

Algoritmos y Estructura de Datos III

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

Trabajo Práctico 1

Grupo 1

Integrante	LU	Correo electrónico
Hernandez, Nicolas	122/13	nicoh22@hotmail.com
Kapobel, Rodrigo	695/12	rok_35@live.com.ar
Rey, Esteban	657/10	estebanlucianorey@gmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1	Ejercicio 1	3
1.1	Descripción de problema	3
1.2	Explicación de resolución del problema	3
1.3	Algoritmos	3
1.4	Análisis de complejidades	3
1.5	Experimentos y conclusiones	4
1.5.1	1.5	4
1.5.2	1.5	5
2	Ejercicio 2	8
2.1	Descripción de problema	8
2.2	Explicación de resolución del problema	8
2.3	Algoritmos	10
2.4	Análisis de complejidades	11
2.5	Demostración de correctitud	12
2.6	Experimentos y conclusiones	14
2.6.1	2.5	14
2.6.2	2.5	15
3	Ejercicio 3	17
3.1	Descripción de problema	17
3.2	Explicación de resolución del problema	17
3.3	Algoritmos	17
3.4	Análisis de complejidades	18

1 Ejercicio 1

1.1 Descripción de problema

En este punto y en los restantes contaremos con un personaje llamado Indiana Jones, el cual buscara resolver la pregunta mas importante de la computacion, es $P=NP?$.

Focalizandonos en este ejercicio, Indiana, ira en busca de una civilizacion antigua con su grupo de arqueologos, ademas, una tribu local, los ayudara a encontrar dicha civilizacion, donde se encontrara con una dificultad, la cual sera cruzar un puente donde el mismo no se encuentra en las mejores condiciones.

Para cruzar dicho puente Indiana y el grupo cuenta con una unica linterna y ademas, dicha tribu suele ser conocida por su canibalismo, por lo tanto al cruzar dicho puente no podran quedar mas canibales que arqueologos.

Nuestra intencion sera ayudarlo a cruzar de una forma eficiente donde cruce de la forma mas rapido todo el grupo sin perder integrantes en el intento.

Por lo tanto, en este ejercicio nuestra entrada seran la cantidad de arqueologos y canibales y sus respectivas velocidades.

/// FALTARIA LA DESCRIPCION MAS FORMAL SI ES QUE VA

1.2 Explicación de resolución del problema

Para solucionar este problema, se debe ver la combinacion de viajes entre lados del puente (lado A, origen y lado B, destino) que nos permiten pasar a todos los integrantes del grupo, del lado A al B en el menor tiempo posible. Siendo esta busqueda una tarea exponencial, buscamos la forma de poder disminuir los casos evaluados, acotandonos solo a los relevantes para la consigna, aplicando de este modo la busqueda del mínimo a travez de backtracking.

Para implementar la solución se atomizo cada ciclo a 1 solo viaje: el envio de gente de A a B o el regreso de 'faroleros' de B a A. De esta forma, el backtracking se puede realizar en la desición de la gente que va de A a B, independientemente de la elección de los faroleros.

Dadas las restricciones del problema, se pudieron aplicar las siguientes podas sobre el arbol de posibilidades:

- Las combinaciones evaluadas son, en el caso de enviar 2 personas, a duplas sin repeticiones.
- Los viajes, tanto de paso de A a B como de B a A que generan un 'desbalance', es decir que en algun lado hay más canibales que arqueologos, son obviados.
- Las secuencias de viajes que tomen más tiempo que una previamente calculada se descartan.
- Solo se consideran viajes de A a B en duplas y de una sola persona de B a A.

En base a la evaluación de todas las soluciones encontradas por el algoritmo, nos quedamos con aquella que menor tiempo acumule con los viajes.

1.3 Algoritmos

ACA VA EL PSEUDOCODIGO LA IDEA DE USAR ESTA MACRO ESTA BUENA PORQUE YA PODEMOS IR PONIENDO LA COMPLEJIDAD Y DSP CUANDO NOS TOQUE HACER LA DEMOSTRACION SALE MAS FACIL

Algoritmo 1 CRUZANDO EL PUENTE

```
1: function MINIMOTIEPODECRUCE(in cableSize : Integer, in stationDistances :  
   List<Integer>) → out res: Integer  
2:   List<Integer> distanceDifferences ← List<Integer>(vacio) //O(1)  
3:   Integer lastStation ← 0 //O(1)  
4:   Integer distance ← 0 //O(1)  
5:   while i < stationDistances do //O(N)  
6:     distance ← stationDistances //O(1)  
7:     distanceDifferences.Agregar(distance - lastStation) //O(1)  
8:     lastStation ← distance //O(1)  
9:     i++ //O(1)  
10:  end while  
11:  res ← getMaxRangeLength(cableSize, distanceDifferences) //O(N)  
12: end function  
Complejidad: O(n)
```

1.4 Análisis de complejidades

ACA IRIA LOS COMENTARIOS DE COMPLEJIDAD Y LA DEMOSTRACION

1.5 Experimentos y conclusiones

1.5.1 Test

Por medio de los tests dados por la cátedra, desarrollamos nuestros tests, para corroborar que nuestro algoritmo era el indicado.

A continuación enunciaremos 4 tipos de casos de nuestros tests:

Rollo de cable cubre todas las estaciones

Para este tipo de testeo mostraremos a continuación un ejemplo del mismo, exponiendo su respectivo resultado.

Con un:

Rollo de Cable : 90

Lista de estaciones : 80 81 82 83 84 85 86 87 88 89 90

Obtuvimos el siguiente resultado:

Cantidad de estaciones conectadas : 11

Rollo de cable no cubre ninguna de las estaciones

Para este tipo de testeo mostraremos a continuación un ejemplo del mismo, exponiendo su respectivo resultado.

Con un:

Rollo de Cable : 60

Lista de estaciones : 80 150 220 290 360

Obtuvimos el siguiente resultado:

Cantidad de estaciones conectadas : 0

Rollo de cable con estaciones de igual o similar Kilometraje en referencia a la distancia

Para este tipo de testeo mostraremos a continuación un ejemplo del mismo, exponiendo su respectivo resultado.

Rollo de Cable : 50

Lista de estaciones : 50 60 69 70 130 190

Obtuvimos el siguiente resultado:

Cantidad de estaciones conectadas : 4

Estaciones con bastante distancia intercaladas con estaciones más cercanas

Aquí veremos, un ejemplo del conjunto de test de este tipo, exponiendo su respectivo resultado.

Rollo de Cable : 200

Listadeestaciones : 15 16 17 40 41 42 70 71 72 100 101 102 140 141 142 170 171 172 200 201 202 230 231 232

Obtuvimos el siguiente resultado:

Cantidad de estaciones conectadas : 21

1.5.2 Performance De Algoritmo y Gráfico

Por consiguiente, mostraremos buenos y malos casos para nuestro algoritmo, y a su vez, daremos el tiempo estimado según la complejidad del algoritmo calculada anteriormente.

Luego de varios experimentos, pudimos llegar a la conclusión que uno de los tipos de casos que resulta más beneficioso para nuestro algoritmo es en el cual el rollo de cable llega a cubrir y conectar todas las estaciones.

Para llegar a dicha conclusión trabajamos con un total de 100 instancias y un n entre 1 y 1000000 obtuvimos que nuestro algoritmo finaliza lo solicitado demorando 184 milisegundos.

Para una mayor observacion desarrollamos el siguiente grafico con las instancias:

Si a esto lo dividimos por la complejidad propuesta obtenemos:

Para realizar esta división realizamos un promedio con el mismo input de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más consisos.

A continuación, adjuntamos una tabla con los considerados “mejor” caso que nos parecieron más relevantes

Tamaño(n)	Tiempo(t)	t/n
610000	114000	0,186
650000	121000	0,185
690000	128000	0,185
730000	135000	0,184
770000	142000	0,184
810000	149000	0,183
850000	156000	0,183
890000	163000	0,183
930000	170000	0,182
970000	177000	0,182
1010000	184000	0,182
Promedio		0.217

Dando un **promedio igual a 0.217**

Luego, uno de los peores casos para nuestro algoritmo es en el cual el rollo de cable no llega a cubrir ninguna distancia entre ciudades.

Para llegar a dicha conclusión trabajamos con un total de 100 instancias y un n entre 1 y 1000000 obtuvimos que nuestro algoritmo finaliza lo solicitado demorando 224 milisegundos.

Si a esto lo dividimos por la complejidad propuesta obtenemos:

Para realizar esta experimentación nos parecio acorde, realizar un promedio con el mismo input de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más relevantes.

La información de los 10 datos mas relevantes refiriendonos al peor caso fueron:

Tamaño(n)	Tiempo(t)	t/n
610000	154000	0,251
650000	161000	0,247
690000	168000	0,243
730000	175000	0,239
770000	182000	0,236
810000	189000	0,233
850000	196000	0,230
890000	203000	0,227
930000	210000	0,225
970000	217000	0,223
1010000	224000	0,221
Promedio		0.469

Dando un **promedio igual a 0.469**

Aquí, podemos observar como la cota de complejidad del algoritmo y la de dicho caso tienden al mismo valor con el paso del tiempo.

2 Ejercicio 2

2.1 Descripción de problema

Como habíamos enunciado en el punto anterior, Indiana Jones buscaba encontrar la respuesta a la pregunta es $P = NP$?

Luego de cruzar el puente de estructura dudosa, Indiana y el equipo llegan a una fortaleza antigua, pero, se encuentran con un nuevo inconveniente, una puerta por la cual deben pasar para seguir el camino se encuentra cerrada con llave.

Dicha llave se encuentra en una balanza de dos platillos, donde la llave se encuentra en el platillo de la izquierda mientras que el otro está vacío.

Para poder quitar la llave y nivelar dicha balanza, tendremos unas pesas donde sus pesos son en potencia de 3.

Nuestro objetivo en este punto consistirá en ayudarlos, mediante dichas pesas, a reestablecer la balanza al equilibrio anterior a haber sacado la llave.

2.2 Explicación de resolución del problema

Para solucionar este problema y poder quitar la llave y dejar equilibrado como se encontraba anteriormente realizamos un algoritmo el cual se encuentra dividido en tres partes, la primera realiza lo siguiente:

Como las pesas, presentan un peso en potencia de 3, recorreremos desde 3^0 hasta un 3^i , donde dicho 3^i sea el primero mayor o igual a P .

Guardamos dicho valor en una variable denominada *pesaActual*, crearemos un array el cual nombraremos *arrayPesasUtilizadas*, crearemos también otra variable *equilibrioActual* que inicializaremos con el valor de P y por último, un booleano *estaEnNegativo* con valor Falso.

Luego, la segunda parte consiste de un ciclo en el cual:

Inicialmente, restamos al valor de *equilibrioActual* el valor que teníamos almacenado anteriormente el cual simboliza a la pesa con valor inmediatamente mayor a P o en su defecto igual a P . Una vez obtenida dicha diferencia, la cual denominaremos N chequearemos la misma en módulo con ciertos valores:

- Si N es igual a 0, eso significa que nuestra diferencia entre el valor inicial y la pesa utilizada presentan el mismo peso, dejamos guardado el valor de la pesa en el array *arrayPesasUtilizadas* y finalizamos la segunda etapa.
- Si $|N|$ es igual a ± 1 , eso significa que con agregar la pesa del menor valor que es 1 finalizaríamos la etapa sin la necesidad de chequear el resto de las pesas, por lo cual, guardaremos en nuestro array el valor de las dos pesas y finalizamos esta segunda etapa.
- Si $|N|$ es distinto de estos dos números pero es menor a *equilibrioActual*, esto significa que todavía no se llegó al peso original pero se pudo disminuir por ende pudimos acercarnos al valor que deseamos llegar, por lo cual, agregamos esta pesa al array, restamos el contador para poder realizar la próxima iteración y modificamos el valor de *equilibrioActual* con el valor de

N. Además chequeamos si el valor de N queda por debajo de 0 o no, en caso de quedar debajo cambiamos el valor de *estaEnNegativo* por Verdadero.

- Si N resulta mayor o igual a P no contaremos esta pesa ya que en vez de acercarnos al valor deseado nos alejamos, por lo tanto restamos el contador para poder realizar la próxima iteración.

Además, chequeamos en cada vuelta de ciclo si el valor de *estaEnNegativo* es verdadero o falso, en caso de ser verdadero guardamos el valor de la pesa en *pesaActual* multiplicado por (-1) y de ser falso se guardará con el valor original de la pesa.

Una vez finalizada la segunda etapa tendremos en nuestro array las pesas que se colocarán en la balanza.

Para saber cuales irán del lado derecho y cual del lado izquierdo realizamos esta tercer etapa, en la cual iremos chequeando en cada índice si el valor es positivo o negativo, en caso de que alguno sea negativo esa pesa irá al array *arrayD* previa multiplicación de la pesa con (-1), para dejar la pesa con su valor real y original. Mientras tanto, las positivas, irán en el array *arrayI* para así poder dejar la balanza como se encontraba inicialmente con la llave.

Por ultimo, devolvemos estos array mencionados invirtiendo las posiciones para que los elementos del arreglo queden ordenados de menor a mayor, ya que los mismos por como fue realizado nuestro algoritmo se encuentran de mayor a menor.

2.3 Algoritmos

Algoritmo 2 BALANZA

```

1: function ALGORITMO(in Integer : P)→ out S : Integer out T : Integer out arrayI :
   List<Integer> out arrayD: List<Integer>
2:   i = 0 //O(1)
3:   Recorro desde  $3^0$  hasta  $3^i \geq P$  //O( $\sqrt{P}$ )
4:   equilibrioActual  $\leftarrow$  P //O(1)
5:   creo arrayPesasUtilizadas vacio //O(1)
6:   estaEnNegativo  $\leftarrow$  falso //O(1)
7:   pesaActual  $\leftarrow 3^i$  //O(1)
8:   N = equilibrioActual - pesaActual //O(1)
9:   while Terminar = False do //O( $\sqrt{P}$ )
10:    if N == 0 then //O(1)
11:      arrayPesasUtilizadas  $\cup$  pesaActual //O(1)
12:      Terminar //O(1)
13:    end if
14:    if |N| == 1 then //O(1)
15:      arrayPesasUtilizadas  $\cup$  pesaActual //O(1)
16:      arrayPesasUtilizadas  $\cup 3^0$  //O(1)
17:      Terminar //O(1)
18:    end if
19:    if |N| > 1  $\wedge$  |N| < equilibrioActual then //O(1)
20:      arrayPesasUtilizadas  $\cup$  pesaActual //O(1)
21:      i- //O(1)
22:      equilibrioActual  $\leftarrow$  N //O(1)
23:      if N < 0 then //O(1)
24:        estaEnNegativo  $\leftarrow$  verdadero //O(1)
25:      else
26:        estaEnNegativo  $\leftarrow$  falso //O(1)
27:      end if
28:      equilibrioActual  $\leftarrow$  |N| //O(1)
29:    end if
30:    if |N|  $\geq$  equilibrioActual then //O(1)
31:      i- //O(1)
32:    end if
33:    if estaEnNegativo then //O(1)
34:      pesaActual  $\leftarrow 3^i * (-1)$  //O(1)
35:      pesaUno  $\leftarrow 3^0 * (-1)$  //O(1)
36:    else
37:      pesaActual  $\leftarrow 3^i$  //O(1)
38:      pesaUno  $\leftarrow 3^0$  //O(1)
39:    end if
40:    N = equilibrioActual - pesaActual //O(1)
41:  end while
42:  devolver(armadoBalanza) //O(1)
43: end function

```

Complejidad: $O(\sqrt{P})$

Algoritmo 3 armadoBalanza

```
1: function ARMADOBALANZA( : ) → out  $S$ : Integer out  $T$ : Integer out  $arrayI$ : List<Integer>
   out  $arrayD$ : List<Integer>
2:   creo  $x = 0$  //O(1)
3:   while  $x < arrayPesasUtilizadas.size$  do //O( $\sqrt{P}$ )
4:     if  $arrayPesasUtilizadas[x] < 0$  then //O(1)
5:        $arrayD \cup arrayPesasUtilizadas[x]*(-1)$  //O(1)
6:     else
7:        $arrayI \cup arrayPesasUtilizadas[x]$  //O(1)
8:     end if
9:      $x-$  //O(1)
10:  end while
11:  devolver  $arrayD.size$  //O(1)
12:  devolver  $arrayI.size$  //O(1)
13:  devolver invertido( $arrayD$ ) //O( $\sqrt{P}$ )
14:  devolver invertido( $arrayI$ ) //O( $\sqrt{P}$ )
15: end function
Complejidad:  $O(\sqrt{P})$ 
```

2.4 Análisis de complejidades

Nuestro algoritmo como mencionamos anteriormente presenta 3 etapas.

La primera de ellas consta en recorrer desde 3^0 hasta 3^i donde este ultimo sea igual a P o en su defecto el inmediato mayor. Por lo tanto mostraremos que recorrer hasta un i donde 3^i sea igual o inmediatamente mayor a P es menor o igual a \sqrt{P} .

Si $i = 0 \Rightarrow$ terminamos.

Luego sea $3^i \geq P \geq 3^{i-1}$ con $i > 0$. Queremos ver que $i \leq \sqrt{P}$:

Sabemos que $P \geq 3^{i-1} \Rightarrow \sqrt{P} \geq \sqrt{3^{i-1}}$

Veamos que $\sqrt{3^{i-1}} \geq i \Rightarrow 3^{i-1} \geq i^2$. Para $i = 1$ tenemos que $3^{1-1} \geq 1$ siempre. Luego, para $i > 1$ como 3^{i-1} es creciente y mayor o igual que i^2 se cumple siempre esta desigualdad. Por lo tanto queda probado que recorrer hasta un i tal que $3^i \geq P$ se encuentra en el orden de $O(\sqrt{P})$.

Luego, creamos dos enteros *equilibrioActual* y N y un array *arrayPesasUtilizadas* inicializado vacío, por lo tanto, como son enteros y el array es vacío esto insume $O(1)$, finalizando así la primera etapa.

Luego, la segunda etapa y más importante de nuestro algoritmo, consiste en un ciclo donde realizaremos iterativamente la búsqueda de las pesas que, sumando sus valores, nos de el valor P . Dicho ciclo en el peor de los casos recorrerá desde el valor de i que contenía la pesa de mayor valor hasta $i = 0$, lo cual será $O(\sqrt{P})$ ya que al trabajar con potencias de 3 la cantidad de vueltas del ciclo entraran en el orden de \sqrt{P} como habíamos demostrado anteriormente.

Dentro de este ciclo, realizamos 6 comparaciones en $O(1)$ las cuales son:

- si $N=0$, si $N=1$, aquí agregamos el valor de la pesa al array y terminamos el ciclo, esto insumirá $O(1)$
- si N es menor al valor de *equilibrioActual*, estas opción no finaliza el ciclo, agrega la pesa n al array, modifica el valor de *saldoEnBalanza* por el valor de N y disminuye en 1 a i , luego se chequea aquí mismo si $N < 0$ de ser verdadero se modifica el valor de *estaEnNegativo* por verdadero o por falso en caso de ser falsa la guarda, lo cual insumirá $O(1)$ cada una de las

operaciones mencionadas, luego se chequea aquí mismo si $N < 0$ de ser verdadero se modifica el valor de *estaEnNegativo* por verdadero o por falso en caso de ser falsa la guarda

- por ultimo, si $N \geq equilibrioActual$ solamente descontamos en uno a i para continuar con el ciclo.

Una vez finalizado esto y por consiguiente la segunda etapa, pasamos a la tercera la cual consiste en guardar en *arrayI* o *arrayD* los valores de los elementos del *arrayPesasUtilizadas* para colocarlos en la balanza del lado derecho o del izquierdo.

Para realizar esto recorreremos el array *arrayPesasUtilizadas* el cual en el peor de los casos tendrá todas las pesas, lo cual como vimos recorrer la totalidad de elementos del array se encuentra en el orden de $O(\sqrt{P})$.

Luego de ver esto, dentro del ciclo realizamos operaciones en $O(1)$.

Por ultimo, devolvemos los array invirtiendo las posiciones de los elementos lo que insumirá en el peor de los caso $O(\sqrt{P})$.

En conclusión nuestro algoritmo realiza 4 ciclos que demandan en el peor de los casos $O(\sqrt{P})$ donde dentro de los mismos se realizan operaciones en $O(1)$, por lo tanto nuestra complejidad final será $O(\sqrt{P})$.

2.5 Demostración de correctitud

En nuestro algoritmo como hemos mencionado anteriormente en la explicación del mismo, la etapa más importante es a la hora de obtener las pesas que equilibran la balanza, la segunda, en donde realizamos un ciclo que va disminuyendo el valor *equilibrioActual* hasta llegar a 0.

Queremos demostrar que en cada iteración del ciclo sea posible disminuir dicho valor hasta llegar a 0 siempre y cuando, la diferencia que obtenemos, a la que nombramos *nuevoEquilibrio_i* que representa el nuevo equilibrio en el paso i –esimo con $i > 0$ sea menor que *equilibrioActual*. Recordemos que i decrece dado que representa el exponente actual que es menor en cada iteración y que en los pasos donde *nuevoEquilibrio_i* > *equilibrioActual* se descarta la potencia de exponente i y se prueba con $i - 1$.

Además enunciemos una propiedad que nos será de utilidad:

$$\forall P > 0, P \in \mathbb{N}, \exists i > 0, i \in \mathbb{N} / (3^i \geq P \geq 3^{i-1}) \quad (1)$$

Queremos ver entonces que siempre existirá $i > 0$ tal que :

$$|equilibrioActual - 3^i| = nuevoEquilibrio_i \text{ y } nuevoEquilibrio_i < equilibrioActual$$

Y que además tendremos siempre una potencia no utilizada que lo cumpla, con lo cual al final del algoritmo, cada potencia será única.

Es importante aclarar que en el algoritmo *nuevoEquilibrio_i* podría ser negativo dado que siempre se realiza la resta *equilibrioActual* – 3^i , lo cual, como hemos mencionado, se utiliza en el algoritmo para saber de manera más rápida que las siguientes potencias son las que restan. Aunque es visible que esta condición se cumple cuando $3^i > equilibrioActual$.

Pero para esta demostración haremos las restas de manera tal que siempre de positivo.

Para demostrar que esto es posible, veamos que el ciclo termina:

Sea $equilibrioActual \leq 4$ podemos formarlo con 3 y 1 dado que

- $4 = 3+1$
- $2 = 3-1$

Además si $equilibrioActual = 4$ entonces $4-3 = 1 < 4$ y $1-1 = 0 < 1$.

Si $equilibrioActual = 2$, $3-2 = 1 < 2$ y $1-1 = 0 < 1$.

Por lo tanto para el caso $i \leq 1$ hemos probado que podemos generar todos los naturales entre 4 y 0 y que además podemos elegir una combinación de potencias diferentes de 3 para lograrlo. Por lo tanto cuando el algoritmo se encuentre con un $equilibrioActual \leq 4$, habrá terminado en 1 o 2 pasos.

Veamos que se cumple para cualquier $i > j > 1$:

Sea $3^i > equilibrioActual > 3^{i-1}$ (usando (2.5)) supongamos que se cumple la primera valuación:

$$1) \ 3^i - equilibrioActual = nuevoEquilibrio_i < equilibrioActual$$

En este caso, nuestro algoritmo usará 3^i como una de las pesas para equilibrar.

Notemos que $nuevoEquilibrio_i$ puede ser mayor que 3^{i-1} : $27-16 = 11 > 9$ o menor: $27-25 = 2 < 9$

En cualquier caso, veamos que como $i-1 > 0$; por la propiedad (2.5) $\exists j \leq i-1$ t.q $|nuevoEquilibrio_i - 3^j| < nuevoEquilibrio_i$:

a) Si $3^j < nuevoEquilibrio_i$ es obvio que $nuevoEquilibrio_i - 3^j < nuevoEquilibrio_i$

b) Si $3^j > nuevoEquilibrio_i$ como $j > 0$ entonces o bien:

$3^j - nuevoEquilibrio_i < nuevoEquilibrio_i$ (siempre evaluamos primero la potencia mayor) o bien lo es

$$nuevoEquilibrio_i - 3^{j-1} < nuevoEquilibrio_i$$

2) La valuación 1) no se cumple, por lo tanto se cumple:

$$equilibrioActual - 3^{i-1} = nuevoEquilibrio_i < equilibrioActual.$$

Aquí debemos notar que para continuar debe cumplirse que

$$\begin{aligned} & nuevoEquilibrio_i < 3^{i-1} \text{ es decir} \\ & equilibrioActual - 3^{i-1} < 3^{i-1} \text{ o sea} \\ & equilibrioActual < 2 * 3^{i-1} = \frac{3^i + 3^{i-1}}{2} \end{aligned}$$

Es decir que *equilibrioActual* sea menor al promedio del intervalo que lo contiene. Lo cual es cierto dado que como 1) no se cumplió, *equilibrioActual* tiene un valor menor a la mitad de ese intervalo (incluso menor a la mitad de 3^i). Por lo tanto la potencia que tendremos que elegir es si o si menor a 3^{i-1} y debe cumplir que la diferencia con *nuevoEquilibrio_i* es menor que *nuevoEquilibrio_i*.

Y nuevamente por la propiedad (2.5) y por ser $i - 1 > 0$ existe un intervalo con $j > 0$ tal que ($3^j \geq \text{nuevoEquilibrio}_i \geq 3^{j-1}$). Si $j < i - 1$ entonces puedo empezar probando con la potencia mayor y luego con la menor.

De manera que $|\text{nuevoEquilibrio}_i - 3^j| < \text{nuevoEquilibrio}_i$:

Supongamos en el peor caso que $j = i - 1$ luego ($3^{i-1} \geq \text{nuevoEquilibrio}_i \geq 3^{i-2}$)

En este caso veamos que se cumplirá que

$$\begin{aligned} 3^{i-1} - \text{nuevoEquilibrio}_i &> \text{nuevoEquilibrio}_i \\ \text{Es decir } 3^{i-1} &> 2 * \text{nuevoEquilibrio}_i = \\ 3^{i-1} &> 2 * (\text{equilibrioActual} - 3^{i-1}) = \\ 3 * 3^{i-1} &> 2 * \text{equilibrioActual} = \\ 3^i &> 2 * \text{equilibrioActual} = \\ \frac{3^i}{2} &> \text{equilibrioActual} \end{aligned}$$

Lo cual es cierto dado que no se había cumplido 1), lo que significaba que

$$\begin{aligned} 3^i - \text{equilibrioActual} &> \text{equilibrioActual} = \\ \frac{3^i}{2} &> \text{equilibrioActual} \end{aligned}$$

Por lo tanto, siempre tendremos una potencia de 3 para elegir que haga que la diferencia sea menor.

Como *equilibrioActual* valdrá 0 en algún momento bien por ser potencia de 3 o bien por ser 4 o 2. Hemos conseguido mediante sumas o restas de potencias de 3 disminuir el valor inicial de la llave a 0, es decir que encontramos un equilibrio mediante las pesas.

2.6 Experimentos y conclusiones

2.6.1 Test

Luego de realizar la implementación de nuestro algoritmo, desarrollamos tests, para corroborar que nuestro algoritmo era el indicado.

A continuación enunciaremos varios de nuestros tests:

El valor de entrada P es de la forma 3^i para un $i \in [0, N]$

Este caso se cumple cuando se recibe un P el cual al realizar nuestro primer ciclo que chequea cual es la potencia igual o mayor, termina siendo igual y de esta forma solo se itera una unica vez el segundo y tercer ciclo.

El valor de entrada P es de la forma $3^i + R$ para $(i, R) \in [0, N]$

Este caso se cumple cuando se recibe un P el cual al realizar nuestro primer ciclo que chequea cual es la potencia igual o mayor, termina siendo mayor y de esta forma se itera mas de una vez el segundo y tercer ciclo.

2.6.2 Performance De Algoritmo y Gráfico

Acorde a lo solicitado, mostraremos los mejores y peores casos para nuestro algoritmo, y además, daremos el tiempo estimado según la complejidad del algoritmo calculada anteriormente.

Luego de chequear varias instancias, pudimos llegar a la conclusión que uno de los tipos de casos que resulta más beneficioso para nuestro algoritmo es en el cual ambas ramas de la mediana se encuentran ya ordenadas

Para llegar a dicha conclusión trabajamos con un total de 100 instancias y un n entre 1 y 1010000 obtuvimos que nuestro algoritmo finaliza lo solicitado demorando 333 milisegundos.

Para una mayor observación desarrollamos el siguiente gráfico con las instancias:

Y dividiendo por la complejidad de nuestro algoritmo llegamos a:

Para realizar esta experimentación nos pareció prudente, realizar un promedio con el mismo input (n entre 1 y 1001000) de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más relevantes.

Se puede observar, como luego de realizar la división por la complejidad cuando el n aumenta el valor tiende a 0.

A continuación mostraremos una tabla con los 10 datos de medición más relevantes y mostraremos un promedio de la totalidad de las instancias probadas.

Tamaño(n)	Tiempo(t)	$t/n.\log(n)$
730000	249000	0,058
770000	261000	0,058
810000	273000	0,057
850000	285000	0,057
890000	297000	0,056
930000	309000	0,056
970000	321000	0,055
1010000	333000	0,055
Promedio		0,104

Promedio final de todas las instancias: 0,104

Verificando el peor caso, llegamos a la conclusión que el tipo de caso en el que resulta menos beneficioso trabajar con nuestro algoritmo será cuando ambas ramas se encuentran desordenadas.

Realizando experimentos con un total de 100 instancias con un n variando desde 1 hasta 1010000 obtuvimos que nuestro algoritmo, resuelve lo mencionado en 385 milisegundos, a continuación mostraremos un gráfico que ejemplifica lo enunciado.

Y dividiendo por la complejidad propuesta llegamos a:

Para realizar esta experimentación nos pareció acorde, realizar un promedio con el mismo input de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más relevantes.

Se puede observar que a pesar de tardar varios milisegundos este tipo de caso, al dividir por vuestra complejidad es propenso a tender a 0 quedando comparativamente por encima del mejor caso.

A continuación mostraremos una tabla de valores de las últimas 10 instancias y mostraremos el promedio total conseguido .

Tamaño(n)	Tiempo(t)	$t/n.\log(n)$
610000	268000	0,076
650000	280000	0,074
690000	292000	0,073
730000	304000	0,071
770000	316000	0,070
810000	328000	0,069
850000	340000	0,068
890000	352000	0,067
930000	364000	0,066
970000	376000	0,065
1010000	388000	0,064
Promedio		0,195

Promedio total conseguido: 0,195

Se puede observar como el peor caso presenta un promedio mayor que el mejor caso, concluyendo lo que enunciamos inicialmente.

3 Ejercicio 3

3.1 Descripción de problema

Luego de haber equilibrado la balanza, Indiana y compañía llegan a una habitación la cual se encuentra repleta de objetos valiosos.

Indiana y el grupo poseen varias mochilas las cuales soportan un peso máximo.

Nuestro objetivo en este ejercicio será ayudarlos a guardar la mayor cantidad posible de objetos valiosos en las mochilas teniendo en cuenta el valor de cada objeto y su peso.

3.2 Explicación de resolución del problema

La solución planteada utiliza la técnica algorítmica de *backtracking*. La idea es recorrer todas las configuraciones posibles manteniendo la mejor solución encontrada hasta el momento.

Inicialmente, ordenaremos en base al peso y el valor de todos los objetos.

Luego de realizar esto iremos agregando en las mochilas los objetos de mayor valor teniendo en cuenta el peso de los mismos con la mochila, en caso de que al agregar un objeto la suma de los pesos de los objetos que se encuentran en la mochila diera igual o mayor al peso máximo de la mochila, se quitará el objeto ultimo y se probará con otro objeto de menor peso.

Así realizaremos todas las posibles permutaciones de objetos en la mochila.

Una vez que obtuvimos todas las permutaciones posibles nos quedaremos con la máxima, de esta manera tendríamos en las mochilas una cantidad óptima de objetos con el mayor valor posible y un peso acorde a lo soportado por las mochilas.

3.3 Algoritmos

A continuación se detalla el pseudo-código de la parte principal del algoritmo incluyendo las podas:

Algoritmo 4 Calculate

global: remainingFriendships, girls, currentSum, currentMin, bestRound

```
1: function CALCULATE(in currentIdx: Integer)
2:   if remainingFriendships = 0 then                                     //O(1)
3:     sittingGirls ← copy(girls)                                         //O(n)
4:     subList ← sittingGirls[currentIdx, size(girls)]                     //O(1)
5:     sort(subList)                                                       //O((k * log(k)) where k = size(girls) - currentIdx
6:
7:     if currentSum < currentMin then                                     //O(1)
8:       bestRound ← sittingGirls                                         //O(1)
9:     else
10:      bestRound ← firstLexicographically(bestRound, sittingGirls)      //O(n)
11:    end if
12:    currentMin ← currentSum                                             //O(1)
13:  else
14:    for swapIdx from currentIdx to size(girls) do
15:      swap(girls, currentIdx, swapIdx)                                   //O(1)
16:      partialDistance ← getPartialDistance(currentIdx)                 //O(n log(n))
17:      currentSum ← currentSum + partialDistance                         //O(1)
18:      if currentSum ≤ currentMin then                                   //O(1)
19:        calculate(currentIdx + 1)
20:      end if
21:      swap(girls, currentIdx, swapIdx)                                   //O(1)
22:      currentSum ← currentSum - partialDistance                         //O(1)
23:    end for
24:  end if
25: end function
```

Complejidad: $O(n!.n^2.log(n))$

Algoritmo 5 getMaxDistance

```
1: function GETPARTIALDISTANCE(in currentIdx: Integer, out res: Integer)
2:   sum ← 0                                                             //O(1)
3:   count ← 0                                                           //O(1)
4:   for girlIdx from 0 to currentIdx do                                //O(n)
5:     friendship ← Friendship(girls[girlIdx], girls[currentIdx])        //O(1)
6:     if contains(friendships, friendship) then                       //O(log(n))
7:       idxDiff ← rightIdx - leftIdx                                     //O(1)
8:       distance ← min(idxDiff, size(girls) - idxDiff)                 //O(1)
9:       sum ← sum + distance                                           //O(1)
10:      count ← count + 1                                              //O(1)
11:    end if
12:  end for
13: end function
```

Complejidad: $O(nlog(n))$

3.4 Análisis de complejidades

RESOLUCION DEL PUNTO Y ANALISIS