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

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 cruze 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

Algoritmo 1 CRUZANDO EL PUENTE

```
1: function EJ1(in : Integer, in : List<Integer>)→ out res: Integer
2:     creo bool exitoBackPar con valor verdadero                          //O(1)
3:     creo bool exitoBackLampara con valor verdadero                      //O(1)
4:     while exitoBackLampara ∨ exitoBackPar do                            //O(N??)
5:         if tienenLampara then                                           //O(1)
6:             par ← parPosible()                                          //O(???)
7:             if par > -1 then                                            //O(1)
8:                 funcion enviarPar(par)                                  //O(??)
9:             else
10:                exitoBackLampara ← backtrackRetorno(farolero)           //O(??)
11:            end if
12:        else
13:            farolero ← retornoPosible                                   //O(N?)
14:            if farolero > -1 then                                       //O(1)
15:                retornarLampara(farolero)                               //O(??)
16:            else
17:                exitoBackPar ← backtrackPar(par)                        //O(??)
18:            end if
19:        end if
20:        if pasaronTodos() then                                          //O(1)
21:            guardarTiempo()                                             //O(??)
22:        end if
23:    end while
24: end function
```

Complejidad: O(??)

Algoritmo 2 CRUZANDO EL PUENTE

```
1: function ALGORITMO PRINCIPAL(in : Integer,in : Integer, in : List<Integer>)→ out res : Integer
2:     creo bool exitoBackPar con valor verdadero                                  //O(1)
3:     creo bool exitoBackLampara con valor verdadero                              //O(1)
4:     while exitoBackLampara ∨ exitoBackPar do                                    //O(N??)
5:         if pasaronTodos(escenario) then                                         //O(1)
6:             if escenario.tiempo < minimo ∨ minimo == -1 then                    //O(1)
7:                 minimo ← escenario.tiempo                                       //O(1)
8:             end if
9:             sol++                                                               //O(1)
10:        end if
11:        exitoBackPar ← verdadero                                                //O(1)
12:        exitoBackLampara ← verdadero                                            //O(1)
13:        if escenario.tienenLampara then                                         //O(1)
14:            creo entero par con escenario.parPosible()                          //O(??)
15:            if par > -1 ∧ (minimo == -1 ∨ escenario.tiempo < minimo) then       //O(1)
16:                escenario.printPar(par)                                         //O(??)
17:                escenario.enviarPar(par)                                        //O(??)
18:            else
19:                exitoBackLampara ← escenario.backtrackFarolero()                //O(??)
20:            end if
21:        else
22:            creo entero farolero con faroleroPosible(escenario)                 //O(N?)
23:            if farolero>-1 ∧ (minimo == -1 ∨ escenario.tiempo < minimo then     //O(1)
24:                escenario.printPersona(farolero)                                //O(??)
25:                escenario.enviarFarolero(farolero)                              //O(??)
26:            else
27:                exitoBackPar ← escenario.backtrackPar()                         //O(??)
28:            end if
29:        end if
30:    end while
31: end function
```

Complejidad: O(??)

//ESTOS DOS PUEDEN NO IR //ESTOS DOS PUEDEN NO IR //ESTOS DOS PUEDEN NO IR

Algoritmo 3 CRUZANDO EL PUENTE

```
1: function parPosible→ out res: Integer
2:     creo entero parEvaluar con parActual_x_paso[paso] + 1                    //O(1)
3:     creo entero tot_pares con arq_totales*can_totales                        //O(1)
4:     while ≠parValido(parEvaluar) ∧ parEvaluar ≤ tot_pares do                 //O(N??)
5:         parEvaluar++                                                         //O(1)
6:     end while
7:     if parEvaluar ≤ tot_pares then                                           //O(1)
8:         devolver parEvaluar                                                  //O(1)
9:     else
10:        devolver -1                                                          //O(1)
11:    end if
12: end function
```

Complejidad: O(N??)

Algoritmo 4 CRUZANDO EL PUENTE

```
1: function parValidoin par: Bool→ out res: Integer
2:     creo entero aux_arq_destino con aux_arq_destino                          //O(1)
3:     creo entero aux_can_destino con can_destino                              //O(1)
4:     if par > arq_totales ∗ can_totales then                                  //O(1)
5:         devolver falso                                                       //O(1)
6:     end if
7:     creo a con primero(par)                                                  //O(1)
8:     creo b con segundo(par)                                                  //O(1)
9:     if ≠((esCanibal(a) ∨ esArquitecto(a)) ∧ (esCanibal(b) ∨ esArquitecto(b))) then  //O(1)
10:        devolver falso                                                       //O(1)
11:    end if
12:    aux_can_destino ← esCanibal(a) ∧ canibal_origen[a]                       //O(1)
13:    aux_arq_destino ← esArquitecto(a) ∧ arquitecto_origen[a]                 //O(1)
14:    aux_can_destino ← esCanibal(b) ∧ canibal_origen[b]                       //O(1)
15:    aux_arq_destino ← esArquitecto(b) ∧ arquitecto_origen[b]                 //O(1)
16:    creo arq_origen con arq_totales - aux_arq_destino                        //O(1)
17:    creo can_origen con can_totales - aux_can_destino                        //O(1)
18:    if arq_origen ≥ can_origen ∧ arq_destino ≥ can_destino then              //O(1)
19:        devolver verdadero                                                   //O(1)
20:    else
21:        devolver falso                                                       //O(1)
22:    end if
23: end function
```

Complejidad: O(???)

1.4 Análisis de complejidades

ACA IRIA LOS COMENTARIOS DE COMPLEJIDAD Y LA DEMOSTRACION

1.5 Demostración de correctitud

1.6 Experimientos y conclusiones

1.6.1 Test

Para verificar el correcto funcionamiento de nuestro algoritmo , elaboramos disversos tests, los cuales serán enunciados a continuación.

Todos los arqueologos y canibales presentan la misma velocidad

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

Con un:

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

1.6.2 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 referiendonos 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 habiamos 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 esta vacio.

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

Nuestra objetivo en este punto consistira 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:

Una aclaración previa que será de utilidad, como se dio como precondicion que la llave puede llegar a tomar el valor hasta 10^{15} trabajaremos con variables del tipo long long las cuales nos permitiran llegar hasta dicho valor.

Como las pesas, presentan un peso en potencia de 3, creamos una variable denominada *sumaParcial* como la palabra lo indica iremos sumando las potencias desde 3^0 hasta un 3^i, donde dicha suma sea igual al valor de entrada denominado P o en su defecto el inmediato mayor al mismo.

Luego de tener dicha *sumaParcial* guardada crearemos un array que nombramos *sumasParciales* de tamaño $i+1$ el cual iniciaremos vacio. Una vez creado el mismo, llenaremos el array con cada una de las sumas parciales desde el valor que finalizo i hasta 0 de la forma que nos quede *sumasParciales[i] = sumaParcial* donde *sumaParcial* será *sumaParcial = sumaParcial*-3^{i-1}.

Finalizado esto, realizaremos una busqueda binaria para llevar el valor de p a 0. Para un trabajo más sencillo guardamos el valor inicial de P en la variable *equilibrioActual* a la cual le iremos restando y/o sumando el valor de nuestras pesas.

Dicha busqueda binaria la realizaremos en un ciclo que irẹsde el valor en módulo de equilibrioActual e iteraremos el mismo hasta que sea 0. Luego, como en toda busqueda binaria, trabajaremos con nuestro array *sumasParciales* chequeando si en la mitad del array nuestra sumaParcial es mayor o igual al valor en módulo de *equilibrioActual*. En caso de que fuese verdadero, chequeamos si el valor de *equilibrioActual* es mayor o menor a 0. Si es menor a 0, sumaremos nuestra pesa correspondiente al índice en el que estamos de nuestro array *sumasParciales*, al valor de *equilibrioActual* y guardaremos nuestra pesa en el *arrayD* que simboliza al plato derecho de la balanza. Si es mayor

a 0, en vez de sumarla la restamos y la guardamos en el array $arrayI$ que simboliza el otro plato. Siguiendo el razonamiento de la busqueda binaria, volveremos a partir nuestro array en dos y haremos el mismo chequeo.

En caso de que el valor en módulo de $equilibrioActual$ sea mayor que la mitad de $sumasParciales$ nos quedaremos con la mitad más grande del arreglo e iteraremos nuevamente.

Una vez que llegamos a 0 y por consiguiente salimos de dicho ciclo, tendremos nuestras pesas ordenadas de mayor a menor en $arrayD$ y en $arrayI$, bastara con invertir los arreglos para que queden de menor a mayor y devolver los mismos, finalizando así nuestro algoritmo.

2.3 Algoritmos

Algoritmo 5 BALANZA

```
1: function ALGORITMO(in LongLong : P)→ out S : Long Long out T : Long Long out arrayI :
   List<Long Long> out arrayD : List<Long Long>
2:     creo variable long long equilibrioActual = P                                  //O(1)
3:     creo variable long long i = 0                                                 //O(1)
4:     creo variable long long sumaParcial = 0                                       //O(1)
5:     while sumaParcial < P do                                                      //O(√P)
6:         sumaParcial ← sumaParcial + 3^i                                           //O(1)
7:         i++                                                                       //O(1)
8:     end while
9:     creo long long size = i+1                                                     //O(1)
10:    creo long long sumasParciales[size]                                           //O(√P)
11:    while i ≥ 0 do                                                                //O(√P)
12:        sumasParciales[i] ← sumaParcial                                           //O(1)
13:        sumaParcial ← sumaParcial - 3^(i−1)                                       //O(1)
14:        i–                                                                        //O(1)
15:    end while
16:    creo long long middle = size/2                                                //O(1)
17:    while |equilibrioActual| > 0 do                                               //O(lg(√P))
18:        if sumasParciales[middle] ≥ |equilibrioActual|
19:            ∧ sumasParciales[middle-1] < |equilibrioActual| then                  //O(1)
20:            creo long long potencia = sumasParciales[middle]-sumasParciales[middle-1]  //O(1)
21:            if equilibrioActual < 0 then                                          //O(1)
22:                equilibrioActual ← potencia + equilibrioActual                    //O(1)
23:                arrayD ∪ potencia                                                 //O(1)
24:            else
25:                equilibrioActual ← equilibrioActual -potencia                     //O(1)
26:                arrayI ∪ potencia                                                 //O(1)
27:            end if
28:            size ← middle                                                         //O(1)
29:            middle ← middle/2                                                     //O(1)
30:        end if
31:        if sumasParciales[middle] < |equilibrioActual| then                       //O(1)
32:            middle ← middle + size/2                                              //O(1)
33:        end if
34:        if sumasParciales[middle-1] ≥ |equilibrioActual| then                     //O(1)
35:            size ← middle                                                         //O(1)
36:            middle ← middle + size/2                                              //O(1)
37:        end if
38:    end while
39:    devolver(armadoBalanza)                                                       //O(√P)
40: end function
```

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

Algoritmo 6 armadoBalanza

```
1: function ARMADOBALANZA( : )→ out S : Integer out T : Integer out arrayI : List<Integer>
   out arrayD : List<Integer>
2:     invertir(arrayD)                                   //O(√P)
3:     invertir(arrayI)                                   //O(√P)
4:     devolver arrayD.tamaño                             //O(1)
5:     devolver arrayI.tamaño                             //O(1)
6:     devolver(arrayD)                                   //O(√P)
7:     devolver(arrayI)                                   //O(√P)
8: end function
```

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

2.4 Análisis de complejidades

Nuestro algoritmo como mencionamos anteriormente presenta 3 ciclos predominantes de los cuales uno corresponde a la busqueda binaria.
El primero de ellos consta en recorrer desde 3^0 hasta 3^i donde la suma de estos sea igual a P o en su defecto el inmediato mayor. Por lo tanto, como la suma se realiza en O(1), mostraremos que recorrer hasta un i donde la suma de dichos valores 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

$$\sum_{x=0}^{i} 3^i \geq P$$

se encuentra en el orden de $O(\sqrt{P})$.

Luego, creamos un long long *size* inicializado en i+1 y un array *sumasParciales* de tamaño size inicializado vacio, por lo tanto, la creacion de la variable size y el array vacio insumirán O(1) y $O(\sqrt{P})$.

Siguiendo el desarrollo del algoritmo, pasamos a nuestro segundo ciclo, en el cual llenaremos el array *sumasParciales*, como vimos anteriormente iterar desde el valor i hasta 0 es $O(\sqrt{P})$, y dentro de dicho ciclo lo único que hacemos es ir guardando en la posición i-esima del array el valor de sumaParcial - 3^{i-1} y a sumaParcial le guardamos el valor de sumaParcial - 3^{i-1}. Como estas dos operaciones se realizan en O(1), nuestro segundo ciclo terminará insumiendo $O(\sqrt{P})$.

Luego, nuestro tercer y último ciclo, corresponde a la búsqueda binaria, la cual se realizá en $O(\lg(\sqrt{P}))$ como en toda busqueda binaria, trabajaremos con nuestro array *sumasParciales* chequeando si en la mitad del array nuestra sumaParcial es mayor o igual al valor en módulo de *equilibrioActual*. En caso de que fuese verdadero, chequeamos si el valor de *equilibrioActual* es mayor o menor a 0. Si es menor a 0, sumaremos nuestra pesa correspondiente al índice en el que estamos de nuestro array *sumasParciales*, al valor de *equilibrioActual* y guardaremos nuestra pesa en el *arrayD* que simboliza al plato derecho de la balanza. Si es mayor a 0, en vez de sumarla la restamos y la guardamos en el array *arrayI* que simboliza el otro plato. Siguiendo el razonamiento de la busqueda binaria, volveremos a partir nuestro array en dos y haremos el mismo chequeo.

En caso de que el valor en módulo de *equilibrioActual* sea mayor que la mitad de *sumasParciales* nos quedaremos con la mitad más grande del arreglo e iteraremos nuevamente.
Una vez llegado a 0 el valor de *equilibrioActual* saldremos del ciclo. Como describimos dentro del ciclo realizaremos sumas, restas y chequeos los cuales se realizaran todos en O(1), por lo tanto. Nuestro tercer ciclo insumirá O(lg($\sqrt{P}$)).

Fuera de este último ciclo, tendremos nuestras pesas ordenadas de mayor a menor en $arrayD$ y en $arrayI$, bastará con invertir los arreglos para que queden de menor a mayor y devolver los mismos, finalizando así nuestro algoritmo. Dicho invertir costará O(#elementosArrayD) y O(#elementosArrayI), que como demostramos anteriormente en el caso de que todas las pesas fueran a parar a un único plato y naturalmente a un único array O(#elementos) $\leq$ O($\sqrt{P}$).

Por lo tanto, nuestro algoritmo realizará en su defecto 3 ciclos (como vimos, se puede dar el caso de invertir el array y que estén todas las pesas en un único array) O($\sqrt{P}$) y el ciclo de la busqueda binaria O(lg($\sqrt{P}$)), nos queda que la complejidad total de nuestro algoritmo es O($\sqrt{P}$).

Complejidad total: O($\sqrt{P}$) [O(1) + O(1)] + O(1) + O($\sqrt{P}$) + O($\sqrt{P}$) [O(1) + O(1)] + O(lg($\sqrt{P}$))[O(1) + O(1) + O(1) + O(1) + O(1) + O(1)] = **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.

Este algoritmo utiliza una propiedad particular que es la de poder generar todos los números entre 0 y $\sum_{i=0}^{n}(3^i)$ con potencias de 3 diferentes (El 0 se incluye por definición). Pero para que esto sea válido debemos demostrarlo.

Veamos para empezar que podemos generar todos los números entre $[0, 1]$ que son 0 y 1.

Este es un caso bastante trivial, por lo tanto veamos que sucede en el intervalo

$$[0, \sum_{i=0}^{1}(3^i)] = [0, 4] \tag{1}$$

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

Parece pues que para el caso base, es decir el primer intervalo, podemos generarlos todos con potencias de 3 diferentes.

Asumamos pues que esto vale para $i = n \in \mathbb{N}$ y demostremos que vale para $n + 1$

Es decir, puedo generar todos los números en el intervalo

$$[0, \sum_{i=0}^{n}(3^i)] \tag{2}$$

Veamos que podemos lograr lo mismo para

$$[0, \sum_{i=0}^{n+1}(3^i)] \tag{3}$$

Pues veamos que

$$[0, \sum_{i=0}^{n+1}(3^i)] = [0, \sum_{i=0}^{n}(3^i)] + 3^{n+1} \tag{4}$$

Con lo cual ya puede verse que los numeros entre 0 y $\sum_{i=0}^{n}(3^i)$ podemos generarlos por hipótesis inductiva (2.5).

Luego notemos que

$$3^{n+1} = 3 * 3^n. \tag{5}$$

Como $3^n < \sum_{i=0}^{n}(3^i)$, 3^n está dentro del intervalo de la hipótesis inductiva (2.5) así que tambien podemos formarlo con potencias de 3, y en particular cualquier número menor a 3^n usando la misma hipótesis.

Luego podemos generar cualquier $x \in \mathbb{N}$, $x \leq \sum_{i=0}^{n+1}(3^i)$.

Por lo tanto queda probado que la hipótesis inductiva vale $\forall n \in \mathbb{N}$

Además notemos que el número más cercano a x será la *maxima potencia de* 3 *en* $[0, \sum_{i=0}^{n+1}(3^i)]$ es decir 3^{n+1}.
Pues este es el intervalo que genera a x y sabemos que $\sum_{i=0}^{n}(3^i) < 3^{n+1}$ y que $\sum_{i=0}^{n}(3^i) < x$.

Y como al restar x por 3^{n+1} obtenemos un número más pequeño que $\sum_{i=0}^{n}(3^i)$ en módulo, pues (el caso negativo es simétrico)

$$x < \sum_{i=0}^{n+1}(3^i) = x < \sum_{i=0}^{n}(3^i) + 3^{n+1} = x - 3^{n+1} < \sum_{i=0}^{n}(3^i) \tag{6}$$

Entonces nunca se repetiran potencias de 3 en el proceso.
Con esto se concluye que se pueden generar todos los números mediante potencias de 3 únicas.

2.6 Experimientos 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←[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 única vez el segundo y tercer ciclo.

El valor de entrada P es de la forma

$$\sum_{i=1}^{n} 3_i = P$$

Veremos mas adelante que este caso será el peor a resolver ya que se iterará la totalidad completa de elementos de nuestros arrays.

El valor de entrada P es de la forma $3^i + R$ para (i, R) $\leftarrow$[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.

El valor de entrada P es impar

Este caso se cumple cuando se recibe un P el cual el mismo es de la forma $PMOD2 = 1$.

El valor de entrada P es par

Este caso se cumple cuando se recibe un P el cual el mismo es de la forma $PMOD2 = 0$.

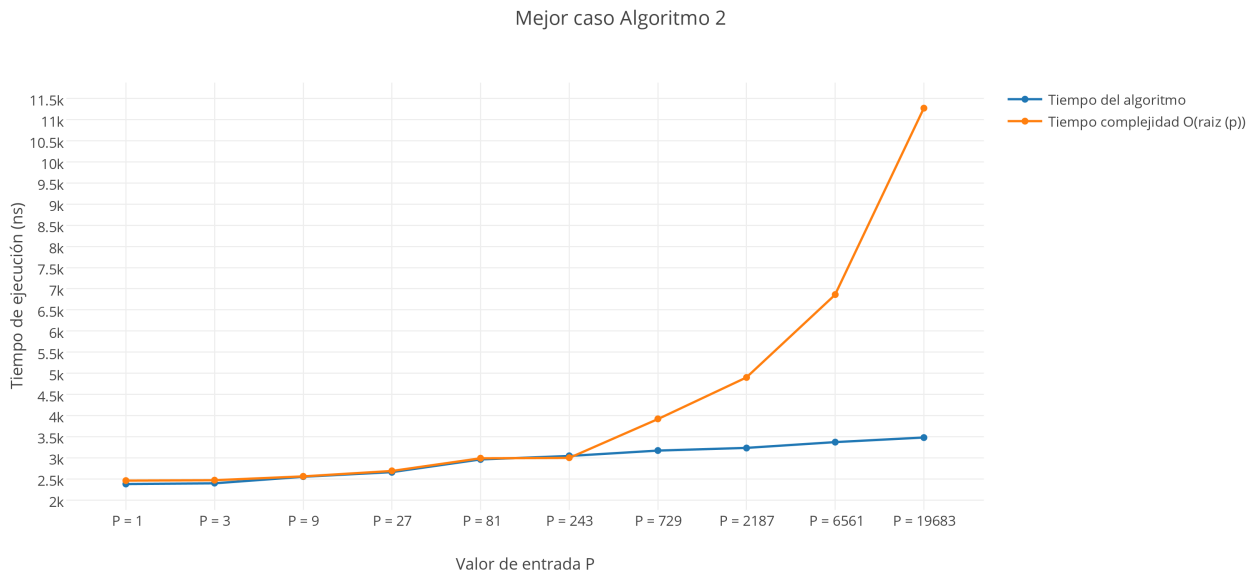
2.6.2 Performance De Algoritmo y Gráfico

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.

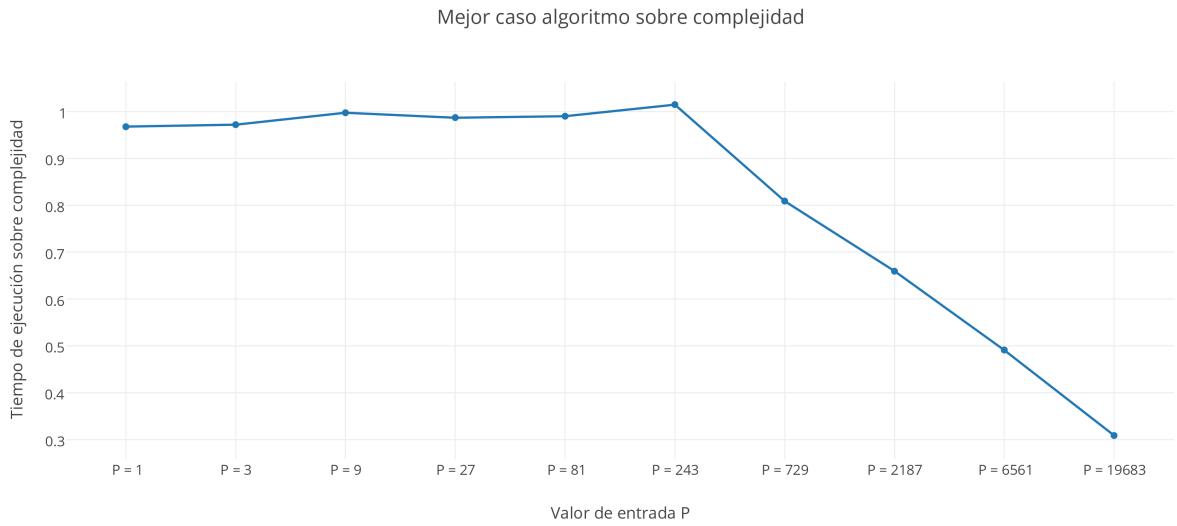
Luego de chequear varios instancias, pudimos llegar a la conclusión que uno de los tipos de casos que resulta más beneficioso para nuestro algoritmo es en el cual se recibe P con un valor exactamente igual a 3^i con $0 \leq i \leq n$

Para llegar a dicha conclusión trabajamos con 30 instancias ya que 3^{30} es el ultimo valor dentro del valor que puede tomar P.

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



Y dividiendo por la complejidad de nuestro algoritmo llegamos a:



Como se puede ver, en el primer gráfico cuando el valor de entrada P crece el tiempo de la funcion de la complejidad tiende a crecer muy rapido por lo cual mostraremos estas instancias en los graficos y mas adelante mostraremos una tabla con los valores restantes.

Para realizar esta experimentación nos parecio prudente, 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, 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 20 datos de medición mas relevantes y mostraremos un promedio de la totalidad de las instancias probadas.

Tamaño(n)	Tiempo(t)	$\sqrt{P}$)	$t/\sqrt{P}$)
3^{10}	3730,82	18626	0,2003017288
3^{11}	4428,36	31370	0,1411654447
3^{12}	4685,66	86265	0,0543170463
3^{13}	4999,42	124006	0,0403159525
3^{14}	5460,14	188705	0,0289347924
3^{15}	5756,1	356824	0,0161314822
3^{16}	5891,5	265657	0,022177093
3^{17}	6026,9	400446	0,0150504687
3^{18}	6162,3	1617468	0,0038098435
3^{19}	6297,7	2542364	0,002477104
3^{20}	6433,1	2608534	0,0024661745
3^{21}	6568,5	3956914	0,0016600058
3^{22}	6703,9	7124699	0,000940938
3^{23}	6839,3	11467843	0,0005963894
3^{24}	6974,7	19803192	0,0003522008
3^{25}	7110,1	35212754	0,0002019183
3^{26}	7245,5	59285080	0,0001222146
3^{27}	7380,9	103014535	7,1649112428649E-005
3^{28}	7516,3	178188535	4,21817262261009E-005
3^{29}	7651,7	308181445	2,48285551390026E-005
3^{30}	7787,1	438174355	1,77716927317666E-005
Promedio			0,026

Promedio total conseguido: 0,026

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 el valor de entrada P sea de la forma

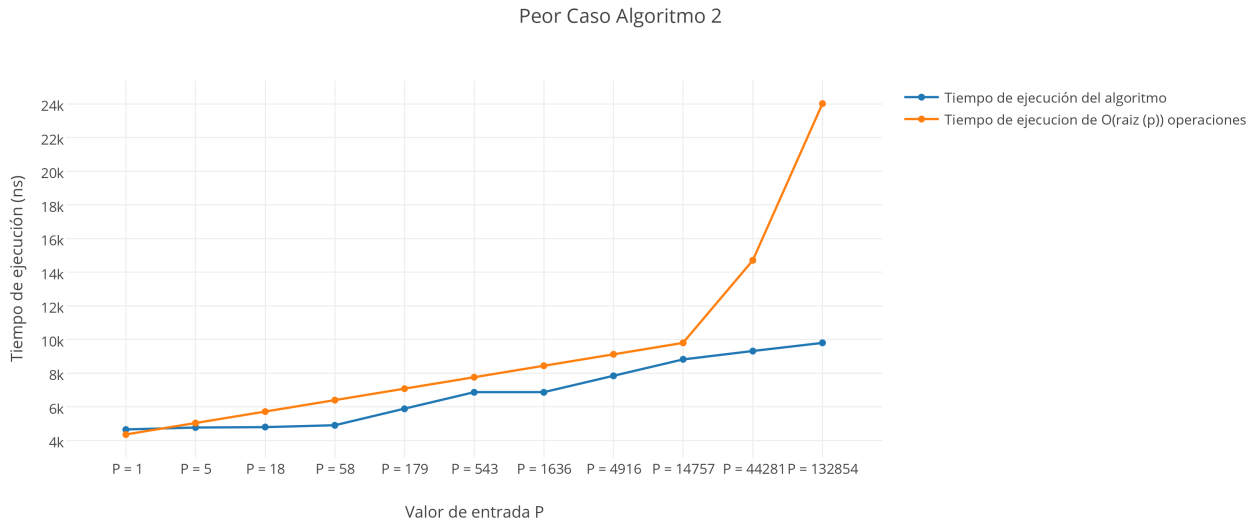
$$\sum_{i=1}^{n} 3_i = P$$

.

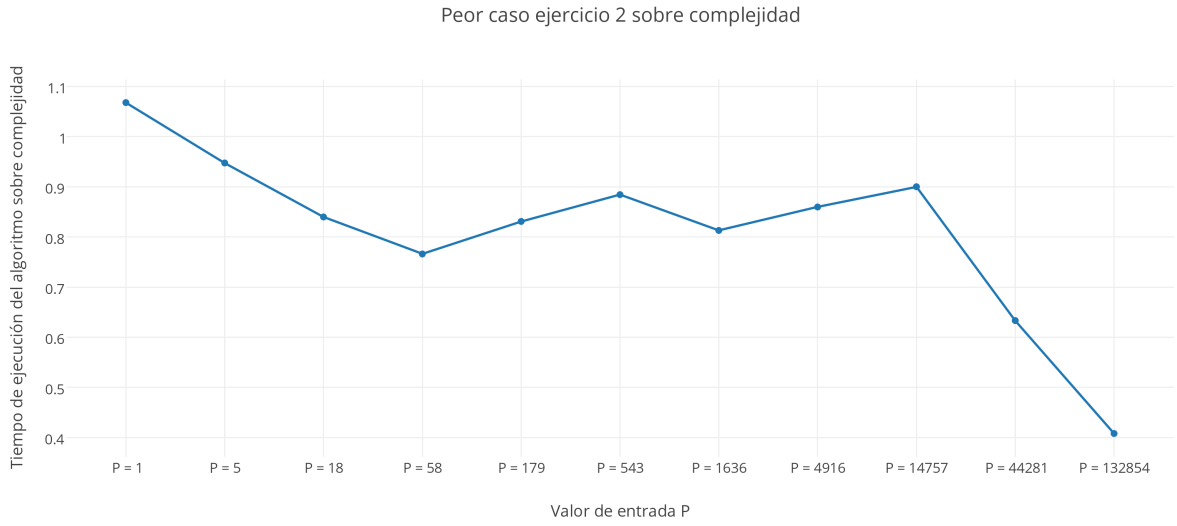
Realizando experimentos con un total de 20 instancias donde

$$\sum_{i=1}^{20} 3_i = 5230176601$$

, desarrollamos una tabla comparativa con los valores mas relevantes y ademas dos graficos los cuales mostraremos a continuación:



Dividiendo por la complejidad propuesta llegamos a:



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.

Se puede observar que a pesar de tardar varios nanosegundos 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 lass ultimas 20 instancias y mostraremos el promedio total conseguido .

Tamaño(n)	Tiempo(t)	$\sqrt{P}$)	$t/\sqrt{P}$)
1	4650	4355	1.06773823191734
5	4771	5036	0.947378872120731
18	4801	5717	0.839776106349484
58	4901	6398	0.766020631447327
179	5882	7079	0.830908320384235
543	6862	7760	0.884278350515464
1636	6862	8441	0.81293685582277
4916	7842	9122	0.859679894759921
14757	8823	9803	0.90003060287667
44281	9312	14704	0.633297062023939
132854	9803	24017	0.408169213473789
398574	18625	39702	0.469119943579669
1195735	18625	67150	0.277364110201042
3587219	19115	57837	0.330497778238844
10761672	26468	132339	0.200001511270298
32285032	26468	203410	0.130121429624896
96855113	31860	307319	0.103670778572103
290565357	32839	548960	0.0598203876420869
581130733	39211	910195	0.0430797796076665
1743392200	41535	1567476	0.0264980133667118
5230176601	53915	2659024	0.0202762366943661
Promedio			0,530

Promedio total conseguido: 0,530

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

Luego de dichos experimentos y casos probados, se puede concluir que a pesar de utilizar todas las pesas como en el peor caso nos mantenemos dentro de la complejidad propuesta como habíamos mostrado en nuestro desarrollo de la complejidad.

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 ṕtima 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 7 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 8 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