anterior dígito era un 2 y como se demostró en la solución propuesta hay que colocar esta pesa en el lado derecho, ya que si el anterior fue un 2 hay que tomar este dígito. Por lo tanto en este último caso se agrega la potencia de 3 correspondiente en la pesa derecha. El valor de la variable izquierda no es necesario actualizarlo ya que si se agrega a la lista de izquierdas hay que modificarlo por verdadero (lo cual ya ocurre porque se agrega solamente a izquierdas si es verdadero) y si se agrega a la lista de derechas hay que modificarlo por falso (lo cual ya ocurre porque se agrega solamente a derechas si es falso).

3. Resto == 2. En este caso se usa también la idea planteada en la solución al problema. Se agrega a la lista de derechas la pesa con el valor de la potencia de 3 correspondiente solamente si la anterior fue colocada en el lado izquierdo (es decir el anterior dígito es un cero o un uno) ya que si el anterior era un 2 no hay que realizar nada (por lo visto en la demostración de la solución). Finalmente se actualiza el valor de izquierda y se pone en falso siempre, ya que es necesario para el siguiente dígito en el desarrollo, saber que el anterior era un 2.

Como el ciclo finaliza cuando P es igual a cero, se puede dar el caso de que el último dígito en el desarrollo del número en base 3 (el más significativo) sea un 2. Por lo tanto y sabiendo que no sigue ningún dígito más, lo que hay que hacer si se da este caso es tomar esta pesa por lo visto en la solución.

En conclusión lo que hace el algoritmo es dividir iterativamente el número por 3, obteniendo de esta forma su desarrollo en base 3 y al mismo tiempo colocando las pesas correspondientes en base a los restos obtenidos (teorema de desarrollo en base d). No es necesario obtener primero la representación del número en la base y después aplicar la solución; se puede realizar mientras se obtiene, tomando decisiones en base al resto de la división en cada iteración. La idea propuesta garantiza que el algoritmo siempre va retornar una solución por lo visto en la sección de "Solución". Las decisiones que se toman con los restos son las inferidas por la solución propuesta: si el dígito es 0, no se toma esa pesa salvo que el anterior sea un 2. si el dígito es 1, si el anterior es 0 o 1 se toma esa pesa, sino se resta la pesa; finalmente si el dígito es 2, si anterior es 2 no se toma ninguna acción y si no se resta la pesa. Además el algoritmo de división garantiza que las pesas que se toman, van a estar ordenadas de menor a mayor cumpliendo con la restricción del problema.

3.3. Complejidad teórica

Para calcular la complejidad teórica de la solución propuesta se hará referencia a la sección **??** donde se posée el pseudocódigo junto a su explicación.

El algoritmo tiene una complejidad temporal de O($log(n)$), por lo tanto es logarítmico y cumple con la restricción de complejidad del enunciado. Esto se justifica con el hecho de que el algoritmo posee un ciclo principal que se ejecuta mientras P sea distinto de cero, es decir desde el número P hasta el número 0. En cada iteracion el tamaño de P se reduce a un tercio, ya que se divide por 3. Por lo tanto el ciclo se ejecuta la cantidad de símbolos necesarios para obtener el número P en base 3. Si P requiere de n símbolos es que:

$3^{n-1} <= P < 3^n$

es decir, $n - 1 <= log_3(P) < n$, lo que implica que $[log_3(P)] = n - 1$ y por lo tanto $n = [log_3(P)] + 1$

Dentro del ciclo sólo se hacen comparaciones que tienen un costo asociado de O(1), una división, una multiplicación y una resta de enteros (costo O(1)), una asignación y una suma, y se agregan elementos a una lista (todas estas operaciones tambien en O(1)). Por lo tanto se concluye que la complejidad del ciclo es O($log(n)$). Fuera del ciclo sólo se realizan incializaciones de variables O(1) al principio del algoritmo y una comparación e inserción en una lista al final también en O(1). Por álgebra de órdenes de funciones, la complejidad temporal resultante del algoritmo es O($log(n)$).

3.4. Experimentacion

Se realizaron pruebas experimentales para verificar que el tiempo de ejecución del algoritmo cumpliera con la cota asintótica de O($log(n)$), demostrada teóricamente. Para ello fue necesario modificar el algoritmo propuesto, ya que como la complejidad está definida por la cantidad de símbolos necesarios para la representación del número en base 3 y se explicó que la misma era log_3 del número más uno, no se observan grandes diferencias en el tiempo de ejecución para números 'razonables', es decir números que puedan ser representados de una forma práctica y eficiente en C++. Por ejemplo para el caso de un número muy grande como 3^{30}, la cantidad de iteraciones que va a realizar el ciclo es sólo 31, que no es muy diferente a la cantidad de iteraciones necesarias para el número 3^3 (4 iteraciones) que es mucho menor. Los tiempos de ejecución de estas instancias son muy similares y dado a que hay muchas variables en cuestión cuando se corren (otros procesos que interfieren, atención de interrupciones y scheduling del S.O, etc) puede hasta darse el caso de que tarde más una instancia menor que una mayor. Se presenta la siguiente modificación del algoritmo para realizar las pruebas:

```
izquierda = verdadero
divisiones = 0
izquierdas = []
derechas = []
Mientras v.size != 0
    Si v[size -1] = 0
      Si no lo puse en izquierda al anterior
        Agrego a izquierdas la pesa con valor (3 ^ divisiones)
      izquierda = verdadero
    Si v[size -1] = 1
      Si anterior lo puse en izquierda
        Agrego a izquierdas la pesa con valor (3 ^ divisiones)
      Si no
        Agrego a derechas la pesa con valor (3 ^ divisiones)
    Si v[size -1] = 2
      Si anterio lo puse en izquierda
        Agrego a derechas la pesa con valor (3 ^ divisiones)
      izquierda = falso
    v.size -= 1
    divisiones += 1
Fin Mientras
Si el último no lo puse en la izquierda
   Agrego a izquierdas la pesa con valor (3 ^ divisiones)
```

Este algoritmo a diferencia del original, recibe como parámetros un arreglo de enteros (v) que representa el desarrollo en base 3 de un número y un entero que es el tamaño del arreglo. El arreglo de enteros puede contener por lo tanto en cada posición un entero entre 0 y 2 inclusive. El arreglo se interpreta como que en la posición 0 está el dígito más significativo y en la última el menos significativo. Con esta idea se puede realizar un mejor análisis del rendimiento del algoritmo ya que se pueden simular números mucho más grandes. Se puede tomar un arreglo de 5000 posiciones, lo cual representaría un número muy grande en base 3 que no se puede representar directamente. La idea es que este algoritmo realiza en el fondo lo mismo que el anterior, ya que lo que se hace es recorrer el arreglo (ciclo principal) y después tomar las mismas decisiones que en el original en base a los valores contenidos en cada posición del arreglo. Es decir lo que determina la complejidad en este caso es el tamaño del arreglo, pero el mismo por lo expuesto anteriormente es igual al logaritmo en base 3 de un número P cualquiera, el cual sería el resultante de realizar la descomposición del desarrollo.

Para las pruebas lo que se realizó es probar con arreglos de tamaño 1 hasta 4600. Para estandarizar y que no haya constantes diferentes para cada arreglo (ya que lo que se hace depende del valor de cada posición) en todos los casos se completaron todas las posiciones de los arreglos con el valor 1, para que el tiempo esté determinado sólo por el tamaño del arreglo. Además cada uno de los arreglos es testeado 5000 veces para disminuir los outliers. Lo esperable es que haya una relación lineal entre el tiempo de corrida y el tamaño del arreglo, pero el mismo se debe interpretar como que la relación es logarítmica entre el número que representa el arreglo y el tiempo.Se utilizó para que esto se visualice mejor la escala logarítica, como se puede observar en el siguiente gráfico:

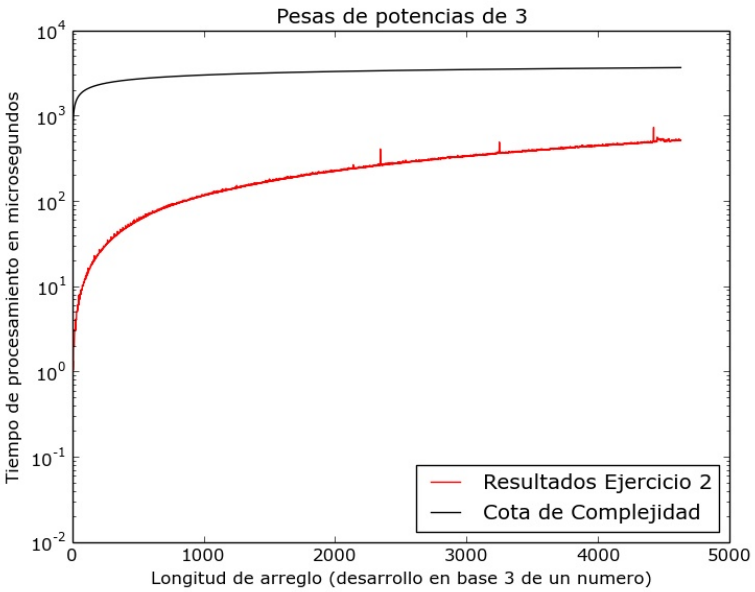

Figura 6

El análisis expuesto de los datos recopilados evidencia que la cota de complejidad demostrada teóricamente es correcta. Además se logra ver que la complejidad depende estrictamente de la cantidad de símbolos que hay que utilizar para el desarrollo del número en base 3.

4. Ejercicio 3: Si esto no es el tesoro, el tesoro dónde está

4.1. Descripción del problema

Figura 7: Fortuna y gloria muñeca, fortuna y gloria

Una vez equilibrada la balanza, las paredes vuelven a su lugar. Con la llave obtenida logran abrir la puerta que estaba trabada.

Al iluminar la habitación se encuentran con una sala de N tesoros.

Por cada tipo de tesoro i, hay C_i cantidades, con un valor V_i y un peso P_i. Si bien el objetivo principal de la expedición era otro, nunca viene mal armarse de algún recuerdo. Tienen M mochilas disponibles para llevarse los tesoros y cada una tiene una capacidad máxima de peso K_j que puede llevar.

El objetivo es llevarse el mayor valor posible de tesoros. Para esto, se debe devolver como resultado el valor total de tesoros guardados y además, por cada mochila, la cantidad de tesoros llevados y de qué tipo son.

Por ejemplo, para la siguiente entrada:

```
M = 2
K = { 1, 3 }

N = 3
C = { 1, 2, 3 }
V = { 2, 4, 10 }
P = { 1, 2, 5 }
```

Una posible salida válida sería:

```
S = 6
M1 = { 1, 1 }
M2 = { 1, 2 }
```

Es decir, para el caso en el que tenemos dos mochilas con capacidades 1 y 3, y nos encontramos con tres tesoros, el primero de peso 1 y valor 2, el segundo de peso 2 y valor 4, y el tercero de peso 5 y valor 10...

Nos llevaremos en la primer mochila el primer tesoro y en la segunda el segundo tesoro. El tercer tesoro ya no nos entra.

4.2. Solución propuesta

Para solucionar el problema planteado se usó la técnica algorítmica de *Programación dinámica*.

La idea es separar el problema principal en subproblemas. En este caso, el subproblema es, en base a la capacidad restante de cada mochila y, al valor y peso de cada tesoro, en qué mochila conviene meter cada uno (incluyendo la opción de no meterlo en ninguna). Más formalmente el subproblema es: si ya resolvimos el beneficio máximo para los tesoros T_j, $j < i$, cómo se calcula el beneficio máximo teniendo

en cuenta al T_i. Entonces la idea es la siguiente: si tenemos un solo tesoro (caso base) el máximo beneficio que se puede obtener es el valor del tesoro en caso de que entre en alguna mochila. Para el tesoro siguiente (recursión) el máximo beneficio que se puede obtener es el máximo entre: el valor del tesoro actual más el valor máximo que se podía llevar con el tesoro anterior con el peso restante de la mochila 1 al colocar este tesoro si entra, a esto lo llamaremos **ValorMochila1**, el mismo valor pero para la mochila 2 o **ValorMochila2**, el mismo valor pero para la mochila 3 o **ValorMochila3** , y así sucesivamente para las n mochilas y el valor que se obtiene si no se coloca, que sería el valor que se puede obtener para el tesoro anterior. Es decir la idea intuitiva es que en cada paso se compara el benficio de poner el tesoro en algunas de las mochilas con el beneficio de no ponerlo, en base al beneficio anterior. Podría verse como que se van comparando de a pares el beneficio de un tesoro y otro y nos quedamos con el mejor, siempre viendo todas las posilibidades de ponerlo en todas las mochilas, y despues de quedarse con la mejor opcion, se repite el procedimiento pero para el beneficio máximo calculado entre esos dos tesoros y uno que sigue. Se puede observar que el orden en el cual se recorren los tesoros no es importante. La solución que se obtiene luego de comparar con el último tesoro es óptima ya que en cada paso se está obteniendo el mayor beneficio,comparando todas las posibilidades. Se propone una función de recursión, la cual está definida en el rango 1..n, siendo n la cantidad de tesoros total y $F(i)$ indicando el beneficio máximo que se puede obtener considerando los primeros i tesoros:

$F(1)$ = valor v del primer tesoro si éste entra en alguna de las mochilas

$F(i)$ = max(valorMochila1,valorMochila2,valorMochila3, $F(i-1)$)

Lo que falta aclarar en esta función es que para cada tesoro se debe calcular el beneficio máximo que se puede obtener para cada peso posible de cada mochila, es decir se calculan los beneficios máximos para todos los posibles pesos de una mochila, desde uno hasta el peso P de la mochila, para que sea posible conocer cuál es el beneficio total que se puede obtener con el próximo tesoro, ya que se necesitaba para eso el beneficio que se puede obtener con el peso restante al poner un tesoro.

Por lo tanto, el algoritmo propuesto consta de una matriz para cada tesoro, de tamaño $\prod_{i=0}^{3} C_i + 1$, siendo C_i las capacidades de las **M** mochilas, donde $M \leq 3$. Y además una matriz nula de igual dimensión que servirá para el caso en donde no hayan tesoros.

El algoritmo funciona de la siguiente manera: 1) Se crea una matriz de $\prod_{i=0}^{M} C_i + 1$ para cada tesoro, y una extra inicial con los valores de todas las posiciones inicializados en 0.

```
for i = 1 .. cantTesoros + 1
    for j = 1 .. capacidadMochila1 + 1
        for k = 1 .. capacidadMochila2 + 1
            for l = 1 .. capacidadMochila3 + 1
                matriz[i][j][k][l] <-- 0
            end
        end
    end
end
```

2) En el caso donde $M < 3$, el programa se encarga de fijar las capacidades de las $3 - M$ mochilas en 0. Para cada posición (i, j, k) de la matriz del tesoro t, siendo i, j y k la capacidad *restante* de las mochilas 1, 2 y 3, se pregunta si t entra en alguna de las tres mochilas. Si entra en la mochila 1, se obtiene valor(t) sumado al valor de la matriz del tesoro t-1 (recordar que el primer tesoro tiene una matriz predecesora con todos sus valores nulos) en la posición correspondiente según la capacidad restante de esa mochila (i - peso(t), j, k). Se repite este procedimiento para cada mochila. Y además se obtiene el valor de la posición (i, j, k) de la matriz del tesoro t-1 para incluir el caso en el que no se mete el objeto en la mochila. Luego se obtiene el resultado mayor de cada suma, y se guarda en la matriz del tesoro t, en la posición (i, j, k)..

```
valorMochila1 = 0
valorMochila2 = 0
valorMochila3 = 0

valorNingunaMochila = matriz[t-1][i][j][k]


if (pesoTesoro <= i ||pesoTesoro <= j || pesoTesoro <= k)

    if pesoTesoro <= i then
        valorMochila1 <-- valor(t) + matriz[t-1][i - pesoTesoro][j][k]
    endif

    if pesoTesoro <= j then
        valorMochila2 <-- valor(t) + matriz[t-1][i][j - pesoTesoro][k]
    endif

    if pesoTesoro <= k then
        valorMochila3 <-- valor(t) + matriz[t-1][i][j][k - pesoTesoro]
    endif

    matriz[t][i][j][k] <-- max(valorMochila1, valorMochila2, valorMochila3, valor(t-1))
else
    matriz[t][i][j][k] < -- valor(t-1)
```

Se repite este procedimiento para cada tesoro, hasta completar todas las posiciones de cada matriz. El último valor obtenido de la matriz del último tesoro, será el valor máximo que se puede acumular en las mochilas con los tesoros.

3) Luego, empezando desde la matriz del último tesoro, desde la última posición (i, j, k), se pregunta si entra ese tesoro en alguna mochila. Si no entra, se repite el procedimiento para el tesoro anterior. Si entra, se pregunta en cuáles de las mochilas entra ese tesoro. Para eso se guarda la capacidad disponible de cada mochila (inicialmente con su capacidad total) y se pregunta si el tesoro tiene peso menor o igual a cada capacidad. En caso de que entre en más de una mochila, se compara el valor resultante de ponerlo en cada una. Por ejemplo, para la mochila 1, se obtiene el valor de la posición (i - peso(t), j, k) de la matriz del tesoro t-1. Una vez obtenido en qué mochila es conveniente poner a ese tesoro, se almacena en cada vector de salida (hay uno por mochila) los tipos de tesoro que guardo en cada mochila. Se incrementa el contador de tesoros de esa mochila, y luego el indice pasa a ser aquella posición donde estaba el valor máximo. Por ejemplo, si la mejor opción es guardar ese tesoro en la mochila 1, (i, j, k) pasa a ser (i - peso(t), j, k), t pasa a ser t-1, y se le resta peso(t) a la capacidad restante de esa mochila.

```
    entraEnAlgunaMochila <-- matriz[t+1][i][j][k] != matriz[t][i][j][k]

    if entraEnAlgunaMochila then
     valorAnterior1 <-- 0
     valorAnterior2 <-- 0
        valorAnterior3 <-- 0
    entraEnMochila1 <-- (pesoTesoro <= i);
entraEnMochila2 <-- (pesoTesoro <= j);
        entraEnMochila3 <-- (pesoTesoro <= k);

if entraEnMochila1 then
valorAnterior1 <-- matriz[t][i - pesoTesoro][j][k]
endif
if entraEnMochila2 then
valorAnterior2 <-- matriz[t][i][j - pesoTesoro][k]
endif
        if entraEnMochila2 then
            valorAnterior3 <-- matriz[t][i][j][k - pesoTesoro]

if entraEnMochila1 && valorAnterior1 >= valorAnterior2  && valorAnterior1 >= valorAnterior3
        then
add(mochila1, tipo(t))
cantTesorosMochila1++
i -= peso(t)
else if entraEnMochila2 && valorAnterior2 >= valorAnterior3
add(mochila2, tipo(t))
cantTesorosMochila2++
j -= peso(t)
        else
            add(mochila3, tipo(t))
            cantTesorosMochila3++
            k -= peso(t)
endif

endif
```

Se repite este procedimiento para cada tesoro.

4) Una vez finalizados los primeros tres pasos, se copia en una tupla el valor obtenido entre todos los tesoros guardados en las mochilas, y un vector mochilas que contiene un vector por mochila. Cada uno de estos vectores guarda la cantidad de tesoros y los tipos de tesoro que hay en esa mochila.

4.3. Complejidad teórica

Para analizar la complejidad teórica, vamos a hacer referencia a la **implementación** del algoritmo. Para esto, el mismo, puede ser dividido en varias etapas independientes, donde al final, sus complejidades serán sumadas, dando así la complejidad teórica final. Las etapas son,

- En primer lugar, se inicializan ciertas variables necesarias para el algoritmo en O(1), además de inicializar la variable *capacidades*, en la cual se insertan las capacidades de las distintas mochilas. Siendo M la cantidad de mochilas, y O(1) la operación de *incersión* en un vector, esto da un total de O(M) la inicialización de variables.
- Además de las variables, se inicializa un vector de matrices denominado **cuboMagico**. Las mismas serán usadas para almacenar las ganancias acumuladas de llevar un tesoro o no. Para esto, sea N los tipos de tesoros y C_i la cantidad del tesoro de tipo i, se redimensionará el vector de matrices

cuboMagico por cada uno de los tesoros además de una matriz nula. Es decir en $O(\sum_{i=0}^{N-1} C_i + 1)$ inicializa el vector de matrices.

- Luego sea T, la cantidad de tesoros total más la matríz nula, es decir $T = \sum_{i=0}^{N-1} C_i + 1$ A cada matríz se la redimensionará con la capacidad de las M mochilas. O sea para cada tesoro, se inicializa una matriz cuyas dimensiones son $\prod_{i=0}^{M-1} K_i$. Esta inicialización tiene una cota asintótica de $O((\prod_{i=0}^{M-1} K_i) * T)$.
- Luego recorro cada una de las matrices, para sumar el valor acumulado por cada posición de cada matriz. Esto es, recorrer todas las matrices una vez cada una tiene un costo de complejidad de $O((\prod_{i=0}^{M-1} K_i) * T)$.
- Finalmente debo recorrer desde la última matriz hasta la primera. Cada vez que el algoritmo determine que un tesoro es óptimo para llevar en una mochila, el tesoro será ingresado al vector de cada mochila. La inserción de un elemento a un vector es O(1).Pero recorrer todas las matrices vuelve a tener una complejidad de $O((\prod_{i=0}^{M-1} K_i) * T)$.

Por lo visto anteriormente, el algoritmo tiene complejidad temporal de

$$O(M) + O(T) + O((\prod_{i=0}^{M-1} K_i) * T) =$$

$$O(M + (\prod_{i=0}^{M-1} K_i) * T) =$$

$$O((\prod_{i=0}^{M-1} K_i) * T) =$$

$$O((\prod_{i=0}^{M-1} K_i) * (\sum_{i=0}^{N-1} C_i + 1)) =$$

Luego, la complejidad final de nuestro algoritmo es

$$O((\prod_{i=0}^{M-1} K_i) * (\sum_{i=0}^{N-1} C_i))$$

Dado que nuestra cota máxima por enunciado era $O((\sum_{i=0}^{M-1} K_i)^M * (\sum_{i=0}^{N-1} C_i))$ resta demostrar que.

$$\prod_{i=0}^{M-1} K_i \leq (\sum_{i=0}^{M-1} K_i)^M$$

Para esto, podemos desarrollar la productoria y la sumatoria como,

$$\prod_{i=0}^{M-1} K_i = K_0 * K_1 * \cdots * K_{M-1}$$

$$(\sum_{i=0}^{M-1} K_i)^M = (K_0 + K_1 + \cdots + K_{M-1})^M$$

Resta probar que,

$$K_0 * K_1 * \cdots * K_{M-1} \leq (K_0 + K_1 + \cdots + K_{M-1})^M$$

Sea K_j la mayor capacidad de todas, entonces

$$K_0 * K_1 * \cdots * K_{M-1} \leq (K_j)^M$$

Luego por propiedad de polinomios, seguro $(K_j)^M$ pertenece a algún coeficiente de elevar a la M

$$(K_0 + K_1 + \cdots + K_{M-1})^M$$

Además todos los coeficientes $K_i \neq K_j$ son positivos.

Por lo tanto se demostró que $K_0 * K_1 * \cdots * K_{M-1} \leq (K_j)^M \leq (K_0 + K_1 + \cdots + K_{M-1})^M$

Es decir, $\prod_{i=0}^{M-1} K_i \leq (\sum_{i=0}^{M-1} K_i)^M$.

Luego la cota dada es correcta.

4.4. Experimentación

Al igual que con los otros dos ejercicios, se realizaron pruebas experimentales para verificar que el tiempo de ejecución del algoritmo cumpliera con la cota asintótica de $(O((\prod_{i=0}^{M-1} K_i) * (\sum_{i=0}^{N-1} C_i))$, teóricamente demostrada para el peor caso. Las pruebas que se realizaron consistieron de probar con distinta cantidad de tamaños para 1 a 3 mochilas, donde la cantidad de tesoros era igual a los tamaños de las mochilas. El objetivo era observar el tiempo que toma el algoritmo cambiando estos valores. Para cada tesoro, su peso y valor fue un número random entre 1 y el tamaño de la mochila, tomado utilizando random de la librería estandar y con una distribución uniforme. El resultado de este experimento puede visualizarse en este gráfico:

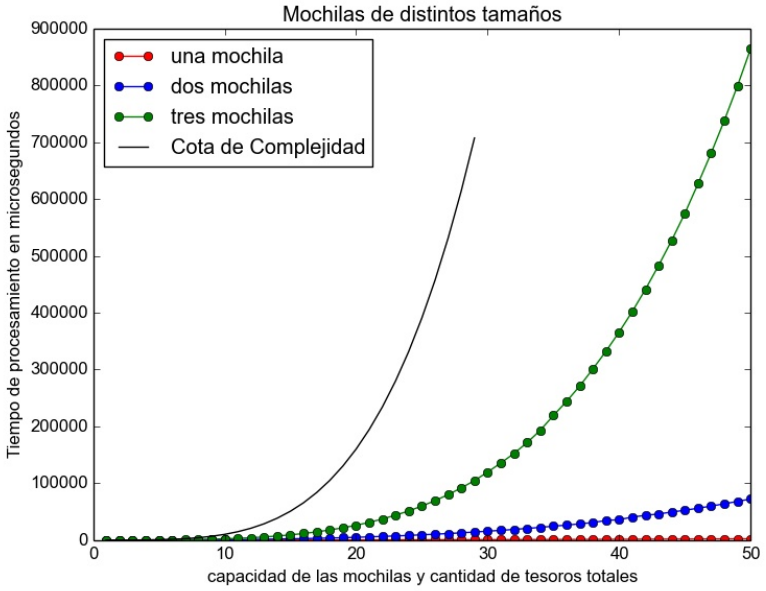

Figura 8

Luego en base a los resultados, se puede observar que dentro de las variables es la cantidad de mochilas una de las que influye fuertemente en el tiempo de ejecución. Esto se debe pues, como fue explicado en la **solución propuesta**, a que las dimensiones de las matrices a recorrer, son del tamaño de la capacidad de las mochilas. Por esto a mayor cantidad de mochilas y mayor capacidad, el tiempo de ejecución es mayor. Por otro lado se puede obesrvar, que la complejidad teórica calculada es correcta según la experimentación.

A. Informe de modificaciones

A continuación se señalan los cambios realizados para la reentrega del trabajo:

- Ejercicio 1
  - Se eliminó la decisión de tomar a los dos más rapidos en cada caso. Se reemplazó por abarcar todos los casos posibles de mandar a cada uno de los arqueólogos y caníbales.
  - Se agregó y modificó el pseudocódigo en solución propuesta.
  - Se ajustó la cota de complejidad con las modificaciones de código hechas. Y se demostró.
  - Se agregó la demostración de correctitud.
  - Se volvió a explicar más detallado los experimentos
  - Se modificaron los gráficos para que estén más claros.
  - Se agregó el tercer experimento.
- Ejercicio 2
  - En el gráfico de la experimentación se ignoraron los primeros valores para no tener picos al comienzo. Esto además se aclara en la descripción de la sección.
  - En el gráfico de la experimentación se añadió la cota de complejidad teórica calculada.
  - Se modificó la explicación de la correctitud (más precisamente la explicación en cuestión de los restos) para que la misma sea más clara y concisa.
- Ejercicio 3
  - Se modificó la cota de complejidad por $\mathrm{O}((\prod_{i=0}^{M-1} K_i) * (\sum_{i=0}^{N-1} C_i))$
  - En la experimentación se cambió las escalas de las funciones para que sean posibles de visualizar y comparar.