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

```
1:  function MAIN(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 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:

Como las pesas, presentan un peso en potencia de 3, recorremos 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 tambien otra variable *equilibrioActual* que inicializaremos con el valor de P y por ultimo, 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 teniamos 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 finalizariamos 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 todavia no se llego 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 5 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 ≥ P                                          //O(√P)
4:     equilibrioActual ← P                                                     //O(1)
5:     creo arrayPesasUtilizadas vacio                                          //O(1)
6:     estaEnNegativo ← falso                                                   //O(1)
7:     pesaActual ← 3^i                                                         //O(1)
8:     N = equilibrioActual - pesaActual                                        //O(1)
9:     while Terminar = False do                                                //O(√P)
10:        if N == 0 then                                                       //O(1)
11:            arrayPesasUtilizadas ∪ pesaActual                                //O(1)
12:            Terminar                                                         //O(1)
13:        end if
14:        if |N| == 1 then                                                     //O(1)
15:            arrayPesasUtilizadas ∪ pesaActual                                //O(1)
16:            arrayPesasUtilizadas ∪ 3^0                                       //O(1)
17:            Terminar                                                         //O(1)
18:        end if
19:        if |N| > 1 ∧ |N| < equilibrioActual then                             //O(1)
20:            arrayPesasUtilizadas ∪ pesaActual                                //O(1)
21:            i–                                                               //O(1)
22:            equilibrioActual ← N                                             //O(1)
23:            if N < 0 then                                                    //O(1)
24:                estaEnNegativo ← verdadero                                   //O(1)
25:            else
26:                estaEnNegativo ← falso                                       //O(1)
27:            end if
28:            equilibrioActual ← |N|                                           //O(1)
29:        end if
30:        if |N| ≥ equilibrioActual then                                       //O(1)
31:            i–                                                               //O(1)
32:        end if
33:        if estaEnNegativo then                                               //O(1)
34:            pesaActual ← 3^i * (-1)                                          //O(1)
35:            pesaUno ← 3^0 * (-1)                                             //O(1)
36:        else
37:            pesaActual ← 3^i                                                 //O(1)
38:            pesaUno ← 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 6 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(√P)
4:         if arrayPesasUtilizadas[x] < 0 then                             //O(1)
5:             arrayD ∪ arrayPesasUtilizadas[x]*(-1)                       //O(1)
6:         else
7:             arrayI ∪ 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(√P)
14:    devolver invertido(arrayI)                                          //O(√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 vacio, por lo tanto, como son enteros y el array es vacio 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 busqueda 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 opcion no finaliza el ciclo, agrega la pesa n el array, modifica el valor de saldoEnBalanza por el valor de N y disminuye en 1 a i, luego se chequea aqui 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 aqui 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 recorremos el array *arrayPesasUtilizadas* el cual en el peor de los casos tendrá todas las pesas, lo cual como vimoss 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

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 ptima 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