

1. Problema 1: Cruzando El puente

1.1. Introducción

El problema trata sobre un grupo de arqueólogos y caníbales que quieren cruzar un puente pero se encuentran con el problema de que solo disponen de una sola linterna para cruzar el puente, solo pueden cruzar como mucho dos personas y no pueden dejar que los caníbales sean mayoría en ningún momento pues estos últimos se los comerían.

Ejemplo de instancia resuelta?

1.2. Resolución

Para la resolución vamos permutando las instancias pero con la restricción del enunciado, que es sólo poder mandar una o dos personas desde donde está la linterna hacia el otro lado, pasando la linterna también.

La instancia inicial es: todas las personas de un mismo lado y la linterna a la izquierda.

Usamos la técnica de backtracking, la instancia inicial sería la raíz, y vamos bajando a medida que permutamos. El primer problema que se presenta es cómo no repetir permutaciones en una misma rama para no entrar en una rama infinita, para ésto cada padre va agregando a su hijo al conjunto de instancias, y le pasas este conjunto en la recursión al hijo. Cuando el hijo vuelve de la recursión el padre lo saca del conjunto (por si la permutación se repite pero en otra rama o sub-rama a partir del padre).

1.2.1. Justificación de la solución

Identificación de cada instancia de manera única: Cada instancia consiste de: una lista de persona en la izquierda, una lista de personas en la derecha y si la linterna esta en la izquierda o en la derecha. Además podemos identificar cada instancia sólo mirando la lista izquierda y dónde está la linterna (no es necesario además mirar la lista derecha ya que dado un grupo de personas iniciales y dos listas izquierdas iguales, las listas de la derecha deben ser iguales).

Para identificar cada instancia de una forma numérica usamos el Teorema fundamental de la aritmética, inicialmente a cada persona le asignamos un número primo distinto p_i .

Al hacer una productoria de los primos de la lista izquierda nos queda un numero que identifica a esa instancia sin tener en cuenta la linterna. Multiplicamos por -1 si la linterna está a la derecha o 1 si está a la izquierda.

Por lo tanto cada instancia queda definida numéricamente como:

$$ID(instancia) = \prod_{i \in \text{izquierda}} p_i * \delta(instancia)$$

Donde:

$$\delta(instancia) = \begin{cases} 1 & \text{si la linterna está del lado izquierdo} \\ -1 & \text{si la linterna está del lado derecho} \end{cases}$$

Estructura del backtracking: Para entrar a la raíz del árbol, y a los demás nodos nos metemos con esta función, que se llama una vez:

`auxResolver(izq, vacia, tiempoOptimo, tiempoActual, linternaIzq, instancias)`

Donde:

También pueden solo guardar la lista de person

- Muestren pseudocódigo a alto nivel, no está bien explicar secciones de código.

- izq: Es la lista de personas que están en la izquierda, en el caso inicial están todas las personas con sus números primos ya asignados
- der: Van a estar todos los que están en la derecha, en el caso inicial está vacía.
- tiempoOptimo: En esta variable nos vamos guardando el tiempo óptimo, al principio vale cero, y mientras valga 0 no habremos llegado a ninguna solución aún (Si al volver de toda la recursión sigue siendo cero, es porque no se pudo encontrar ninguna solución al problema inicial)
- tiempoActual: Nos vamos guardando el tiempo que vamos sumando a medida que pasamos personas al otro lado, inicialmente es 0.
- instancias: es un conjunto tipo árbol que vamos manteniendo a medida que bajamos por la rama. Es decir, el padre inserta el ID del hijo al que va a saltar la recursión, y lo borra cuando el hijo vuelve, para así poder bajar por otra rama de instancias.
Es necesario que el padre borre al hijo del conjunto cuando éste vuelve, ya que al bajar por otra rama está bien que se repitan instancias de otras sub-ramas, porque sería una solución distinta.

Cada vez que entramos a un nodo tenemos:

```
if(izq.size() == 0){
    if(tiempoOptimo == 0) return tiempoActual;
    else return (tiempoActual < tiempoOptimo ? tiempoActual : tiempoOptimo);
}
else if(tiempoActual > tiempoOptimo && tiempoOptimo != 0) return tiempoOptimo;
```

estos if se entabla si comenza
tempo Optimo = ∞

es más claro min(tiempoActual, tOptimo)

- Chequeamos si la lista nos viene vacía, es decir llegamos a una solución. En caso afirmativo tenemos que ver si actualizar tiempoOptimo o no.
Si tiempoOptimo es 0 significa que llegamos a nuestra primera solución, por lo tanto tenemos que poner nuestro tiempoActual como el óptimo.
Si tiempoOptimo es mayor que 0 entonces simplemente compararemos el actual con el óptimo para ver si hay que actualizar o no.
- Si no llegamos a una solución y alguna vez llegamos a una (o sea tenemos un tiempoOptimo mayor a 0) entonces tenemos la posibilidad de fijarnos si podemos hacer una poda. Esto es, si el tiempoActual es mayor que el tiempoOptimo quiere decir que las soluciones que encontramos a partir de este nodo van a ser peores que lo que ya tenemos, por lo tanto no vale la pena seguir bajando por el árbol.

Armar la instancia "hijo":

- Linterna derecha: Si la linterna está en la derecha la instancia siguiente la armamos mandando de a uno, cuando ya recorrimos la lista der de esa forma, empezamos a mandar de a dos a la izquierda.
- Linterna izquierda: En este caso comenzamos a mandar de a dos, y luego mandamos de a uno. Esto lo hacemos porque en todas las soluciones que intentamos a mano, en ninguna había que mandar de a dos personas de izquierda a derecha, por lo tanto nos pareció pertinente mandar así para poder llegar a una primera solución de manera más rápida.
Sin embargo puede ser que en algún problema haya que mandar de a uno de izquierda a derecha, entonces también tenemos que intentar buscar esa posible solución.
- Una vez armada la instancia siguiente, calculamos el tiempo máximo de la o las personas que mandamos, para sumárselo a tiempoActual.

```

tiempoActual+= tMAX;

Integer instID = sacarID(auxIzq, linternalIzq);
if(valido(auxIzq) && valido(auxDer) && !instancias.contains(instID)){

    instancias.add(instID);

    tiempoOptimo = auxResolver( auxIzq, auxDer, tiempoOptimo, tiempoActual,
        !linternalIzq, instancias);

    instancias.remove(instID);
}
tiempoActual-=tMAX;

```

- ✓ ■ Al principio sumamos a tiempoActual el máximo de las dos o una personas que tenemos para mandar. Como se ve, la instancia candidata ya está armada en listas auxiliares.

■ ~~Integer instID = sacarID(auxIzq, linternalIzq);~~

Nos guardamos en *instID* el valor numérico de la instancia, lo hacemos de la misma manera que explicamos antes.

■ ~~if(valido(auxIzq) && valido(auxDer) && !instancias.contains(instID))~~

Chequeamos si nos quedó una instancia válida, o sea mas arqueólogos que caníbales o sólo caníbales, en cada lado. Y también si la instancia que armamos ya la repetimos mas arriba en la rama.

Como explicamos antes, repetir instancias en una misma rama no tendría sentido para la solución óptima e incluso tendríamos una rama infinita.

■ ~~instancias.add(instID);~~

~~tiempoOptimo = auxResolver(auxIzq, auxDer, tiempoOptimo, tiempoActual, !linternalIzq, in~~

~~instancias.remove(instID);~~

Como la instancia es válida y no está repetida en la rama, entonces la agregamos al conjunto de instancias por las que ya pasamos.

Luego nos metemos en la recursión con lo que tenemos y al volver sacamos la instancia del conjunto de instancias de la rama, esto es para que no influya en otras ramas distintas o sub-ramas de este nodo.

- Por último actualizamos el tiempoActual restándole el tiempo que habíamos sumado de las personas que mandamos, para que quede el tiempoActual del nodo actual.

Nota: Cuando volvemos a la primer recursión original, nos fijamos si el tiempoOptimo es 0, en ese caso no llegamos a ninguna solución y devolvemos -1, si no, devolvemos el tiempoOptimo.

1.3. Cota de Complejidad

Puede estimarse una cota superior acotando ~~grossamente~~ una rama, y así obteniendo la altura del árbol.

Ya que lo único que hacemos es ir permutando las instancias a medida que bajamos por una rama, mirando toda la rama puedo tener todas las permutaciones posibles de la lista inicial junto con la linternal. Esta cantidad es la suma de combinaciones de ir tomando de a dos elementos, de a tres,

4/19
Describir en una sección aparte todas las podas y estrategias. Yo vi una de che según lo escrito, sería importante que haya más.

Como no tienen podas, ^{compleja} no tienen la parte de demostrar que una poda no tiene soluciones y no lo puedo evaluar \therefore (correctitud de las podas)

1.4 Experimentación

1 PROBLEMA 1: CRUZANDO EL PUENTE

etc. Todo esto multiplicado por 2 por la interna. Entonces, siendo A la altura del árbol y n la cantidad de personas:

$$A = 2 * \sum_{i=1}^n \binom{n}{i} = 2 * (2^n - 1) \quad (\text{pelear por } g)$$

Para obtener la cantidad de hojas, falta calcular la cantidad de hijos que tiene un nodo. Como vamos mandando de a una persona, y luego de a dos, es claro que cada nodo tiene $\binom{n}{1} + \binom{n}{2} = n + (n * (n - 1)) / 2 = ((n^2 + n) / 2)$ hijos.

Por lo tanto la cantidad de hojas es: $((n^2 + n) / 2)^A$

El costo en cada nodo es:

- $\mathcal{O}(2n)$ por clonar dos listas.
- $\mathcal{O}(n + \log(n))$ por fijarse si una instancia es válida y fijarse en el conjunto de instancias si no es repetida.
- $\mathcal{O}(2\log(n))$ por agregar y borrar sobre el conjunto de instancias recorridas.
- En total la suma de todos estos costos es $\mathcal{O}(n)$

Cota total: Por lo tanto la complejidad total es $\mathcal{O}(((n^2 + n) / 2)^A) * \mathcal{O}(n) = \mathcal{O}((n^2)^A * n) = \mathcal{O}(n^{2A+1})$

1.4. Experimentación

Para realizar pruebas sobre el algoritmo implementado, dividimos los casos de entrada en 4 tipos:

- **Igualas:** Todas las velocidades de las personas son iguales. Para realizar las pruebas las seteamos todas en 10, pero este valor es anecdótico, ya que no influye realmente en el rendimiento del programa.
- **Pocas Diferencias:** Las velocidades de las personas varían en un rango acotado. Nos pareció interesante testear casos donde la diferencia de velocidades no sea grande. Para las pruebas seteamos valores entre 10 y 20.
- **Algunas Diferencias:** Las velocidades de las personas varían en un rango mucho mayor al caso anterior, pero sigue siendo un rango acotado. Este caso es interesante para verificar las podas agregadas al algoritmo y como utiliza estas diferencias de velocidades para descartar casos. Para realizar las pruebas tomamos los valores 10 y 500 y los repartimos entre las personas.
- **Uno Muy Diferente:** La velocidad de uno difiere mucho con la del resto de las personas. Este caso también es interesante de ver para verificar que las podas realmente sirvan. Para realizar las pruebas le asignamos a una persona el valor 500 y al resto el valor 10.

Para cada uno de los casos mencionados vamos a correr todas las combinaciones de personas posibles ~~repetidas veces~~. Mediremos el tiempo de ejecución y tomaremos la mediana de estos valores para evitar outliers y otros inconvenientes. Luego observaremos y analizaremos los resultados.

Antes que nada queremos aclarar que la medida utilizada para los tiempos de ejecución es un número que solo utilizaremos de guía para aproximar diferencias. No es una medida estándar, como pueden ser los segundos, microsegundos o ticks del procesador.

En los primeros cuatro gráficos (Figura 1, Figura 2, Figura 3, Figura 4) están plasmados los tiempos de ejecución de las distintas combinaciones de personas posibles en cada uno de los 4 casos mencionados previamente. Se quitaron del gráfico las combinaciones '6 arqueólogos y 0 caníbales' y '0 arqueólogos y 6 caníbales' porque su tiempo de ejecución era muy elevado en comparación al

unidad?

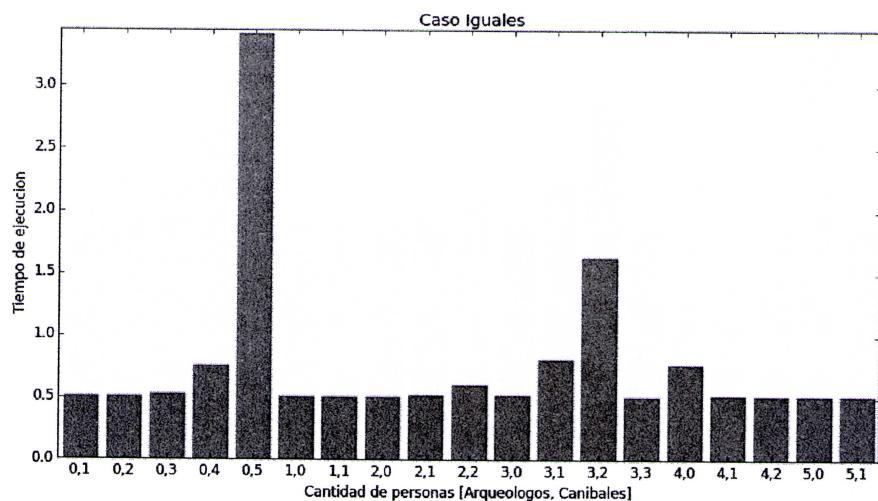


Figura 1: Caso Iguales

unidad?

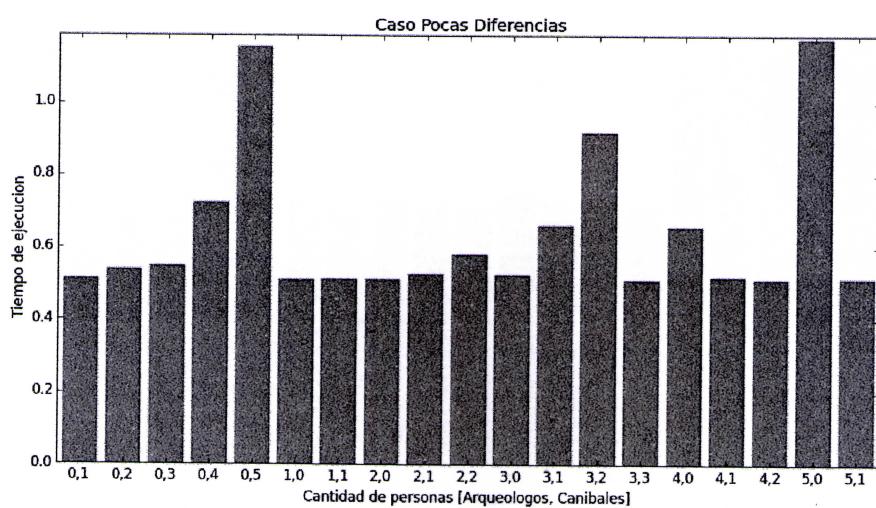


Figura 2: Caso Pocas Diferencias

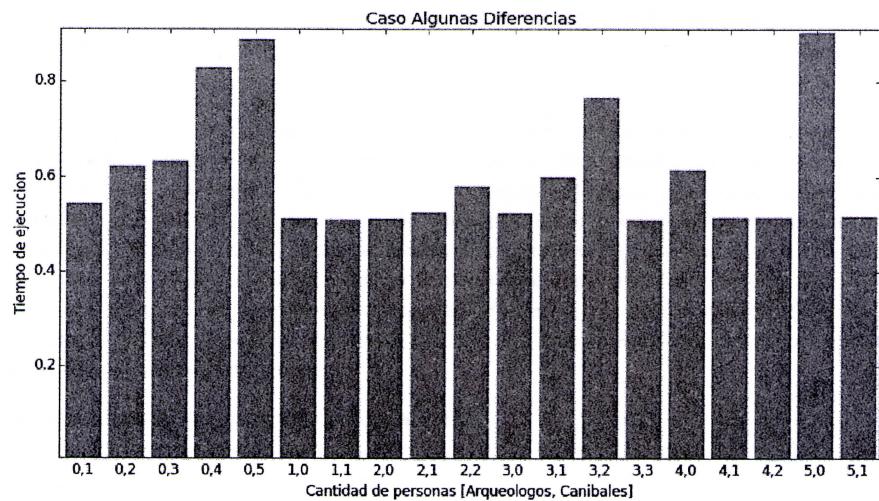


Figura 3: Caso Algunas Diferencias

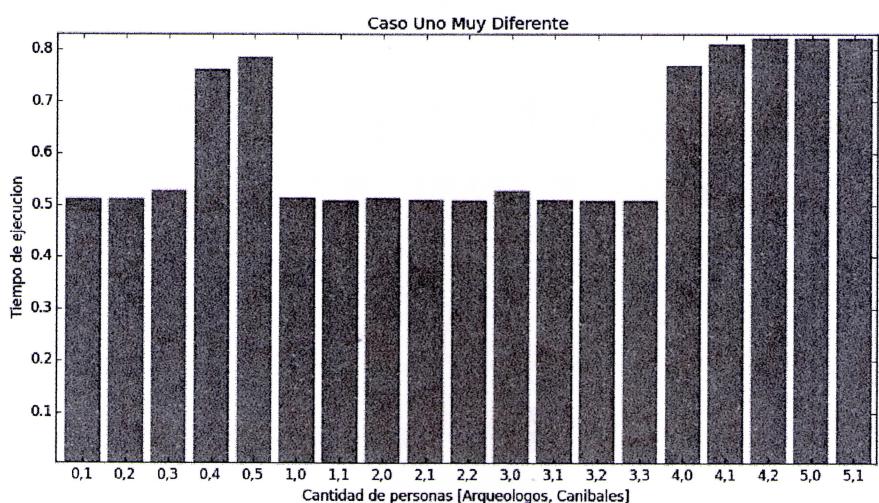


Figura 4: Caso Uno Muy Diferente

resto y los gráficos se hacían difíciles de leer. Recordemos que la complejidad estaba basada, entre otras cosas, de la cantidad de personas. y en este caso la complejidad resultante es muy elevada.

Algo que podemos notar en el gráfico es que los picos más altos se dan en combinaciones de personas con un tipo de persona (caníbal o arqueólogo) igual a cero ó en casos de muchas personas. El resto de los casos se mantienen en un número similar. Vamos a ver estos dos casos que nombramos:

- Caso con un valor igual a cero:

Para este caso hay que remitirse a la condición más importante que se debe respetar: *No puede haber en ningún momento más personas del tipo caníbal que del tipo arqueólogo en un mismo lado del puente*. Esta condición es muy relevante para este caso, ya que al ser la cantidad de uno de los dos tipos de personas igual a cero (no hay caníbales o no hay arqueólogos) entonces esta condición siempre se cumple. Lo cual nos lleva a que no hay casos que se puedan evitar (podar), se deben tomar todas las variantes posibles. Esto hace que su tiempo de ejecución se amplie notablemente. Podemos ver en los gráficos (Figura 1, Figura 2, Figura 3, Figura 4) que los valores 0,1, 0,2 y 0,3 van en aumento, mientras que los valores 0,4 ya son notablemente grandes

- Caso con cantidad de personas cercana a 6:

Observando los resultados se nota que otro pico en los tiempos de ejecución se da en la combinación '3 arqueólogos y 2 caníbales'. Es en estos casos donde notamos que la cota de complejidad depende en gran parte de la cantidad de personas. Pero también hay otras podas que influyen y en algunos casos hacen que esta cota disminuya. Para el caso '3 arqueólogos y 1 caníbal' y '3 arqueólogos y 2 caníbales' se observa que la cantidad 3 de arqueólogos influye en que se pueden elegir más movimientos sin caer en ninguna condición que no lo permita. El primero de estos dos casos es más particular, ya que en ningún momento puede quedar en superioridad numérica una persona de tipo caníbal, por lo que esa condición no influye (similar al caso con un valor igual a cero).

Falta

• Comprobación de la complejidad teórica?

• Comparación de tiempos d y s/podas y estrategias. Para esto van a necesitar pensar más podas -

• Consideren gráficos de línea donde pueda verse la evolución respecto del tiempo de una variable



2. Problema 2: Problemas en el camino

2.1. Introducción

El problema a resolver se trata de una balanza que se encuentra desequilibrada y debemos, por medio de diversas pesas con las que contamos, devolverle el equilibrio.

La balanza, en un principio, se encontraba en equilibrio. Quitaron una llave (de la cual conocemos su peso) de en uno de los platillos y la balanza se desequilibró. Por suerte, contamos con una serie de pesas con la particularidad de que todos sus pesos son potencias de 3 y no hay repetidas. Por lo tanto queremos determinar qué pesas se deben colocar en cada platillo para reestablecer el equilibrio.

Siendo más específicos, a partir de una entrada $P = \text{'Peso de la llave quitada'}$ de tipo entero debemos encontrar una combinación de potencias de 3 multiplicadas por +1 o -1 nos dé como resultado P . Esto se resume en:

$$\forall P \in \mathbb{N}, P = \sum_{i=0}^n 3^i * a_i \quad (1)$$

(esta es la que queremos ver si v)

con $a_i \in \{-1, 0, +1\}$, según el caso para algún $n \in \mathbb{N}_0$

2.2. Idea General de Resolución

Continuando con la notación utilizada en la introducción, vamos a describir el algoritmo que utilizamos para resolver el problema.

El valor n (exponente máximo de las potencias de 3) que utilizaremos es $\lceil \log_3(P) \rceil$ (el por qué de la elección de este valor se explicará en el *Lema 2 del Ej2* en la subsección 2.3.1).

El algoritmo, en primer lugar, determina cuál es el valor de $a_n \in \{-1, 0, +1\}$. Esto lo hace quedándose con el que cumpla $|P - a_n * 3^n| < \frac{3^n}{2}$ (para comprender el por qué de este valor ver el *Lema 1 del Ej2* en la subsección 2.3.2). Luego, reemplaza a P por $P - a_n * 3^n$ y a n por $n - 1$ y se llama recursivamente con los valores actualizados de P y n . Así, va determinando los valores de cada a_i (desde $i = n$ hasta $i = 0$).

2.2.1. Ejemplo ej2:

Veamos un ejemplo del funcionamiento del algoritmo. Tomamos como entrada el número $P = 12$, tenemos $n = \lceil \log_3(12) \rceil = 3$. El algoritmo compara entre los 3 casos posibles para a_k y calcula la diferencia con P :

- $a_3 = -1 \rightarrow 12 - (-1) * 3^3 = 39.$
- $a_3 = 0 \rightarrow 12 - (0) * 3^3 = 12.$
- $a_3 = +1 \rightarrow 12 - (+1) * 3^3 = -15.$

Como $\frac{3^3}{2} = 14,5$, se determina que $a_3 = 0$. Se actualiza $P = 12$ y $n = n - 1$ y se realizan nuevamente los mismos pasos. Se comparan los 3 casos:

- $a_2 = -1 \rightarrow 12 - (-1) * 3^2 = 21.$
- $a_2 = 0 \rightarrow 12 - (0) * 3^2 = 12.$
- $a_2 = +1 \rightarrow 12 - (+1) * 3^2 = 4.$

Ahora tenemos que $\frac{3^2}{2} = 4,5$, entonces se determina que $a_2 = +1$. Se actualiza $P = 12 - (+1) * 3^2 = 4$ y $n = n - 1$ y se continúa. Se comparan los 3 casos:

2.3 Presentación del algoritmo y demostración PROBLEMA 2: PROBLEMAS EN EL CAMINO

- $a_1 = -1 \rightarrow 4 - (-1) * 3^1 = 7$.
- $a_1 = 0 \rightarrow 4 - (0) * 3^1 = 4$.
- $a_1 = +1 \rightarrow 4 - (+1) * 3^1 = 1$

✓ En este caso tenemos $\frac{3^1}{2} = 1,5$, por lo tanto se determina que $a_1 = +1$. Actualizamos $P = 3 - (+1) * 3^1 = 0$. Como $P = 0$, terminamos de determinar todos los valores a_i necesarios. Entonces nos queda que $12 = (+1) * 3^2 + (+1) * 3^1$.

2.3. Presentación del algoritmo y demostración de correctitud

Antes que nada, vamos a enunciar dos lemas que nos servirán para entender y demostrar el correcto funcionamiento del algoritmo.

2.3.1. Lema 1 del Ej2

Queremos probar que dado un $P \in \mathbb{N}$ y dado un $k \in \mathbb{N}_0$, si $|P| \leq \frac{3^{k+1}}{2} \Rightarrow \exists a_k \in \{-1, +1, 0\}$ tal que $|P + a_k * 3^k| \leq \frac{3^k}{2}$.

Demostración:

Dado $P \in \mathbb{N}$, $k \in \mathbb{N}_0$ tal que $|P| \leq \frac{3^{k+1}}{2}$. Tenemos 3 posibilidades:

- Si $|P| \leq \frac{3^k}{2} \Rightarrow$ cumple con $a_k = 0$.
- Si $P > \frac{3^k}{2}$: sabíamos que $P \leq \frac{3^{k+1}}{2}$, entonces tenemos que

$$\frac{3^k}{2} < P \leq \frac{3^{k+1}}{2} \quad (2)$$

Sumando $-1 * 3^k$ tenemos

$$\frac{3^k}{2} - 1 * 3^k < P - 1 * 3^k \leq \frac{3^{k+1}}{2} - 1 * 3^k \quad (3)$$

$$\Leftrightarrow -\frac{3^k}{2} < P - 1 * 3^k \leq \frac{3^k}{2} \quad (4)$$

Concluimos entonces que

$$|P - a_k * 3^k| \leq \frac{3^k}{2} \quad (5)$$

✓ con $a_k = -1$

- Si $P < -\frac{3^k}{2}$: sabíamos que $P \geq -\frac{3^{k+1}}{2}$, entonces tenemos que

$$-\frac{3^{k+1}}{2} \leq P < -\frac{3^k}{2} \quad (6)$$

Sumando $+1 * 3^k$ tenemos

$$-\frac{3^{k+1}}{2} + 1 * 3^k \leq P < -\frac{3^k}{2} + 1 * 3^k \quad (7)$$

$$\Leftrightarrow -\frac{3^k}{2} \leq P < \frac{3^k}{2} \quad (8)$$

Concluimos entonces que

$$|P - a_k * 3^k| \leq \frac{3^k}{2} \quad (9)$$

✓ con $a_k = +1$

De esta manera pudimos ver que en las 3 posibilidades se cumple el lema planteado.

2.3 Presentación del algoritmo y demostración PROBLEMA 2: PROBLEMAS EN EL CAMINO

2.3.2. Lema 2 Ej2

Queremos probar que $P \leq \frac{3^{n+1}}{2}$ con $n = \log_3(P) \forall P \in \mathbb{N}_0$.

Demostración:

Tenemos $P \in \mathbb{N}_0$. Queremos ver que cumple:

$$P \leq \frac{3^{\log_3(P)+1}}{2} \quad (10)$$

$$\Rightarrow P \leq \frac{3^{\log_3(P)} * 3}{2} \quad (11)$$

$$\Rightarrow P \leq \frac{P * 3}{2} \quad (12)$$

$$\Rightarrow 0 \leq \frac{1}{2} * P \quad (13)$$

Como $P \in \mathbb{N}_0$, sabemos que se cumple. Por lo tanto tenemos que $P \leq \frac{3^{n+1}}{2} \forall P \in \mathbb{N}_0$ con $n = \log_3(P)$. *OK, luego hay que decir que $P \leq \frac{3^{\log_3(P)+1}}{2} \leq \frac{3^{\log_3(P)}}{2}$*

Luego de los dos lemas planteados, podremos demostrar la correctitud del algoritmo usado. Para esto, veamos el pseudocódigo del algoritmo:

Algoritmo 1: Función encargada de calcular el valor asignado al exponente pasado como parámetro, según el número pasado por parámetro.

```

1 Function DameCombinacion(P: int, exponente: int, listaPositivos: lista(int), listaNegativos: lista(int)):
2   if exponente < 0 then
3     | return
4   end
5   potenciaActual ←  $3^{\text{exponente}}$ 
6   if  $|P - (+1) * \text{potenciaActual}| < \frac{3^{\text{exponente}}}{2}$  then
7     | agregar exponente al principio de listaPositivos
8     | P = P - (+1) * potenciaActual
9   else if  $|P - (-1) * \text{potenciaActual}| < \frac{3^{\text{exponente}}}{2}$  then
10    | agregar exponente al principio de listaNegativos
11    | P = P - (-1) * potenciaActual
12   DameCombinacion(P, exponente - 1, listaPositivos, listaNegativos)
13 EndFunction

```

Analicemos el algoritmo siguiendo el pseudocódigo. En la linea 5 se calcula la potencia. Luego se comparan los casos de a_k (lineas 6, 9) según el criterio del *Lema 1 del Ej2* (subsección 2.3.1). Si el valor elegido es $+1$, se agrega el exponente a la lista de exponentes positivos y se actualiza el número P . En cambio si el valor elegido es -1 , se agrega el exponente a la lista de exponentes negativos y se actualiza el número P . Si no entró en ninguno de los casos anteriores se determina que el valor de a_k es 0. Esto lo podemos asegurar porque el lema nos decía que sí o sí uno de los 3 valores cumple la implicación, como ninguno de los otros dos casos cumplía, sabemos que el 0 si cumple. Luego se llama recursivamente con los valores P , *listaPositivos*, *listaNegativos* y *exponente* decrementado en 1.

Si P y *exponente* cumplen las condiciones necesarias ($|P| \leq \frac{3^{\text{exponente}+1}}{2}$), sabemos que al final de la función se realiza una llamada recursiva a sí misma con valores P y *exponente* que cumple

* Mencionar que cuando $\exp = 0$, por el bucle se garantiza que el P restante cumple $|P| \leq \frac{3^{\exp}}{2} = 0,5 \Rightarrow P = 0$ (si no, cómo se que termina bien?)

2.4 Cota de Complejidad

2 PROBLEMA 2: PROBLEMAS EN EL CAMINO

$|P| \leq \frac{3^{\exp} + 1}{2}$ (teniendo en cuenta que *exponente* se disminuyó en 1). Por lo tanto, al ingresar a la función en la llamada recursiva cumple las condiciones necesarias. Este ciclo se da hasta que *exponente* es menor que 0. En este caso (en la linea 3) se termina la función.

Cabe destacar además, que para cada exponente la función solo se llama una vez. Esto nos asegura que los exponentes no se repitan y solo haya un a_k asignado a un exponente. ✗

Volviendo un poco al problema planteado en la introducción, veamos como se utiliza la función planteada para resolverlo. Vamos a llamar a la función con los valores: $P =$ el peso de la llave; $\exp = \log_3(P)$; *listaPositivos* y *listaNegativos* dos listas vacías. Luego de que la función termine su ejecución (y sus respectivas llamadas recursivas), en las listas *listaNegativos* y *listaPositivos* estarán los valores de los exponentes necesarios para equilibrar la balanza.

2.4. Cota de Complejidad

calcular la exponencial O(1)? Por qué?

Vamos a dar una cota de complejidad del algoritmo y analizaremos el por qué de ese valor. Para esto observaremos el pseudocódigo del Algoritmo 1.

Veamos la función *DameCombinacion*. En las líneas 2-4 hay una comparación y la salida de la función de costo $O(1)$. Luego, en la línea 5 hay una asignación de una exponencial que tiene costo $O(1)$. Las guardas de las líneas 6 y 9 tienen costo $O(1)$. En las líneas 7 y 10 se agregan dos enteros al inicio de dos listas. La implementación utilizada para esas listas nos asegura que esa acción tiene costo $O(1)$. Luego en las líneas 8 y 11 se realizan dos asignaciones con costo $O(1)$. Luego en la línea 12 se ejecuta la llamada recursiva. Por lo tanto, para una ejecución de la función (sin contar la llamada recursiva) se obtiene un costo de $O(1)$.

Recordemos que la función es llamada con exponente $\log_3(P)$, y este exponente se disminuye de a 1 antes de hacer una llamada recursiva. Entonces obtenemos una complejidad $O(\log_3(P)) * O(\text{DameCombinacion}) = O(\log_3(P)) * O(1) = O(\log_3(P))$.

Luego, para imprimir los resultados en el formato pedido, se recorren las listas *listaNegativos* y *listaPositivos*. Esto nos lleva $O(\text{longitud}(\text{listaNegativos}) + \text{longitud}(\text{listaPositivos}))$. Pero sabemos que a lo sumo se recorren $\log_3(P)$ exponentes y ninguno se repite, por lo que sabemos entonces que $\text{longitud}(\text{listaNegativos}) + \text{longitud}(\text{listaPositivos}) \leq \log_3(P)$. Por lo tanto imprimir los resultados tiene costo $O(\log(P))$.

En conclusión, la cota de complejidad del algoritmo utilizado para resolver el ejercicio 2 es $O(\log(P))$.

Para este problema debíamos cumplir una cota de complejidad $O(\sqrt{P})$. Como $\log_3(P) \leq \sqrt{P}$ podemos decir que cumplimos la cota pedida.

2.4.1. Justificación

citar por qué
deberíamos

2.5. Compilación y ejecución

2.6. Casos de Test

2.7. Experimentación

Como se puede ver en la figura 1 nuestra hipótesis de que la complejidad de nuestro algoritmo es $O(\log(P))$ se verifica.

No hay experimentos. Recuerden poner peores y mejores casos, explicar cómo se generan y corren los casos, etc.

3. Problema 3: Guardando el tesoro

3.1. Introducción

En este problema unos arqueólogos quieren llevarse algunos tesoros, para esto tienen algunas mochilas (a lo sumo tres). Los tesoros tienen un peso y un valor, y el objetivo es organizar los tesoros de tal manera que el valor total de los tesoros llevados sea el máximo posible. La dificultad radica en que la capacidad de las mochilas es limitada (podrían tener distinta capacidad cada una). Formalmente, dada una lista de tesoros representados como una tupla $(peso, valor)$ y m enteros K_i representando la capacidad máxima de las mochilas ($0 < m \leq 3$), se piden m listas de tesoros tales que la sumatoria de los pesos de la i -ésima lista sea menor o igual a K_i y que la sumatoria de los valores de todos los tesoros de las m listas sea el máximo posible.

3.2. Resolución

Es bueno dar una explicación de alto nivel antes de la fórmula

Este problema claramente es muy similar al *knapsack problem*, pero con la variante de que hay varias mochilas para llenar. La técnica algorítmica utilizada es programación dinámica. La función recursiva que devuelve el valor máximo posible nosotras la definimos así:

```
maximizar_beneficio(0, m1, m2, m3) = if peso(0) <= m1 || peso(0) <= m2 || peso(0) <= m3 then valor(tesoros0)
maximizar_beneficio(i, m1, m2, m3) = max(if peso(i) <= m1 then valor(i) + maximizar_beneficio(i-1, m1 - peso(i), m2, m3) else 0,
if peso(i) <= m2 then valor(i) + maximizar_beneficio(i-1, m1, m2 - peso(i), m3) else 0, if peso(i) <= m3 then valor(i) + maximizar_beneficio(i-1, m1, m2, m3 - peso(i)) else 0, maximizar_beneficio(i-1, m1, m2, m3))
```

Vamos a explicar esta función, el primer parámetro es el índice de la lista de tesoros y los demás son las respectivas capacidades disponibles en cada mochila. Sabemos que formalmente la función debería tener como parámetro la lista tesoros pero para simplificar hicimos ese abuso de notación. También se pide que el lector asuma definidas las funciones peso y valor que devuelven respectivamente el peso y el valor del tesoro i -ésimo de la lista. Esta función en el caso base se fija si puede colocar el tesoro en alguna mochila y en ese caso devuelve el valor, si no lo puede colocar devuelve 0. En el caso recursivo la función se fija el máximo de cuatro casos. Éstos son: coloco el objeto en la mochila 1, lo coloco en la 2, lo coloco en la 3 o no lo coloco en ninguna. Notar que f también sirve en los casos en los que hay 1 o 2 mochilas, sencillamente se ponen las capacidades en 0.

Como este algoritmo es de programación dinámica además de definir la función recursiva tengo que decidir como voy a implementar la parte de la memoización. Los resultados parciales me los voy a guardar en una matriz que va a tener una dimensión con los índices de cada tesoro y una dimensión por cada mochila con las capacidades parciales posibles, es decir desde cero hasta la capacidad máxima de cada mochila. En cada posición de esta matriz voy a guardarme el beneficio máximo con esas capacidades y la lista de tesoros desde cero hasta i como así también desde que posición de la matriz llegué a ese máximo. Es decir, con cual de los 4 casos del max de maximizar_beneficio obtuve dicho máximo.

3.2.1. Pseudocódigo

Antes del pseudocódigo es oportuno aclarar que cada casillero de la matriz es una tupla de:

- beneficio: int,
- mochila: int,
- visitado: bool,
- hijoWeight1: int,
- hijoWeight2: int,
- hijoWeight3 : int

El campo mochila indica en qué mochila puse el tesoro de esa posición, los campos hijoWeight dan las coordenadas de la matriz desde donde calcule ese máximo. Estos campos van a ser cruciales

para imprimir la salida con el formato pedido.

Decidimos inclinarnos por un *approach top down* o recursivo.

Algoritmo 2: Función encargada de devolver el beneficio maximo y de poner valores en la matriz para luego imprimir los tesoros correspondientes a cada mochila

```

1 Function knapsack(matriz: Casilla[][][], tesoros: lista(int,int), i: int, weight1: int, weight2: int, weight3: int):
2     if i < 0 then
3         | return 0
4     end
5     if visita matriz[i][wieght1][weight2][weight3] then
6         | return matriz[i][wieght1][weight2][weight3].beneficio
7     end
8     else
9         array beneficios[4] if tesoros[i] entra en la mochila 1 then
10            beneficios[0] =
11            valor(i) + knapsack(matriz, tesoros, i - 1, weight1 - peso(i), weight2, weight3)
12        end
13        if tesoros[i] entra en la mochila 2 then
14            beneficios[1] =
15            valor(i) + knapsack(matriz, tesoros, i - 1, weight1, weight2 - peso(i), weight3)
16        end
17        if tesoros[i] entra en la mochila 3 then
18            beneficios[2] =
19            valor(i) + knapsack(matriz, tesoros, i - 1, weight1, weight2, weight3 - peso(i))
20        end
21        beneficios[3] = knapsack(matriz, tesoros, i - 1, weight1, weight2, weight3)
22        indMax ← indiceDelMaximo(beneficios)
23        actualizar la posicion matriz[i][weight1][weight2][weight3] segun indMax
24        return matriz[i][wieght1][weight2][weight3].beneficio
25    end
26 EndFunction

```

Es fácil ver que knapsack computa la función maximizar_beneficio.

Para imprimir los tesoros colocados en cada mochila sencillamente nos situamos en el extremo de la matriz que tiene el último objeto y las capacidades máximas de las mochilas dado que en esa casilla está beneficio acumulado. Luego con los campos hijoWeight voy saltando a otras posiciones de la matriz hasta terminar de recorrer todos los objetos.

3.3. Cota de Complejidad

La complejidad de este algoritmo es $O(K_i m * C)$ donde K_i es la capacidad de cada mochila, m es la cantidad de mochilas y C es la cantidad total de tesoros. Notar que cuando hay menos de tres mochilas los arreglos interiores tienen longitud 1, por ende no empeoran la complejidad.

Nuestra justificación se basa en que necesitamos inicializar nuestro objeto Casilla de la matriz antes de llamar a la función knapsack. Eso tiene un costo de $O(K_i m * C)$.

Luego cuando se llama a la función knapsack también es $O(K_i m * C)$, dado que todos los llamados recursivos se guardan en la matriz. Entonces en el peor caso knapsack llena la matriz. Por último el ciclo que dice en mochila se colocaron los tesoros es $O(C)$

Falta mucho detalle en el análisis de complejidad

Falta probar que se cumple la complejidad pedida $O(\sum_i^C \sum_{k=1}^{K_i} k)$

• Falta demo de corretitud.

3.4. Experimentación

Nosotros tenemos como hipótesis que nuestro algoritmo tarda aproximadamente lo mismo para todos los casos. Los experimentos los hicimos con todos los objetos enormes, es decir que no entran en las mochilas, con todos los objetos cuyo peso es 1 así todos entran y se llenaría más la matriz; y con pesos aleatorios.

- Con objetos enormes:
- Con objetos pequeños:
- Con objetos de peso aleatorio

Incompleto