FACULTAD DE INGENIERÍA
Universidad de Buenos Aires

75.29 / 95.06 Teoría de Algoritmos

Trabajo Práctico Nº 1

Teoría de Logaritmos

Fecha de Entrega: Lunes 23 de Septiembre de 2024

Integrantes:

Alumno	Padrón
Joaquin Andresen	102707
Federico Penic	102501
Facundo Pareja	99719
Federico Pratto	96223
Rogger Aldair Paredes Tavara	97976

Índice

Parte 1: Maximizando la ganancia del proyecto

1. Explique cómo resolvería el problema utilizando generar y probar. ¿Cuál sería la complejidad?

En el enunciado se plantean las siguientes consideraciones:

1. Los proyectos están compuestos por tareas.
2. Cada tarea puede tener otras tareas que deben ser realizadas primero para poder comenzarla.
3. Solo se puede hacer una tarea a la vez.
4. La ganancia de cada tarea varía según la cantidad de tareas que se realizaron anteriormente.
5. Sabemos cual es la ganancia de cada tarea por cada orden posible de ejecución.
6. No importa la duración de las tareas.

Se nos pide el orden de todas las tareas para que se obtenga la mayor ganancia posible y a su vez se cumplan todas las precedencias, como se detalla en el segundo punto.

En este caso se nos pide resolverlo utilizando “Generar y Probar”.

Para identificar cada tarea utilizaremos un número 1 ... N .

Por su parte, en un vector podemos representar el orden en el cual se ejecutaron las tareas, de izquierda a derecha. Ejemplo:

3	4	1	7	2	5	6

Esto quiere decir que primero se ejecutó la tarea 3, luego la 4, luego la 1, etc. hasta que finalmente se ejecutó la tarea 6.

De esta manera, podemos representar todas las soluciones posibles. Deberemos generar todos los vectores de N elementos, donde N es la cantidad de tareas a realizar. Para esto recurrimos a las denominadas “Permutaciones”. Esa será nuestra **función generativa**.

Una vez tengamos todas las soluciones posibles, tendremos que hacer foco en las restricciones del problema, en nuestro caso, las precedencias de las diferentes tareas.

Por lo tanto, en nuestro algoritmo tendríamos que considerar que aquellos órdenes que no cumplen con la precedencia de tareas no son válidos y por lo tanto tienen una ganancia nula.

Para el resto de órdenes válidos, es cuestión de probar todos calculando las ganancias concretas. Una vez se han revisado todos los posibles órdenes, se puede obtener aquel orden que cumpla con las precedencias establecidas que maximiza las ganancias del proyecto. Esto lo podemos lograr, ya que conocemos la ganancia de cada tarea según cada orden posible de ejecución, como se indica en el punto 5.

Si bien no es el algoritmo más eficiente, se nos solicitó en este punto "Generar y Probar", es decir, generar todas las soluciones posibles, probarlas todas, y obtener la solución óptima del problema.

En cuanto a la complejidad temporal, para generar todas los órdenes posibles utilizando permutaciones contamos con O(n!).

Para cada uno de estos órdenes o permutaciones, se tendrán que verificar las precedencias de las tareas. Aquí, se cuenta con un O(n), ya que en el peor de los casos se tendrán que verificar n-1 tareas previas para asegurarse que se cumple con las precedencias.

Por último para calcular las ganancias de cada orden, tenemos un O(n).

La complejidad temporal será entonces: O($n \,.\, n!$)

2. Proponga y explique una solución del problema mediante Branch and Bound (B&B)

La solución pensada se estructura alrededor del recorrido de un árbol de estados, aplicando las funciones de límite y costo para podarlo donde corresponde. La estructura del mismo es tal que:

- Los estados representan tareas.
- La raíz representa un estado en el que no se realizó tarea alguna.
- Cada nivel del árbol representa el orden de realización de la tarea ahí escogida (por ejemplo, la rama del árbol correspondiente al nodo de la tarea 4 en el primer nivel significa que esa tarea fue realizada primero).

Las funciones características de backtracking y branch and bound respectivamente son:

- La función **límite** que verifica que se pueda realizar una tarea en base a las tareas realizadas anteriormente. En caso negativo ese orden de realización de tareas no es viable, y por lo tanto se poda esa rama del árbol.
- La función **costo** que verifica que, si para cada tarea restante hasta terminar se tuviese una ganancia equivalente a la máxima ganancia de cada una de estas semanas restantes, se obtendría una mayor

ganancia potencial que la calculada actualmente. En caso negativo se poda la rama del árbol, puesto que no hay forma de que explorando esa rama se obtenga una mayor ganancia que la de la rama con el beneficio máximo asociado.

3.Brinde pseudocódigo y estructuras de datos a utilizar para B&B

```
Sea R un diccionario de listas donde R[x] indica las  tareas requeridas para
poder realizar la tarea x.
Sea V un diccionario de listas donde V[s] son las ganancias de las tareas
ordenadas de mayor a menor  para la semana s.
Sea M un diccionario de valores donde M[s] es el valor máximo de entre todas las
ganancias para semanas superiores  a s.
Sea tareas la lista de tareas a ejecutar tal que su posición en el vector indica
el orden de realización.

Sea NRO_SEMANAS la cantidad de semanas total disponible.
Sea MEJOR_RESULTADO la lista de tareas a realizar para obtener la mayor ganancia.
Sea MEJOR_GANANCIA  la ganancia máxima hasta el momento.

backtrack(tareas)
       Sea semana = len(tareas) + 1
Sea ganancia_actual suma de las ganancias según el orden en el vector tareas.
       si semana == NRO_SEMANAS o no quedan tareas para realizar:
              si mejor_ganancia < ganancia_actual:
              MEJOR_GANANCIA = ganancia_actual
              MEJOR_RESULTADO = tareas
       sino:
para cada tarea en V[semana]:
                     Si tarea no está en tareas y R[tarea] está(n)  realizada(s):
                     tareas += tarea
                     Ganancia_tarea = ganancia de la tarea para esa semana
                     ganancia_actual += ganancia tarea
                     Sea maxima_ganancia_posible = ganancia_actual + (NRO_SEMANAS
- semana) * M[semana]
                     Si maxima_ganancia_posible > mejor_ganancia:
                            backtrack(tareas)
                     tarea.pop()
```

4.Realice el análisis de complejidad temporal y espacial para B&B

A lo largo de todo este análisis se utiliza el siguiente símbolo

- Número de tareas -> **n**

Complejidad temporal

Para realizar un análisis de la complejidad temporal, se deberá pensar en el peor caso posible. En nuestro caso, éste corresponde a recorrer todos los nodos de nuestro árbol de estados. Como estamos recorriendo todos los susodichos nodos, a su vez, deberemos verificar si se supera la función límite y además comprobar que se superen todas las precedencias para cada una de las tareas.

En el análisis, será relevante la cantidad de tareas precedentes que tiene cada una de las mismas. Por lo que podríamos plantear que en el peor de los casos cada tarea tiene n - 1 tareas precedentes. Si bien este valor no es factible en la realidad, ya que provocaría que las precedencias sean "cíclicas" impidiendo que se puedan realizar tareas, nos sirve para proponer una cota superior.

Por lo tanto:

- Recorrer cada estado del árbol de estados nos toma un trabajo de O(n^n).
- Por cada uno de estos estados, deberemos hacer dos verificaciones: La de la función límite, que requiere un trabajo de O(n) y la de las precedencias, que también requiere un trabajo de O(n).

Haciendo el producto de ambas complejidades, nos queda que la complejidad temporal, en el peor de los casos es:

$$O(n \cdot n^n)$$

Complejidad espacial

Para la ejecución del algoritmo se requiere:

- Un diccionario de n listas donde cada una tiene longitud n-1.
- Un diccionario de n listas donde cada una tiene longitud n.
- Un diccionario de longitud n.
- Una lista de longitud n.

Por lo tanto, sumando todos los costos espaciales, la complejidad espacial resulta:

$$O(n^2)$$

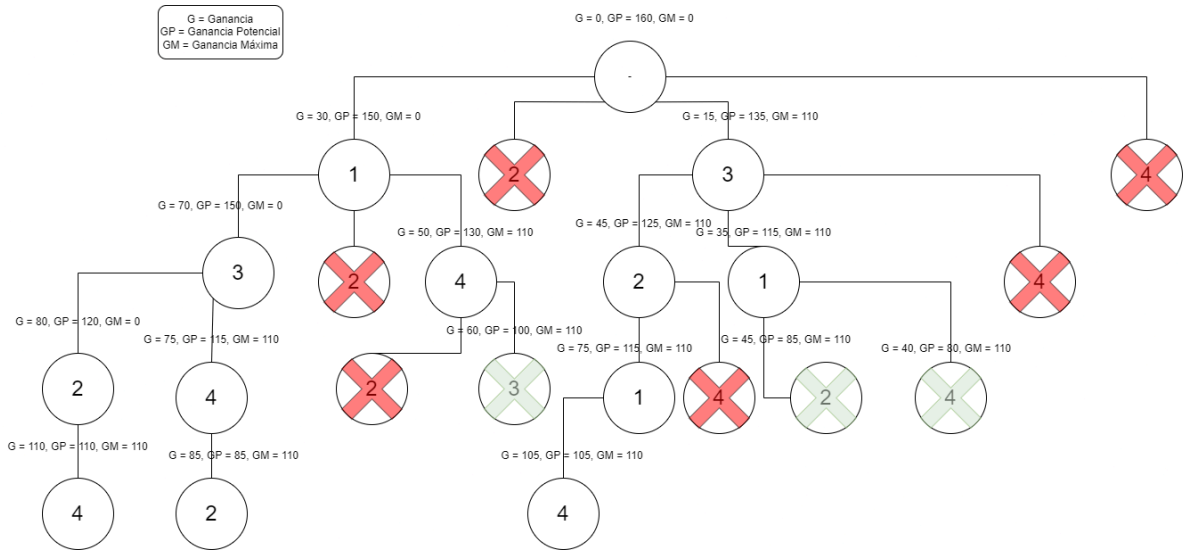
5. Brinde un ejemplo simple paso a paso del funcionamiento de su solución para B&B

Supongamos 4 tareas con las siguientes dependencias y ganancias por su realización:

Tarea	Ganancia	Dependencia

1	30,20,30,40	-
2	20,30,10,10	3
3	15,40,10,10	-
4	10,20,5,30	1

Resulta el siguiente árbol (el algoritmo se ejecuta de izquierda a derecha en profundidad):



Obsérvese que:

- El orden de análisis de los descendientes de cada nodo se corresponde con un orden mayor a menor en cada semana.
- Las cruces rojas indican que no se analiza ese descendiente por no tener las tareas requeridas
- Las cruces verdes indican que no se analiza ese descendiente por no poder superar el valor actual de ganancia incluso teniendo en cuenta el mejor caso posible.
- Para cada vértice se indica:
    - Ganancia hasta ese vértice inclusive.
    - Ganancia potencial en el mejor caso posible si su descendientes tienen la ganancia máxima de entre todas las tareas posibles.
    - Ganancia máxima hasta el momento.

Las tareas elegidas con la mayor ganancia posible resultan entonces [1, 3, 2,4]

6. Programe su propuesta de B&B

El programa resuelto se encuentra en el siguiente repositorio. Para correr el programa el único requisito es tener instalado Docker.

Se deberá buildear el contenedor en el directorio en el cual se encuentra el archivo Dockerfile con el comando:

```
docker build -t tp_tda:ej1 .
```

Y luego correr el contenedor con el comando:

```
docker run -it tp_tda:ej1
```

Alternativamente, se puede correr el programa con python instalado localmente utilizando el comando:

```
python3 proyecto.py tareas.txt ganancias.txt
```

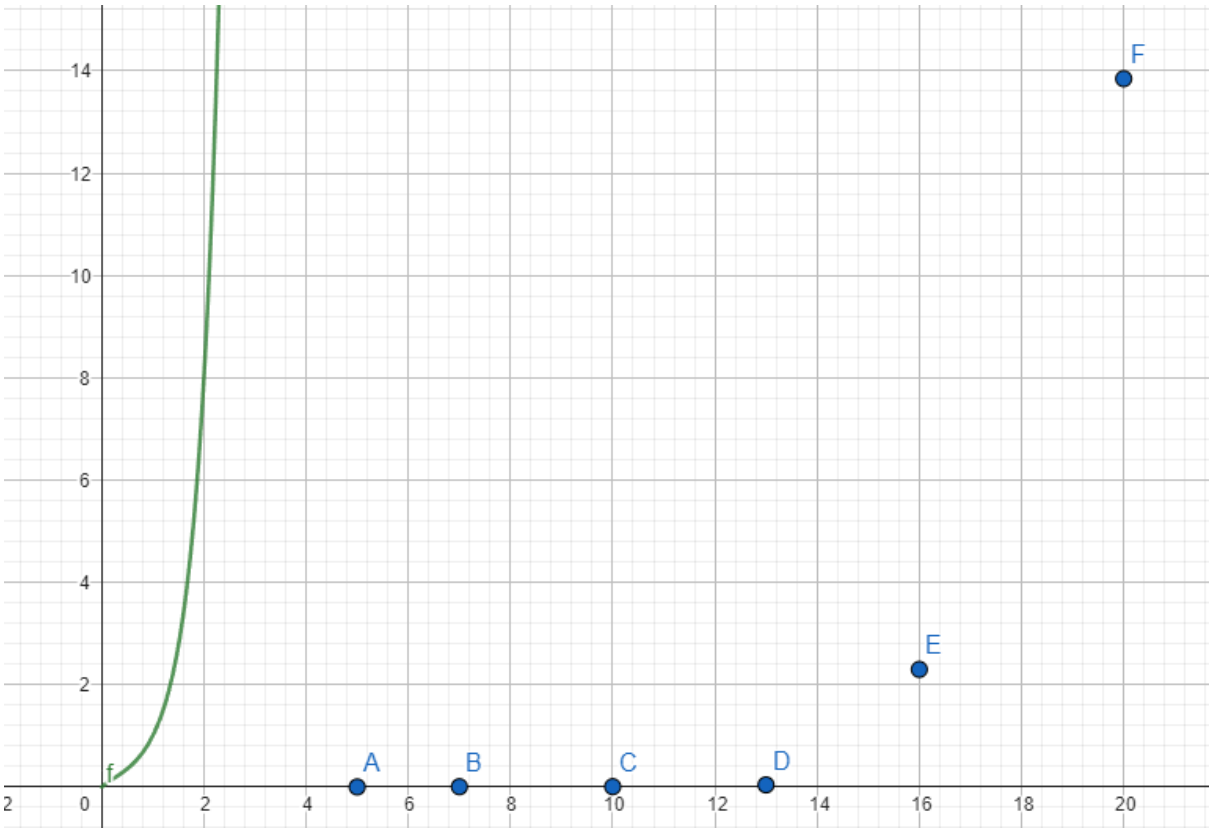
7. Determine si su programa tiene la misma complejidad que su propuesta teórica

En pos de determinar la complejidad de nuestro programa, decidimos crear un script que se encargue de generar información aleatoria para los archivos de tareas.txt y ganancias.txt (Incluyendo cantidad de precedencias). Una vez hecho esto, corrimos el programa para diferentes cantidades de tareas y medimos los tiempos de resolución de nuestro algoritmo. Estos fueron los resultados de algunas pasadas:

Cantidad de Tareas	Tiempo de resolución (En segundos)
5	0.0001556873321533203
7	0.0047179718017578125
10	0.006201412200927734
13	0.040895709991455078
16	2.2979090213775635
20	13.84592890739441

Vale la pena destacar que estos valores expuestos fueron calculados como el promedio de 5 corridas con la cantidad de tareas respectivas, ya que dependiendo de la información aleatoria que se generaba en cada corrida, los tiempos de resolución variaban.

Posteriormente, decidimos comparar los puntos de la tabla con la complejidad temporal teórica propuesta en el punto 4 ($n \cdot n^n$). Este fue el resultado (Donde el eje X representa la cantidad de tareas y el eje Y el tiempo en segundos):



Si bien los puntos no coinciden con la función expresada $n \cdot n^n$ (En verde) en los primeros 4 puntos (Hasta 13 tareas), hacen un salto exponencial con el número de tareas 16 y 20. Por lo tanto, parecería que a mayor cantidad de tareas, más se va a acercar a una función de ese estilo.

Aunque es correcto mencionar que la función $n \cdot n^n$ hace alusión al peor de los casos, e incluso se tomaron supuestos (Como la cantidad de precedencias) que nunca se deberían cumplir. Por dichos motivos, no deberían ser exactamente iguales las complejidades teóricas y prácticas.

Parte 2: El costo del mantenimiento

1. Determinar y explicar cómo se resolvería este problema utilizando la metodología greedy

Considerando una conexión la unión entre dos nodos, con un coste **'k'** constante con un ingreso por patrocinador **'y'** variable, lo que haremos es:

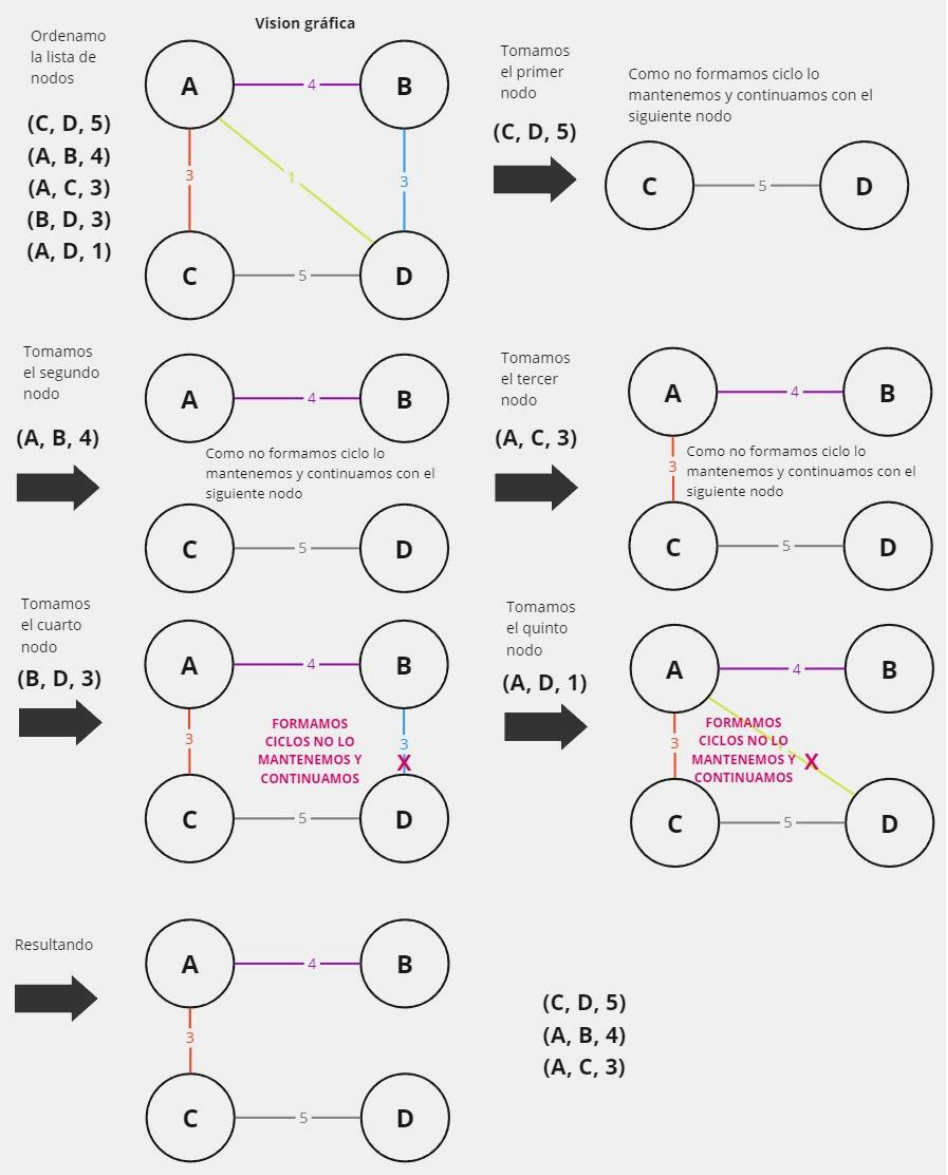
- Primero lo que hacemos es ordenar la lista de mayor a menor **ingreso por patrocinador**
- Después recorreremos la lista ordenada
- nos fijamos si al agregar la primera arista formamos ciclo
- si no hay ciclos, agregamos la arista a nuestra grafo resultante
- si formamos ciclos, continuamos con la siguiente arista

2. Brinde pseudocódigo y estructuras de datos a utilizar

```
resolucion(COSTE_POR_CONEXION, conexiones):
    grafo = inicializar_grafo()
    total_conexiones = 0
    conexiones_ordenadas = ordenar_por_ingreso(conexiones)

    por cada conexion en conexiones_ordenadas:
        nodo1, nodo2, ingresoPatrocinador = conexion
        si no forma_ciclos(nodo1, nodo2, grafo):
            agregar_conexion(grafo, nodo1, nodo2)
            total_conexiones++

    return total_conexiones * COSTE_POR_CONEXION

forma_ciclos(nodo1, nodo2, grafo):
    visitados = conjunto_vacio()
    pila = [nodo1]

    mientras pila no esté vacía:
        nodo_actual = pop(pila)
        si nodo_actual es nodo2:
            return true
        si nodo_actual no está en visitados:
            agregar a visitados(nodo_actual)
            por cada vecino en vecinos(nodo_actual, grafo):
                si vecino no está en visitados:
                    push(pila, vecino)

    return false
```

```
agregar_conexion(grafo, nodo1, nodo2):
    agregar_arista(grafo, nodo1, nodo2)
    agregar_arista(grafo, nodo2, nodo1)

ordenar_por_ingreso(conexiones):
    return ordenar(conexiones, por ingreso de patrocinador de mayor a menor)
```

3. De un ejemplo paso a paso. ¿Qué complejidad temporal y espacial tiene la solución?



Para la complejidad temporal: Vemos que para ordenar las conexiones podemos usar una algoritmo mergesort con coste $O(aLoga)$, $a = cant\ conexiones$, para revisar si formo ciclos vamos a tener que revisar tanto los nodos y las aristas usando una algoritmo DFS $O(n + a)$ siendo n = cantidad de nodos, a = cantidad de aristas en las conexiones y al agregar una conexión $O(1)$, pero esto lo estaremos realizando $a\ veces$ lo que nos dejaría una complejidad de $O(a(n + a))$

Para la complejidad espacial: en el ordenar las conexiones es $O(1)$ ya que son funciones básicas en la revisión de formar ciclos lo también será $O(n + a)$ y el agregar conexiones es solo $O(1)$, siendo la complejidad espacial total $O(n + a)$

4. Justifique por qué corresponde su propuesta a la metodología greedy

El enfoque es greedy pues tenemos:

- **Elección greedy:** Por que al analizar cada arista por el maryor ingreso estaremos seleccionando nuestro óptimo local.
- **Subestructuras óptimas:** Pues iremos pasando lo que da el grafo en la siguiente iteración , ya previamente habiendo seleccionado la conexión con el patrocinio más óptimo en ese momento.

5. Demuestre que su solución es óptima

Si existe una solución óptima O**,** con elementos O_1,O_2,...,O_N

Llamaremos a la solución greedy G, y a las conexiones elegidas g_1,g_2,...,g_N

Sí el Ingreso Total de **valor(**O**) > valor(**G**)** $\rightarrow O_i > g_i$, entonces quiere decir que tenemos al menos un conexión con un mayor ingreso que no fue incluida en G

Esto implica que Greedy no seleccionó una conexión con mayor ingreso según su programación

¡Lo que sería un absurdo! Por lo tanto greedy es óptimo

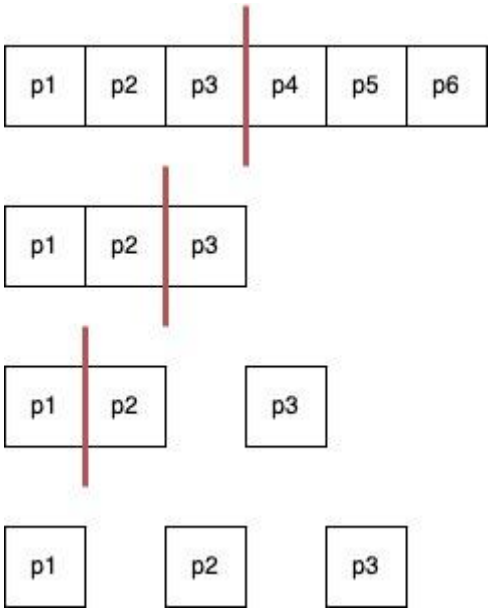
Parte 3: Los planos de las piezas

1. Presentar un algoritmo que lo resuelva utilizando división y conquista.

Sea P una lista de n piezas P = (p1, p2, p3, ..., pn) donde pi = (xi, h, xf). Queremos obtener el contorno de la combinación de todas las piezas de P.

El algoritmo para obtener el contorno consiste en los siguientes pasos:

1- Dividimos nuestro listado de piezas en la mitad hasta llegar al caso base



2- En el caso base tendremos una sola pieza a la cual le calcularemos su contorno

pi = (xi, h, xf)
c = [(xi, h), (xf, 0)]

3- Combinamos los contornos

Para obtener el contorno resultante entre dos debemos comparar las coordenadas x de ambos contornos, tomando siempre el punto que tenga el menor valor de x.

Tomamos la altura máxima en cada coordenada x de los dos contornos (uno a la izquierda, otro a la derecha). Si la altura cambia en relación con el último punto añadido al contorno combinado, añadimos el nuevo punto. De lo contrario, lo ignoramos.

Avanzamos en el contorno correspondiente (izquierdo o derecho) y repetimos el proceso hasta fusionar todos los puntos.

~~4~~ Seguimos hasta comparar todas las mitades

2. Mostrar la relación de recurrencia

Sea $T(n) = aT(\frac{n}{b}) + f(n)$ la relación de recurrencia.

Analizamos nuestro algoritmo

- La lista de partes se divide en dos mitades, por lo tanto b = 2.
- Al dividir, nos quedamos con las dos mitades, por lo tanto a = 2.
- Al combinar debemos recorrer una lista de coordenadas que en el peor de los casos contendrá a todas las piezas (suponiendo que ninguna intersecta con otra), por lo tanto $f(n) = \Theta(n)$.

Finalmente, la relación de recurrencia de nuestro algoritmo es:

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$

3. Presentar pseudocódigo

```
Sea piezas una lista de piezas donde cada pieza tiene la forma (x inicial,
altura, x final)

obtener_contorno(piezas):
    SI LONGITUD(piezas) ES 1:
        RETORNAR calcular_coordenadas(piezas[0])
    SI NO
        lado_izquierdo, lado_derecho <- dividir(piezas)
        lado_izquierdo <- obtener_contorno(lado_izquierdo)
        lado_derecho <- obtener_contorno(lado_derecho)
        RETORNAR calcular_contorno(lado_izquierdo, lado_derecho)

calcular_contorno(lado_izquierdo, lado_derecho)
    contorno <- LISTA VACÍA
    h1 <- 0, h2 <- 0    // Alturas de las piezas
    i <- 0, j <- 0      // Índices para recorrer lado_izquierdo y lado_derecho
    posición_actual <- 0
    altura_maxima <- 0

    MIENTRAS i < LONGITUD(lado_izquierdo) Y j < LONGITUD(lado_derecho)
        SI lado_izquierdo[i][0] < lado_derecho[j][0] ENTONCES
```

```
            posición_actual, h1 <- lado_izquierdo[i]
            altura_maxima <- MÁXIMO(h1, h2)
            i <- i + 1
        SI NO SI lado_izquierdo[i][0] > lado_derecho[j][0] ENTONCES
            posición_actual, h2 <- lado_derecho[j]
            altura_maxima <- MÁXIMO(h1, h2)
            j <- j + 1
        SI NO
            posición_actual <- lado_izquierdo[i][0]
            h1 <- lado_izquierdo[i][1]
            h2 <- lado_derecho[j][1]
            altura_maxima <- MÁXIMO(h1, h2)
            i <- i + 1
            j <- j + 1

        SI contorno ESTÁ VACÍO O contorno[-1][1] != altura_maxima ENTONCES
            AÑADIR (posición_actual, altura_maxima) A contorno

    AÑADIR las coordenadas restantes de left_piece[i:] A contorno
    AÑADIR las coordenadas restantes de right_piece[j:] A contorno

    RETORNAR contorno
```

4. Analice la complejidad del algoritmo utilizando el teorema maestro y desenrollando la recurrencia

Teorema Maestro:

Sean:

- $a \geq 1 \, y \, b \geq 1 \, constantes$
- $f(n) \, una \, función$
- $T(n) = aT(\frac{n}{b}) + f(n)$

Entonces tenemos 3 casos:

1) **Si** $f(n) = O(n^{log_b a - e}), \, e > 0 \Rightarrow T(n) = \Theta(n^{log_b a})$
2) **Si** $f(n) = \Theta(n^{log_b a}) \Rightarrow T(n) = \Theta(n^{log_b a} * log(n))$
3) **Si** $f(n) = \Omega(n^{log_b a + e}), \, e > 0 \Rightarrow T(n) = \Theta(n^{log_b a}) \, Y \, af(\frac{n}{b}) \leq cf(n), \, c < 1 \, y \, n >>$

Por el teorema tenemos:

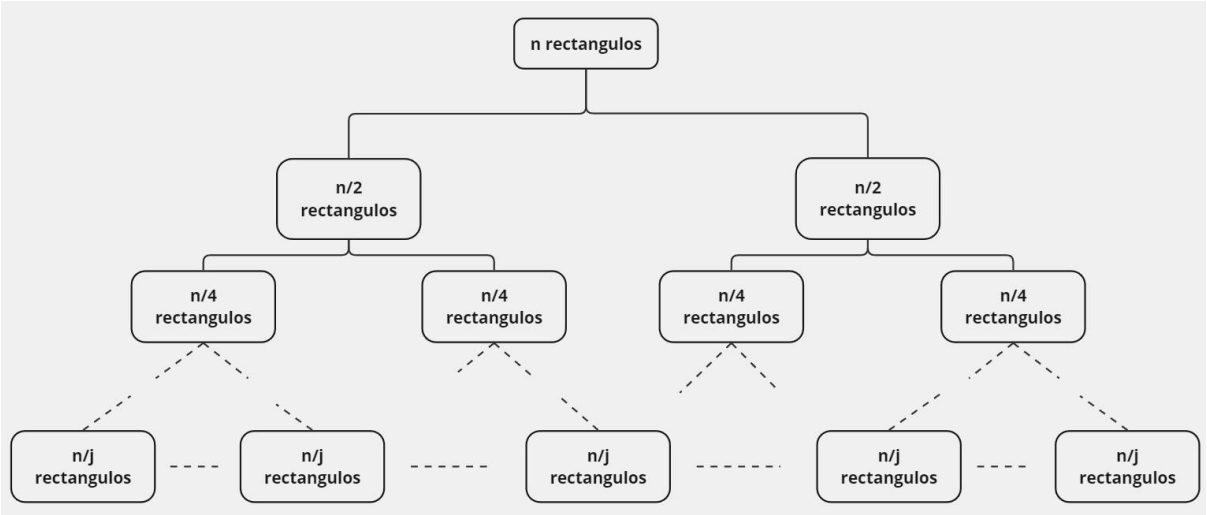
El problema se está llamando recursivamente 2 veces(a = 2) en los cuales se le está asignando la mitad de la cantidad de elemento(b = 2) que vaya tomando, y en la f(n) será la comparación en cada llamado del borde resultante con el cuadrado entrante, pudiendo tener n comparaciones.

$$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$

con lo cual obtendremos por el teorema, una complejidad temporal

$$f(n) = \Theta(n^{log_2 2}) = \Theta(n),\ la\ cual\ está\ acotada\ inferior\ y\ superiormente \Rightarrow T(n) = \Theta(n * log(n))$$

Desarrollo de Recurrencia



En cada nivel estaremos cn operaciones:

Nivel 0: cn
Nivel 1: $2 * cn/2 = cn$
Nivel 2: $4 * cn/4 = cn$
...
Nivel j: $2^j * cn/2^j = cn$

Si despejamos j nos quedaría

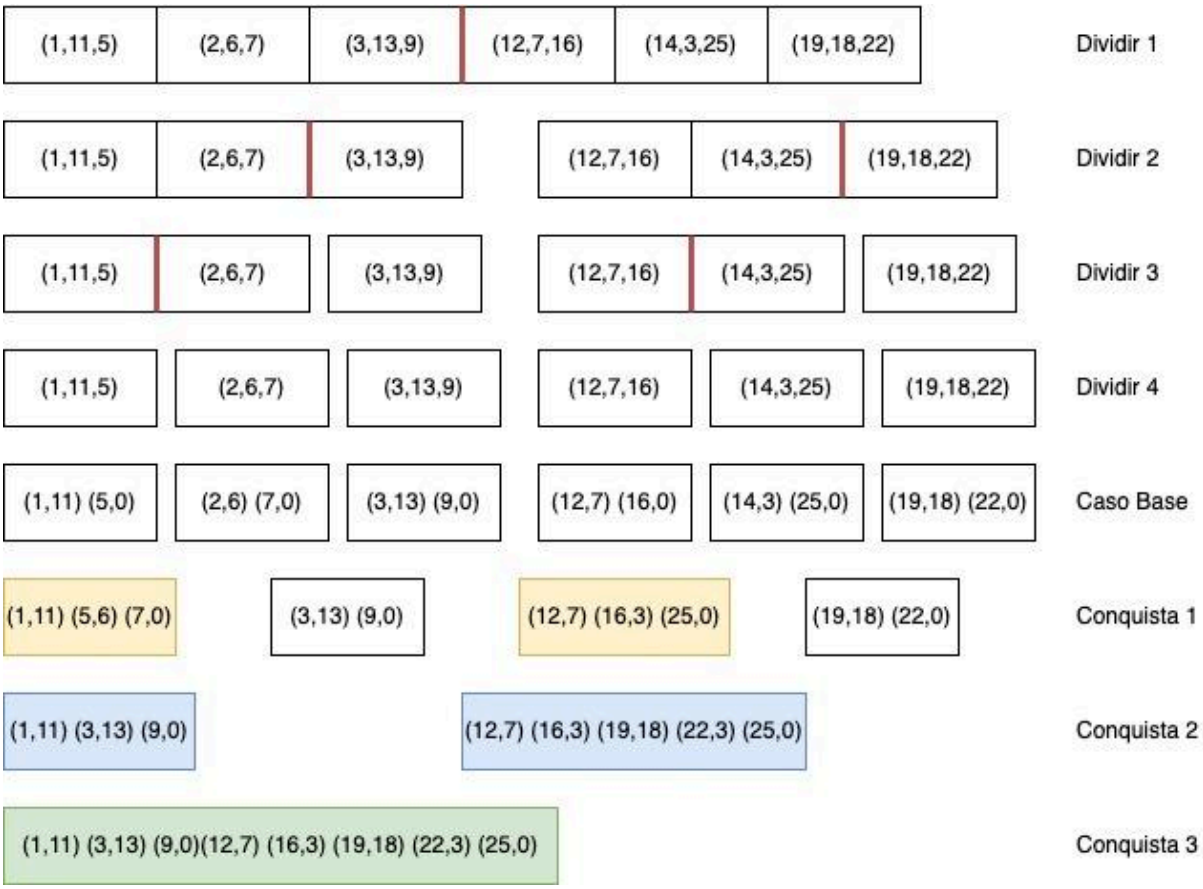
$$j = log_2 n$$

Y finalmente
$\Theta(n * log_2 n)$

5. Brindar un ejemplo de funcionamiento

Dada la lista de rectángulos dados vemos que en cada paso iremos dividiendo el problema **(Dividir i)** en 2 mitades llegando hasta su **caso base** donde obtenemos los contornos de cada pieza. Una vez allí iremos subiendo y comparando los contornos de cada pieza **(Conquistar i)**, cuyo contorno resultante de esa comparación se irá heredando y comparando con los otros contornos hasta obtener el contorno final.

La obtención del contorno se hace iniciando la comparación con los puntos iniciales de cada pieza e iniciando con la que esté antes, quedándonos con la altura máxima de los dos puntos comparados e ir agregando la de mayor altura con el punto obtenido y si tenemos un punto que tengamos que modificar la altura le cambiamos y agregamos al contorno resultante.



Ejemplo de Conquista 1:
lado izquierdo: [(1,11) (5,0)]
lado derecho: [(2,6),(7,0)]
h1 = 0
h2 = 0
posición actual = 0
altura máxima = max{h1, h2}

Comparo (1,11) con (2,6):
1 < 2, por lo tanto mi posición actual nueva es 1 y h1 es 11. Mi altura máxima por lo tanto queda en 11 ya que es mayor que h2 = 0. Guardo **(1,11)** y avanzó del lado izquierdo.

Comparo (5,0) con (2,6):

5 > 2, por lo tanto mi nueva posición actual es 2 y h2 es 6, pero mi altura máxima sigue siendo 11 ya que es el máximo entre h1 y h2. Como no hubo un cambio de altura máxima sigo.

Comparo (5,0) con (7,0):
5 < 7, mi nueva posición actual es 5 y mi nuevo h1 es 0. La altura máxima ahora es 6 ya que h2 es mayor al nuevo h1. Como hubo un cambio en la altura guardo **(5,6)**.

Como el único punto que me queda es **(7,0)** y no lo puedo comparar con nada lo agrego al contorno resultante. De esta manera me queda **(1,11),(5,6) y (7,0)**.

6. Programe su solución

Resolución de Algoritmo

Para ejecución del algoritmo lo que primero debemos hacer es posicionarnos en la TP1/parte_3 y ejecutar el comando:

- $python\ contorno.py\ partes.txt$

7. Analice si la complejidad de su programa es equivalente a la expuesta en el punto 4

Analizando cada una de las funciones que tiene el algoritmo podemos ver que los métodos

- **split_pieces(pieces) y calculate_coordinates(piece)** tiene una complejidad $O(1)$. ya que una solo particionamos sin procesar y en la siguiente solo obtenemos coordenadas numéricas.
- **read_file(file_path)** es $O(n)$, ya que podemos leer **n** elementos
- **calculate_contour(left_piece, right_piece)** en el peor de los casos estaremos comparando los n puntos lo que nos da $O(n)$
- **get_contour(pieces)** como en cada nivel vamos dividiendo la lista a la mitad la complejidad es logarítmica $O(log\ n)$.

Lo que nos da al combinar una complejidad temporal de $O(n\ log\ n)$**.**
En la complejidad espacial vamos a estar teniendo:

- **Espacio para almacenar las piezas**: $O(n)$ debido a la lista de piezas que contiene todas las piezas del archivo.
- **Espacio para la pila de recursión**: $O(log\ n)$, dado que la recursión tiene una profundidad de O(logn).
- **Espacio para combinar contornos**: $O(n)$ en cada nivel de recursión.

Como el espacio más solicitado viene de la lista de piezas y los contornos combinados obtenemos una complejidad espacial de $O(n)$**.**

Referencias

- Algoritmos y Complejidad - Algoritmos sobre Grafos
- Diferencia entre Branch & Bound y Backtracking