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 resolver este ejercicio, la solución que planteamos utiliza la técnica algorítmica de *backtracking*. La idea es recorrer todas las configuraciones posibles manteniendo la mejor solución encontrada hasta el momento.

Para poder resolver este punto, nos fue de gran utilidad la creacion de una clase la cual denominamos *Escenario* la cual, como la palabra lo indica nos dará entre otras cosas la cantidad de canibales y arqueologos que hay en cada lado, como tambien la cantidad de pares y el parActual. Dicha clase exportará varias funciones como si existe un $faroleroPosible$ y de existir que farolero se enviaría, que par se envía, y por ultimo la funciones de backtracking que habíamos enunciado.

Una vez nombrado nuestra clase ya podemos centrarnos en la explicación del algoritmo central. Inicialmente, creamos dos booleanos los cuales nos dira si el backtracking realizado tanto para la lampara como para el par es el óptimo Dichas variables fueron denominadas como $exitoBackPar$ y $exitoBackLampara$. Por lo cual, ambos son inicializados con el valor verdadero. Además, creamos 3 enteros sol, $minimo$ e i los cuales inicializamos con 0,-1 y 0 respectivamente. Y fundamentalmente, creamos una variable del tipo escenario denominada $escenario$ con los valores que recibimos de entrada.
Luego, iniciamos la etapa principal de nuestro algoritmo, la cual se desarrolla dentro de un ciclo el cual valdrá cuando los valores de $exitoBackPar$ y $exitoBackLampara$ sean verdaderos. Aqui chequeamos inicialmente si ya pasaron todos las personas (tanto canibales como arquelogos) de haber pasado, chequemos si el tiempo que paso en esta solución es mejor al mínimo que teníamos hasta el momento, de ser así, la contamos como solución óptima hasta el momento.
Siguiendo el algoritmo, chequeamos si el par que esta por pasar posee lámpara, si esto es cierto, se selecciona un par y si es un par en el cual el tiempo que le lleva al mismo es el mínimo, en ese caso enviamos el par, sino realizamos backtracking al farolero utilizando la función $backtrackFarolero$ asignando el resultado de este backtracking a $exitoBackLampara$. En el caso que no se tenga la lámpara aún, se elije un $faroleroPosible$ chequeandose nuevamente si con dicho farolero que se elijio se obtiene el minimo posible en esta ocasión, en caso afirmativo, enviamos el farolero, en caso

negativo hacemos backtracking al par con la funcíon *backtrackPar* asignando el resultado de este backtracking a *exitoBackPar*.

En las funciones *backtrackPar* y *backtrackFarolero* lo que realizamos es retroceder 1 paso hacia atras en principio, tomando el par que viajo ultimo y se lo manda devuelta al sector A. llevando con él la lámpara de vuelta y se resta el tiempo al tiempo total de la rama que se esta evaluando en este momento. En el caso que no haya pasos hacia atras se devolvera falso, sino se realizara el retroceso que mencionamos actualizando las estructuras que implica dicho retroceso y por último devolvemos verdadero.

Luego, una vez finalizado el ciclo, retornamos el mínimo obtenido, el cual será el óptimo total.

1.3 Algoritmos

Algoritmo 1 CRUZANDO EL PUENTE

```
function MAIN(in : Integer, in : List<Integer>)→ out res: Integer
    creo bool exitoBackPar con valor verdadero                          //O(1)
    creo bool exitoBackLampara con valor verdadero                      //O(1)
    while exitoBackLampara ∨ exitoBackPar do                            //O(N??)
        if tienenLampara then                                           //O(1)
            par ← parPosible()                                          //O(???)
            if par > -1 then                                            //O(1)
                funcion enviarPar(par)                                  //O(??)
            else
                exitoBackLampara ← backtrackRetorno(farolero)           //O(??)
            end if
        else
            farolero ← retornoPosible                                   //O(N?)
            if farolero > -1 then                                       //O(1)
                retornarLampara(farolero)                               //O(??)
            else
                exitoBackPar ← backtrackPar(par)                        //O(??)
            end if
        end if
        if pasaronTodos() then                                          //O(1)
            guardarTiempo()                                             //O(??)
        end if
    end while
end function
```

Complejidad: O(??)

Algoritmo 2 CRUZANDO EL PUENTE

```
function Algoritmo Principal(in : Integer,in : Integer, in : List<Integer>)→ out res : Integer
    creo bool exitoBackPar con valor verdadero                                //O(1)
    creo bool exitoBackLampara con valor verdadero                            //O(1)
    while exitoBackLampara ∨ exitoBackPar do                                  //O(N??)
        if pasaronTodos(escenario) then                                       //O(1)
            if escenario.tiempo < minimo ∨ minimo == -1 then                  //O(1)
                minimo ← escenario.tiempo                                     //O(1)
            end if
            sol++                                                             //O(1)
        end if
        exitoBackPar ← verdadero                                              //O(1)
        exitoBackLampara ← verdadero                                          //O(1)
        if escenario.tienenLampara then                                       //O(1)
            creo entero par con escenario.parPosible()                        //O(??)
            if par > -1 ∧ (minimo == -1 ∨ escenario.tiempo < minimo) then     //O(1)
                escenario.printPar(par)                                       //O(??)
                escenario.enviarPar(par)                                      //O(??)
            else
                exitoBackLampara ← escenario.backtrackFarolero()              //O(??)
            end if
        else
            creo entero farolero con faroleroPosible(escenario)               //O(N?)
            if farolero>-1 ∧ (minimo == -1 ∨ escenario.tiempo < minimo then  //O(1)
                escenario.printPersona(farolero)                              //O(??)
                escenario.enviarFarolero(farolero)                            //O(??)
            else
                exitoBackPar ← escenario.backtrackPar()                       //O(??)
            end if
        end if
    end while
end function
```

Complejidad: O(??)

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

Algoritmo 3 CRUZANDO EL PUENTE

```
function parPosible→ out res: Integer
    creo entero parEvaluar con parActual_x_paso[paso] + 1                 //O(1)
    creo entero tot_pares con arq_totales*can_totales                     //O(1)
    while ≠parValido(parEvaluar) ∧ parEvaluar ≤ tot_pares do              //O(N??)
        parEvaluar++                                                      //O(1)
    end while
    if parEvaluar ≤ tot_pares then                                        //O(1)
        devolver parEvaluar                                               //O(1)
    else
        devolver -1                                                       //O(1)
    end if
end function
```

Complejidad: O(N??)

Algoritmo 4 CRUZANDO EL PUENTE

```
function parValidoin par: Bool→ out res: Integer
    creo entero aux_arq_destino con aux_arq_destino                       //O(1)
    creo entero aux_can_destino con can_destino                           //O(1)
    if par > arq_totales ∗ can_totales then                               //O(1)
        devolver falso                                                    //O(1)
    end if
    creo a con primero(par)                                               //O(1)
    creo b con segundo(par)                                               //O(1)
    if ≠((esCanibal(a) ∨ esArquitecto(a)) ∧ (esCanibal(b) ∨ esArquitecto(b))) then   //O(1)
        devolver falso                                                    //O(1)
    end if
    aux_can_destino ← esCanibal(a) ∧ canibal_origen[a]                    //O(1)
    aux_arq_destino ← esArquitecto(a) ∧ arquitecto_origen[a]              //O(1)
    aux_can_destino ← esCanibal(b) ∧ canibal_origen[b]                    //O(1)
    aux_arq_destino ← esArquitecto(b) ∧ arquitecto_origen[b]              //O(1)
    creo arq_origen con arq_totales - aux_arq_destino                     //O(1)
    creo can_origen con can_totales - aux_can_destino                     //O(1)
    if arq_origen ≥ can_origen ∧ arq_destino ≥ can_destino then           //O(1)
        devolver verdadero                                                //O(1)
    else
        devolver falso                                                    //O(1)
    end if
end function
```

Complejidad: O(???)

1.4 Análisis de complejidades

ACA IRIA LOS COMENTARIOS DE COMPLEJIDAD Y LA DEMOSTRACION

1.5 Experimientos 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 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 iresde 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

```
function ALGORITMO(in LongLong : P)→ out S : Long Long out T : Long Long out arrayI : List<Long Long> out arrayD : List<Long Long>
    creo variable long long equilibrioActual = P                                   //O(1)
    creo variable long long i = 0                                                  //O(1)
    creo variable long long sumaParcial = 0                                        //O(1)
    while sumaParcial < P do                                                       //O(√P)
        sumaParcial ← sumaParcial + 3^i                                            //O(1)
        i++                                                                        //O(1)
    end while
    creo long long size = i+1                                                      //O(1)
    creo long long sumasParciales[size]                                            //O(√P)
    while i ≥ 0 do                                                                 //O(√P)
        sumasParciales[i] ← sumaParcial                                            //O(1)
        sumaParcial ← sumaParcial - 3^(i−1)                                        //O(1)
        i–                                                                         //O(1)
    end while
    creo long long middle = size/2                                                 //O(1)
    while |equilibrioActual| > 0 do                                                //O(lg(√P))
        if sumasParciales[middle] ≥ |equilibrioActual|
            ∧ sumasParciales[middle-1] < |equilibrioActual| then                   //O(1)
            creo long long potencia = sumasParciales[middle]-sumasParciales[middle-1]   //O(1)
            if equilibrioActual < 0 then                                           //O(1)
                equilibrioActual ← potencia + equilibrioActual                     //O(1)
                arrayD ∪ potencia                                                  //O(1)
            else
                equilibrioActual ← equilibrioActual -potencia                      //O(1)
                arrayI ∪ potencia                                                  //O(1)
            end if
            size ← middle                                                          //O(1)
            middle ← middle/2                                                      //O(1)
        end if
        if sumasParciales[middle] < |equilibrioActual| then                        //O(1)
            middle ← middle + size/2                                               //O(1)
        end if
        if sumasParciales[middle-1] ≥ |equilibrioActual| then                      //O(1)
            size ← middle                                                          //O(1)
            middle ← middle + size/2                                               //O(1)
        end if
    end while
    devolver(armadoBalanza)                                                        //O(√P)
end function
```

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

Algoritmo 6 armadoBalanza

function ARMADOBALANZA(:)→ **out** S : `Integer` **out** T : `Integer` **out** $arrayI$: `List<Integer>` **out** $arrayD$: `List<Integer>`
    invertir(arrayD) //O($\sqrt{P}$)
    invertir(arrayI) //O($\sqrt{P}$)
    devolver arrayD.tamaño //O(1)
    devolver arrayI.tamaño //O(1)
    devolver(arrayD) //O($\sqrt{P}$)
    devolver(arrayI) //O($\sqrt{P}$)
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 $O(\#\text{elementos}) \leq O(\sqrt{P})$.

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: $O(\sqrt{P})$ $[O(1) + O(1)] + O(1) + O(\sqrt{P}) + O(\sqrt{P})$ $[O(1) + O(1)] + O(\lg(\sqrt{P}))[O(1) + O(1) + O(1) + O(1) + O(1) + O(1)] = \mathbf{O(\sqrt{P})}$

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

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

Pues veamos que

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

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

$$3^{n+1} = 3 * 3^n. \quad (5)$$

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)

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

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 unica vez el segundo y tercer ciclo.

El valor de entrada P es de la forma $3^i + R$ para (i, R) ←[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 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 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 observacion desarrollamos el siguiente grafico con las instancias:

Y dividiendo por la complejidad de nuestro algoritmo llegamos a:

Para realizar esta experimentación nos parecio 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 mas 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 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 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 lass ultimas 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 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
function CALCULATE(in currentIdx: Integer)
    if remainingFriendships = 0 then                                        //O(1)
        sittingGirls ← copy(girls)                                          //O(n)
        subList ← sittingGirls[currentIdx, size(girls)]                     //O(1)
        sort(subList)              //O((k * log(k)) where k = size(girls) - currentIdx

        if currentSum < currentMin then                                     //O(1)
            bestRound ← sittingGirls                                        //O(1)
        else
            bestRound ← firstLexicographically(bestRound, sittingGirls)     //O(n)
        end if
        currentMin ← currentSum                                             //O(1)
    else
        for swapIdx from currentIdx to size(girls) do
            swap(girls, currentIdx, swapIdx)                                //O(1)
            partialDistance ← getPartialDistance(currentIdx)                //O(n log(n))
            currentSum ← currentSum + partialDistance                       //O(1)
            if currentSum ≤ currentMin then                                 //O(1)
                calculate(currentIdx + 1)
            end if
            swap(girls, currentIdx, swapIdx)                                //O(1)
            currentSum ← currentSum - partialDistance                       //O(1)
        end for
    end if
end function
```

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

Algoritmo 8 getMaxDistance

```
function GETPARTIALDISTANCE(in currentIdx: Integer, out res: Integer)
    sum ← 0                                                                 //O(1)
    count ← 0                                                               //O(1)
    for girlIdx from 0 to currentIdx do                                     //O(n)
        friendship ← Friendship(girls[girlIdx], girls[currentIdx])          //O(1)
        if contains(friendships, friendship) then                           //O(log(n))
            idxDiff ← rightIdx - leftIdx                                    //O(1)
            distance ← min(idxDiff, size(girls) - idxDiff)                  //O(1)
            sum ← sum + distance                                            //O(1)
            count ← count + 1                                               //O(1)
        end if
    end for
end function
```

Complejidad: O($nlog(n)$)

3.4 Análisis de complejidades

RESOLUCION DEL PUNTO Y ANALISIS