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		
Kapobel, Rodrigo		rok_35@live.com.ar
Rey, Esteban		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

ACA IRIA LA EXPLICACION DE LA SOLUCION

1.3 Algoritmos

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

Algoritmo 1 CRUZANDO EL PUENTE

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

Complejidad: O(n)

1.4 Análisis de complejidades

ACA IRIA LOS COMENTARIOS DE COMPLEJIDAD Y LA DEMOSTRACION

1.5 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, crearemos un array el cual nombraremos *arrayPesasUtilizadas* y crearemos tambien otra variable *saldoEnBalanza* que inicializaremos con el valor de P.

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

Inicialmente, restamos al valor de *saldoEnBalanza* 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 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 *saldoEnBalanza*, 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 *saldoEnBalanza* con el valor de N.

- 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.

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 al valor inicial de P iremos restando cada una de las pesas, hasta llegar a 0, en caso de que alguna diferencia diera negativa esa pesa irá al array *arrayIzquierda* mientras que el resto, irán en el array *arrayDerecha* para así poder dejar la balanza como se encontraba inicialmente con la llave.

2.3 Algoritmos

ACA EL PSEUDOCODIGO

Algoritmo 2 BALANZA

```
1:  function SOLVE(in list: List<Integer>)→ out res: List<Integer>
2:      if list == null then                                          //O(1)
3:          res ← Error                                               //O(1)
4:      end if
5:      List<Integer> ret ← List<Integer>(vacio)                      //O(1)
6:      Heap medianCalculator ← Heap(list.size)                       //O(log(N))
7:      Integer i ← 0                                                 //O(1)
8:      while i < list do                                             //O(N)
9:          Integer element ← list(i)                                 //O(1)
10:         medianCalculator.Agregar(element)                         //O(log(N))
11:         ret.Agregar(medianCalculator.getMedian())                 //O(log(N))
12:         i++                                                       //O(1)
13:     end while
14:     res ← ret                                                     //O(N.log(N))
15: end function
```

Complejidad: O(N.log(N))

2.4 Análisis de complejidades

2.5 Experimientos y conclusiones

2.5.1 Test

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

A continuación enunciaremos varios de nuestros tests:

Ambas ramas estan Ordenadas

Este caso se cumple cuando se recibe una lista con una sucesión con una pinta de la forma:
i ← n/2 [X_i, X_i+1]

Ambas ramas se encuentran desordenadas

Este caso se cumple cuando se recibe una lista con una sucesión con una pinta de la forma:
i ← n/2 [X_i, X_n-i+1]

2.5.2 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

DESCRIPCION DEL EJERCICIO 3
PROBLEMA DE LA MOCHILA

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. Se representa a la ronda como un array donde cada posición equivale a un lugar para sentarse, y su valor en dicha posición indica quién esta sentado allí. Para evaluar todas las posibles configuraciones, vamos sentando a las niñas desde la posición 1 a la posición |ronda| en el array. Para esto se implementa una función que dado el índice que indica la siguiente posición a ocupar, prueba sentar a cada chica posible (es decir, dentro de las que no estan sentadas aún) en la posición vacía y llama recursivamente aumentando en uno el índice (Cuando finaliza cada llamada recursiva, se "vuelve hacia atras" y se coloca a la niña que estaba originalmente en esa posición). Cuando el array (o la ronda) está llena, ya se puede calcular la suma de las distancias y por lo tanto se puede decidir si es la mejor solución encontrada (hasta ese momento).

Se puede demostrar por inducción en la cantidad de personas que esta función evalúa todas las posibilidades.

- Caso base: Si hay solo una chica, la función calcula la suma de distancias para la única configuración posible.
- Paso inductivo: Si la cantidad de chicas es $n > 1$. Como la función coloca todos las posibles en la primera posicioń, y para las restantes llama recursivamente ($n-1$ chicas). Por nuestra hipótesis inductiva, sabemos que la función para $n-1$ chicas evalúa todas las posibles. Por lo tanto, como la función para n se llama recursivamente para $n-1$ para todas las particiones posibles de [chica de la primer posicion] - [chicas de las demas posiciones], podemos afirmar por inducción que f evalúa todas las permutaciones de las n chicas.

Como nuestro algoritmo pasa por todas las permutaciones posibles y estas son $n!$, su complejidad tiene un factor $n!$. Calcular la suma de distancias de todas las amigas tiene costo $n^2 * log(n)$, ya que para cada pareja de chicas se hace lo siguiente:

- Se fija si son amigas (Costo $log(n)$ por la implementacion de set)
- Si son amigas se calcula la distancia. (Costo $O(1)$)

Podas: Se puede mejorar el algoritmo propuesto anteriormente efectuando algunas podas. Es decir, si pensamos al *backtracking* como un árbol en el nos vamos moviendo por diversas posiciones intermedias (nodos internos) hasta llegar a las finales (hojas). Si a partir de un nodo intermedio podemos afirmar que todas las soluciones que derivan del mismo no son la óptima, podemos rechazarlas y pasar a analizar otras ramas, ahorrando asi un importante costo en cálculos. A esta técnica se la conoce como "podas" (o *backtrack*) en un *backtracking*.

- Poda de permutaciones circulares: Como estamos representando a la ronda en una array, existen permutaciones equivalentes a la misma ronda. Por ejemplo, ABC es equivalente a BCA. Este problema se soluciona dejando fija a la niña de la primera posición. Ahora bien, como el problema nos pide devolver la configuración con el menor orden lexicográfico, fijamos a la niña de menor orden lexicográfico en la primera posición del array. De esta manera todas las soluciones generadas serán la de menor orden lexicográfico entre sus equivalentes.

- Poda de distancia parcial: Si en vez de calcular las distancias al final las calculamos a medida que agregamos a alguien nuevo, podemos calcular esta suma parcial con la mejor suma obtenida hasta ahora. Es decir, si en un "nodo interno" ya nos pasamos de la mejor suma obtenida hasta ese momento, no tiene sentido seguir investigando esta rama del árbol (ya que esta suma sólo puede aumentar, y por lo tanto nunca será la mejor). ¿Suma complejidad calcular la suma de distancias parcial cada vez que sentamos a una niña? Si bien esto parece agregar complejidad, lo cierto es que cada vez que se sienta una niña sólo se actualiza la suma actual con las distancias de las amistades de la niña agregada, es decir, no se vuelven a mirar amistadas ya miradas previamente. Por lo tanto, para cada hoja del arbol del *backtracking*, solo habremos mirado una vez cada pareja de niñas, que no es más veces que en el caso sin la poda. De hecho, sin esta poda hay distancias que se calculan repetidas veces. Si calculamos las distancias al final, (A y B amigos) para las permutaciones $ABCD$ y $ABDC$ estaremos calculando varias veces la distancia entre A y B, mientras que con la poda esa distancia ya la tenemos calculada.

- Poda de amistades restantes: Resulta evidente que si ninguna de las niñas que resta sentar tiene amigas, no tiene sentido probar todas las permutaciones, ya que el orden en que se sienten no modificará la suma total. Por lo tanto, esta poda consiste en llevar la cuenta de cuántas amistades ya se calculó la distancia (i.e. ambas niñas de la amistad están sentadas) y si ya no quedan amistades para calcular ordena a las siguientes niñas. Esto último es porque de todas las permutaciones queremos la de menor orden lexicográfico.

3.3 Algoritmos

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

Algoritmo 3 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 4 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