Universidad de Buenos Aires
Facultad de Ingeniería
1er Cuatrimestre de 2023

Teoría de Algoritmos I (75.29)

TRABAJO PRÁCTICO Nº2
FECHA: 17/05/2023
GRUPO: Dijkstra fan club

INTEGRANTES:

Ramirez Scarfiello, Nicolás Alberto <nramirez@fi.uba.ar>	- #102090
Ménager, Jeanne <jmenager.ext@fi.uba.ar>	- #2255240
Civini, Armando Tomás <acivini@fi.uba.ar>	- #104350
Puglisi, Agustín <apuglisi@fi.uba.ar>	- #104245
Rico, Mateo Julián <mrico@fi.uba.ar>	- #108127
Zuretti, Agustin Santiago <azuretti@fi.uba.ar>	- #95605

Índice

Primera entrega

1. Preselección de inversiones

1.1. Fuerza bruta

La solución que se propone consiste en comparar la ganancia de cada proyecto A con las ganancias de los demás proyectos B. Cuando se encuentra un proyecto con una ganancia superior a la de A, se compara el prestigio de A con el de B y, si este último también es superior, el proyecto A es rechazado y se estudia el siguiente proyecto. De no tener un prestigio superior se seguirá comparando el proyecto A contra otros proyectos hasta haberlo comparado con todos . Por tanto, un proyecto es aceptable si se compara con todos los demás proyectos y su ganancia y prestigio no son inferiores a los del último proyecto.

La complejidad de es algoritmos es de $O(n^2)$.

1.2. División y Conquista

La solución que se propone comienza por ordenar los proyectos de menor a mayor según sus ganancias. Luego se divide el problema en dos subproblemas, cada uno de estos contienen la mitad de los proyectos del problema.

Llamaremos a uno de estos subproblemas el subproblema izquierdo y al otro de estos el subproblema derecho, haciendo referencia a la porción del arreglo de proyectos que tiene cada uno. El subproblema izquierdo tendrá los proyectos con menores ganancias mientras que el subproblema derecho tendrá los proyectos con mayor ganancias.

Para este problema nos interesa descartar aquellos proyectos que sean menores en ganancias y prestigio contra algún otro proyecto. Al ser un algoritmo recursivo podemos inducir que ninguno de los proyectos en el resultado de un subproblema descarta a otro de ese mismo resultado, por lo que solo habrá que compararlos contra los resultados del otro subproblema. Como sabemos que en el resultado del subproblema derecho todos los proyectos son mayores en ganancias a los proyectos del resultado del subproblema izquierdo, ninguno de los proyectos que nos da el subproblema derecho puede ser descartado por alguno de los del resultado del subproblema izquierdo. De esta misma manera un proyecto del izquierdo solo podrá permanecer si es que tiene un prestigio mayor al mayor prestigio de todos los proyectos. Para obtener todos los proyectos validos, solo habrá que descartar los proyectos del subproblema izquierdo que no superen al mayor prestigio entre los proyectos del resultado del subproblema derecho.

Este algoritmo funciona bien al ser todas las ganancias valores distintos. Como el problema indica, al haber un empate, el proyecto no se descarta. Para resolver este caso, el algoritmo guardará, para cada subproblema, el mayor valor de prestigio

que no pertenezca a un elemento con la menor ganancia. Esto es porque, al estar ordenados, solo los proyectos con menor ganancia del subproblema derecho pueden empatar con los proyectos de mayor ganancia del subproblema derecho.

1.2.1. Estructura de Datos

Para la estructura de datos usaremos una lista modificada que guarde el máximo prestigio agregado, la mínima ganancia y el máximo prestigio que no sea parte de un proyecto con la mínima ganancia. El método para encolar de esta lista se vería de esta forma.

Algorithm 1 Agregar *proyecto*

```
lista agregar proyecto
if proyecto.ganancia > minGan then
  if proyecto.prestigio > MaxPresNoMin then
    MaxPresNoMin ← proyecto.prestigio
  end if
else if proyecto.ganancia < minGan then
  MaxPresNoMin ← MaxPres
  MinGan ← proyecto.ganancia
end if
if proyecto.prestigio > MaxPres then
  MaxPres ← proyecto.prestigio
end if
```

Adicionalmente al inicializar esta lista, MaxPresNoMin y MaxPres son los menores valores posibles mientras que MinGan es el mayor valor posible. También la lista cuenta con metodos para obtener estas variables.

1.2.2. Algoritmo

Tener en cuenta que previo a la primera llamada de SelectInversión se ordenó *proyectos* de menor a mayor según sus ganancias.

1.2.3. Complejidad

El algoritmo presentado divide el problema en dos partes para resolverlo. Con cada una de las dos partes teniendo la mitad de los elementos del problema. Adicionalmente por cada elemento del problema izquierdo hay que agregarlo o no a los elementos del problema derecho, por lo que la complejidad de esta operación es $O(n)$. Sabiendo esto la relación de recurrencia del problema es:

$$T(n) = 2T(n/2) + O(n)$$

Aplicando esta relación de recurrencia al teorema del maestro. Tenemos que $a = 2$, $b = 2$ y $f(n) = O(n)$. Calculamos $log_b(a) = log_2(2) = 1$ y observamos que este problema pertenece al caso 2 del teorema del maestro. Por lo tanto la complejidad temporal del problema será $O(nlog(n))$. Adicionalmente hay que recordar que la

Algorithm 2 SelectInversión *proyectos*

```
n ← proyecto.largo()
if n == 1 then
  return [proyectos[0]]
end if
if n == 0 then
  return []
end if
proyectosIzq ← SelectInversión(proyectos[..n//2])
proyectosDer ← SelectInversión(proyectos[n//2..])
MaxPres ← proyectosDer.MaxPres
MaxPresNoMin ← proyectosDer.MaxPresNoMin
MinGan ← proyectosDer.MinGan
for all proyectoIzq IN proyectosIzq do
  if proyectoIzq.ganancia == MinGan then
    if proyectoIzq.prestigio > MaxPresNoMin then
      proyectosDer.agregar(proyectoIzq)
    end if
  end if
  if proyectoIzq.presigio > MaxPres then
    proyectosDer.agregar(proyectoIzq)
  end if
end for
return proyectosDer
```

operación de ordenamiento del comienzo es de complejidad $O(nlog(n))$ por lo que no modifica la complejidad de la solución de este problema.

También podemos encontrar esta complejidad desarrollando la recurrencia. Digamos que la cantidad de veces que se divide el problema en subproblemas es k. Sabemos que el problema deja de dividirse cuando se tiene 1 elemento por lo que k sería igual a la cantidad de veces que podemos dividir a n hasta obtener 1. Por lo tanto $n = 2^k$, que despejado es $log_2(n) = k$. Habíamos definido la complejidad dentro de cada subproblema como $O(n)$ y concluimos que el problema se subdivide $log_2(n)$ veces podemos concluir que la complejidad temporal del problema es $O(nlog(n))$.

La complejidad espacial se puede razonar de la misma forma ya que el arreglo se divide en dos y cada mitad se copia a otro subproblema. El arreglo de n elementos será copiado la cantidad de veces que se divida el subproblema por lo que la complejidad espacial del problema es de $O(nlog(n))$.

1.2.4. Ejemplos

Ejemplo 1 Se tiene la siguiente lista de proyectos

Proyectos	Proyecto 1	Proyecto 2	Proyecto 3	Proyecto 4	Proyecto 5	Proyecto 6
Ganancia	14	16	35	36	12	55
Prestigio	23	22	44	16	40	46

Como la solución lo propone ordenamos de menor a mayor según sus ganancias:

Proyectos	Proyecto 5	Proyecto 1	Proyecto 2	Proyecto 3	Proyecto 4	Proyecto 6
Ganancia	12	14	16	35	36	55
Prestigio	40	23	22	44	16	46

Dividimos en subproblemas:

Proyectos	Proyecto 5	Proyecto 1	Proyecto 2
Ganancia	12	14	16
Prestigio	40	23	22

Proyectos	Proyecto 3	Proyecto 4	Proyecto 6
Ganancia	35	36	55
Prestigio	44	16	46

Del primer Subproblema por recursión dividimos en subproblemas:

Proyectos	Proyecto 5
Ganancia	12
Prestigio	40

Proyectos	Proyecto 1	Proyecto 2
Ganancia	14	16
Prestigio	23	22

Que a su vez se dividen en:

Proyectos	Proyecto 5
Ganancia	12
Prestigio	40

Ya que en este subproblema se llega a un caso base no se vuelve a dividir

Proyectos	Proyecto 1	Proyecto 2
Ganancia	14	16
Prestigio	23	22

Y en este caso el subproblema se subdivide en:

Proyectos	Proyecto 1
Ganancia	14
Prestigio	23

Proyectos	Proyecto 2
Ganancia	16
Prestigio	22

Donde ambos son un caso base y los llamamos

proyectosIzq	Proyecto 1
Ganancia	14
Prestigio	23

proyectosDer	Proyecto 2
Ganancia	16
Prestigio	22
MaxPres	22
MaxPresNoMin	-inf
MinGan	16

Al recorrer proyectosIzq tenemos el Proyecto 1 cuyo prestigio supera al valor MaxPres de proyectosDer, por lo tanto se le agrega en proyectosDer

proyectosDer	Proyecto 1	Proyecto 2
Ganancia	14	16
Prestigio	23	22
MaxPres	23	
MaxPresNoMin	22	
MinGan	14	

Como se puede observar se actualizaron los valores de MaxPres y MinGan debido a que el proyecto que se agregó tenia mayor prestigio y menor ganancia respectivamente.

Como ya se recorrieron todos los proyectos del lado izquierdo, devolvemos los de lado derecho y volvemos al subproblema padre de este.

proyectosIzq	Proyecto 5
Ganancia	12
Prestigio	40

proyectosDer	Proyecto 1	Proyecto 2
Ganancia	14	16
Prestigio	23	22
MaxPres	23	
MaxPresNoMin	22	
MinGan	14	

Al recorrer proyectosIzq tenemos el Proyecto 5 cuyo prestigio supera al valor MaxPres de proyectosDer, por lo tanto se le agrega en proyectosDer

proyectosDer	Proyecto 5	Proyecto 1	Proyecto 2
Ganancia	12	14	16
Prestigio	40	23	22
MaxPres	40		
MaxPresNoMin	23		
MinGan	12		

Nuevamente se actualizaron los valores de MaxPres, MaxPresNoMin y MinGan debido a que el proyecto que se agregó tenia mayor prestigio y menor ganancia respectivamente.
Como ya se recorrieron todos los proyectos del lado izquierdo, devolvemos los de lado derecho y volvemos al subproblema padre de este.
Recordamos que el segundo subproblema original era

Proyectos	Proyecto 3	Proyecto 4	Proyecto 6
Ganancia	35	36	55
Prestigio	44	16	46

Y que por lo tanto requería ser subdivido en problemas

Proyectos	Proyecto 3
Ganancia	35
Prestigio	44

Ya que en este subproblema se llega a un caso base no se vuelve a dividir

Proyectos	Proyecto 4	Proyecto 6
Ganancia	36	55
Prestigio	16	46

Y en este caso el subproblema se subdivide en:

Proyectos	Proyecto 4
Ganancia	36
Prestigio	16

Proyectos	Proyecto 6
Ganancia	55
Prestigio	46

Donde ambos son un caso base y los llamamos

proyectosIzq	Proyecto 4
Ganancia	36
Prestigio	16

proyectosDer	Proyecto 6
Ganancia	55
Prestigio	46
MaxPres	46
MaxPresNoMin	-inf
MinGan	55

Al recorrer proyectosIzq notamos que ninguno cuenta con un prestigio mayor al de proyectosDer y por lo tanto devolvemos proyectosDer así como está. Volvemos al subproblema

proyectosIzq	Proyecto 3
Ganancia	35
Prestigio	44

proyectosDer	Proyecto 6
Ganancia	55
Prestigio	46
MaxPres	46
MaxPresNoMin	-inf
MinGan	55

Nuevamente vemos que los proyectos de la izquierda no pueden superar el prestigio de los de la derecha y entonces devolvemos nuevamente los de la derecha sin modificar. Quedándonos al llegar al problema padre:

proyectosIzq	Proyecto 5	Proyecto 1	Proyecto 2
Ganancia	12	14	16
Prestigio	40	23	22

proyectosDer	Proyecto 6
Ganancia	55
Prestigio	46
MaxPres	46
MaxPresNoMin	-inf
MinGan	55

Donde al recorrer los prestigios de todos los del grupo de la izquierda ninguno supera al del Proyecto 6 de la derecha. Devolvemos proyectosDer y la solución encontrada es:

Proyectos Preseleccionados	Proyecto 6
Ganancia	55
Prestigio	46

Ejemplo 2 Se tiene la siguiente lista de proyectos

Proyectos	Proyecto 1	Proyecto 2	Proyecto 3	Proyecto 4	Proyecto 5	Proyecto 6
Ganancia	80	73	73	52	52	94
Prestigio	11	70	55	20	75	90

Como la solución lo propone ordenamos de menor a mayor según sus ganancias:

Proyectos	Proyecto 5	Proyecto 4	Proyecto 2	Proyecto 3	Proyecto 1	Proyecto 6
Ganancia	52	52	73	73	80	94
Prestigio	20	75	70	55	11	90

Dividimos en subproblemas:

Proyectos	Proyecto 5	Proyecto 4	Proyecto 2
Ganancia	52	52	73
Prestigio	20	75	70

Proyectos	Proyecto 3	Proyecto 1	Proyecto 6
Ganancia	73	80	94
Prestigio	55	11	90

Del primer Subproblema por recursión dividimos en subproblemas:

Proyectos	Proyecto 5
Ganancia	52
Prestigio	20

Proyectos	Proyecto 4	Proyecto 2
Ganancia	52	73
Prestigio	75	70

Que a su vez se dividen en:

Proyectos	Proyecto 5
Ganancia	52
Prestigio	20

Ya que en este subproblema se llega a un caso base no se vuelve a dividir

Proyectos	Proyecto 4	Proyecto 2
Ganancia	52	73
Prestigio	75	70

Y en este caso el subproblema se subdivide en:

Proyectos	Proyecto 4
Ganancia	52
Prestigio	75

Proyectos	Proyecto 2
Ganancia	73
Prestigio	70

Donde ambos son un caso base y los llamamos

proyectosIzq	Proyecto 4
Ganancia	52
Prestigio	75

proyectosDer	Proyecto 2
Ganancia	73
Prestigio	70
MaxPres	70
MaxPresNoMin	-inf
MinGan	73

Al recorrer proyectosIzq tenemos el Proyecto 4 cuyo prestigio supera al valor MaxPres de proyectosDer, por lo tanto se le agrega en proyectosDer

proyectosDer	Proyecto 4	Proyecto 2
Ganancia	52	73
Prestigio	75	70
MaxPres	75	
MaxPresNoMin	70	
MinGan	52	

Como se puede observar se actualizaron los valores de MaxPres y MinGan debido a que el proyecto que se agregó tenia mayor prestigió y menor ganancia respectivamente.

Como ya se recorrieron todos los proyectos del lado izquierdo, devolvemos los de lado derecho y volvemos al subproblema padre de este.

proyectosIzq	Proyecto 5
Ganancia	52
Prestigio	20

proyectosDer	Proyecto 4	Proyecto 2
Ganancia	52	73
Prestigio	75	70
MaxPres	75	
MaxPresNoMin	70	
MinGan	52	

Al recorrer proyectosIzq tenemos el Proyecto 5, que tiene la misma ganancia que MinGan pero cuyo prestigio no supera al valor MaxPresNoMin de proyectosDer, por lo tanto no se agrega en proyectosDer

proyectosDer	Proyecto 4	Proyecto 2
Ganancia	52	73
Prestigio	75	70
MaxPres	75	
MaxPresNoMin	70	
MinGan	52	

Como ya se recorrieron todos los proyectos del lado izquierdo, devolvemos los de lado derecho y volvemos al subproblema padre de este.
Recordamos que el segundo subproblema original era

Proyectos	Proyecto 3	Proyecto 1	Proyecto 6
Ganancia	73	80	94
Prestigio	55	11	90

Y que por lo tanto requería ser subdivido en problemas

Proyectos	Proyecto 3
Ganancia	73
Prestigio	55

Ya que en este subproblema se llega a un caso base no se vuelve a dividir

Proyectos	Proyecto 1	Proyecto 6
Ganancia	80	94
Prestigio	11	90

Y en este caso el subproblema se subdivide en:

Proyectos	Proyecto 1
Ganancia	80
Prestigio	11

Proyectos	Proyecto 6
Ganancia	94
Prestigio	90

Donde ambos son un caso base y los llamamos

proyectosIzq	Proyecto 1
Ganancia	80
Prestigio	11

proyectosDer	Proyecto 6
Ganancia	94
Prestigio	90
MaxPres	90
MaxPresNoMin	-inf
MinGan	94

Al recorrer proyectosIzq notamos que ninguno cuenta con un prestigio mayor al de proyectosDer y por lo tanto devolvemos proyectosDer así como está. Volvemos al subproblema

proyectosIzq	Proyecto 3
Ganancia	73
Prestigio	55

proyectosDer	Proyecto 6
Ganancia	94
Prestigio	90
MaxPres	90
MaxPresNoMin	-inf
MinGan	94

Nuevamente vemos que los proyectos de la izquierda no pueden superar el prestigio de los de la derecha y entonces devolvemos nuevamente los de la derecha sin modificar. Quedándonos al llegar al problema padre:

proyectosIzq	Proyecto 4	Proyecto 2
Ganancia	52	73
Prestigio	75	70

proyectosDer	Proyecto 6
Ganancia	94
Prestigio	90
MaxPres	90
MaxPresNoMin	-inf
MinGan	94

Donde al recorrer los prestigios de todos los del grupo de la izquierda ninguno supera al del Proyecto 6 de la derecha. Devolvemos proyectosDer y la solución encontrada es:

Proyectos Preseleccionados	Proyecto 6
Ganancia	94
Prestigio	90

2. Triangulación de polígonos

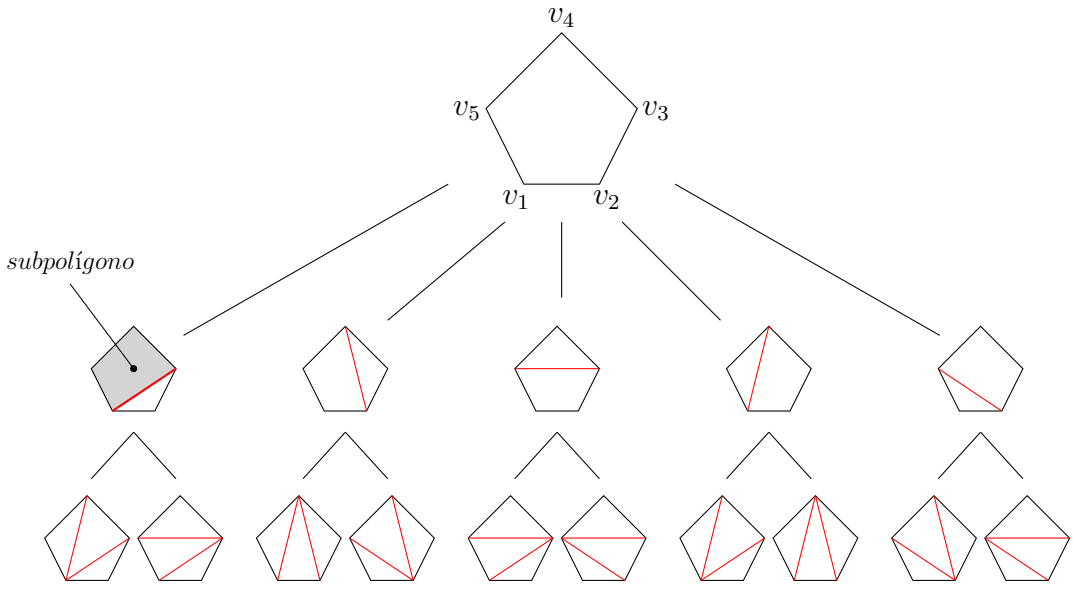
2.1. Fuerza bruta

Este problema puede resolverse fácilmente por fuerza bruta. El algoritmo para resolverlo consiste en formar todas las posibles triangulaciones del polígono. Luego para cada triangulación calculamos la suma de las longitudes de los lados de cada triángulo construido. Por último nos quedamos con la triangulación que minimice dicha suma.

Supongamos que tenemos los vértices del polígono enumerados de 1 a n. Para formar todas las posibles triangulaciones lo que hacemos es iterar sobre cada vértice del polígono. Para cada vértice v_i, lo unimos mediante una cuerda con el vértice v_{i+2}. Esto nos genera un triángulo de vértices v_i, v_{i+1} y v_{i+2} y a su vez nos divide el polígono en ese triángulo y un "subpolígono". Luego nos guardamos el triángulo y volvemos a repetir lo mismo recursivamente pero con el subpolígono resultante, hasta llegar a un polígono de 3 vértices lo cual sería el caso base.

2.1.1. Ejemplo

Supongamos que queremos triangular un polígono de 5 vértices $\{v_1, v_2, v_3, v_4, v_5\}$. En este caso el durante la ejecución del algoritmo por fuerza bruta se va a creando el siguiente árbol de recursión, en el cual cada nodo es un subproblema y las soluciones se encuentran en el último nivel del árbol



Tener en cuenta que los subproblemas del segundo nivel del árbol tienen a su vez 4 subproblemas (hijos), sin embargo en este caso mostramos solo 2 de esos subproblemas pues los otros 2 restantes son repeticiones de los que se muestran en la figura. Podemos ver que al agregar una cuerda al polígono, este se divide en un triángulo y un subpolígono. Al agregar la cuerda se calcula la suma de la longitud de los lados cada triángulo construido. Una vez formadas todas las triangulaciones, el algoritmo retorna aquella que tiene la menor suma total de los lados de cada

triángulo.

2.1.2. Complejidad

Podemos observar que durante la ejecución del algoritmo para cada uno de los n vértices se recorren otros $n-1$ vértices del subpolígono resultante. De la misma forma para cada uno de los $n-1$ vértices del subpolígono se recorren $n-2$, y así sucesivamente hasta llegar a 3 vértices. En cada iteración se realizan operaciones $O(1)$ como sumar las longitudes del triángulo construido y agregar el triángulo a la triangulación, entonces la complejidad de esta parte es

$$O(n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 4) = O(n!/6) = O(n!)$$

Una vez que están construidas las triangulaciones el algoritmo recorre cada una de ellas y devuelve la que tenga menor suma. Del árbol de recursion vemos que se construyen $n!/6$ triangulaciones, con lo cual hallar la de menor suma tiene un costo $O(n!/6) = O(n!)$. Por lo tanto la complejidad temporal es $O(n!)$

Para calcular la complejidad espacial, tenemos que tener en cuenta que con este algoritmo formamos $n!/6$ triangulaciones (muchas están repetidas) que hay que almacenar. Cada triangulación contiene $n-2$ triángulos y cada triángulo contiene 3 vértices, por lo tanto la complejidad espacial será

$$O(3(n-2)(n!/6)) = O(n!)$$

2.2. Programación dinámica

A partir del árbol de recursión del algoritmo por fuerza bruta podemos ver que hay subproblemas (subpolígonos) ya resueltos que vuelven a aparecer, es decir, este problema tiene la propiedad de subproblemas superpuestos. Por lo tanto podemos diseñar un algoritmo que utilice programación dinámica para resolver el problema, almacenando los resultados parciales para no volver a calcularlos.

2.2.1. Relación de recurrencia

Dentro del árbol de recursión cada nodo es un subproblema, cuyos subproblemas consisten en elegir el siguiente triángulo a construir de forma que se minimice la suma total de los lados de los triángulos. Cada subproblema se caracteriza por el conjunto de vértices V del subpolígono y el conjunto de triángulos ya construidos T. Para simplificar la notación llamaremos *peso* a la suma de las longitudes de los lados de un triángulo. Sea t_i un triángulo, entonces denotamos su peso como $w(t_i)$. Denotamos $\text{OPT}(V, T)$ al mínimo de los pesos de las triangulaciones que se pueden formar a partir del polígono V con un conjunto T de triángulos ya construidos. De esta forma cada subproblema queda expresado por la recurrencia

$$\text{OPT}(V,T) = \begin{cases} \min\limits_{1 \le i \le n} (w(t_i) + \text{OPT}(V - \{v_{i+1}\}, T \cup \{t_i\})), & \text{si } |V| > 3 \\ w(V), & \text{si } |V| = 3 \end{cases}$$

donde $|V| = 3$ es el caso base y por lo tanto el óptimo en este caso es el peso del triángulo formado por los vértices de V.

2.2.2. Pseudocódigo

```
Triangular(polígono)
  Sea n la cantidad de vértices del polígono
  Sea triangulaciones[i] el conjunto de triangulaciones de i+1 triángulos
  Desde i = 0 hasta n-1
    Sea t el triángulo de vértices i, i+1 y i+2
    Sea peso[t] el peso del triángulo t
    Sea S el subpolígono {polígono} - {polígono[i+1]}
    Agregar (t, peso, S) a triangulaciones[0]

  Desde i = 1 hasta n-3
    Para cada triangulación (T, peso, S) en triangulaciones[i-1]
      Sea m la cantidad de vértices del subpolígono S
      Desde j = 1 hasta m
        Sea t el tríangulo de vértices j, j+1, j+2 en S
        Si t es compatible con la triangulación T
          Si (T + {t}) no pertenece a triangulaciones[i]
            triangulaciones[i] += (T + {t}, peso + peso[t], S - {j+1})

  Retornar la triangulacion con menor peso en triangulaciones[n-3]
```

2.2.3. Explicación del algoritmo

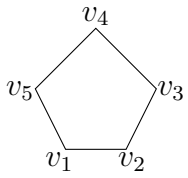
El algoritmo por programación dinámica consiste en ir construyendo las triangulaciones de forma iterativa agregando un triángulo a la vez. En una primera iteración construimos n "triangulaciones parciales" formadas cada una por un solo triángulo y su correspondiente peso. Luego a cada una de estas triangulaciones le agregamos cada uno de los triángulos con los cuales es compatible (actualizando el peso), formando triangulaciones de ahora 2 triángulos, y así sucesivamente hasta llegar a triangulaciones de $n-2$ triángulos. En cada iteración corroboramos que la triangulacion a agregar no haya sido construida previamente, de esta forma estamos memorizando los subproblemas que se repiten. Una vez que tenemos todas las triangulaciones formadas nos quedamos con aquellas que tengan $n-2$ triangulos, es decir, las que están completas. Entre las triangulaciones completas nos quedamos con la que tenga menor peso.

2.2.4. Estructuras de datos

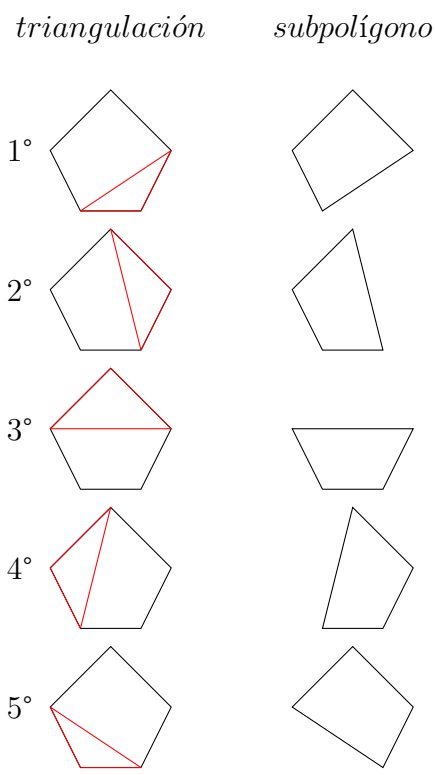
Las estructuras de datos que utiliza este algoritmo son principalmente diccionarios y sets de python. El diccionario nos sirve para guardar las triangulaciones (y sus correspondientes pesos) y poder acceder a ellas eficientemente, por ejemplo para comprobar si una triangulación ya fue construida y de esta forma no repetir cálculos. Por otro lado para cada triangulación utilizamos un set en el cual almacenamos el subpolígono de esa triangulación, es decir, la porción del polígono que aún no fue triangulada. A partir de este subpolígono podemos hallar los triángulos compatibles con los triángulos ya construidos.

2.2.5. Ejemplo

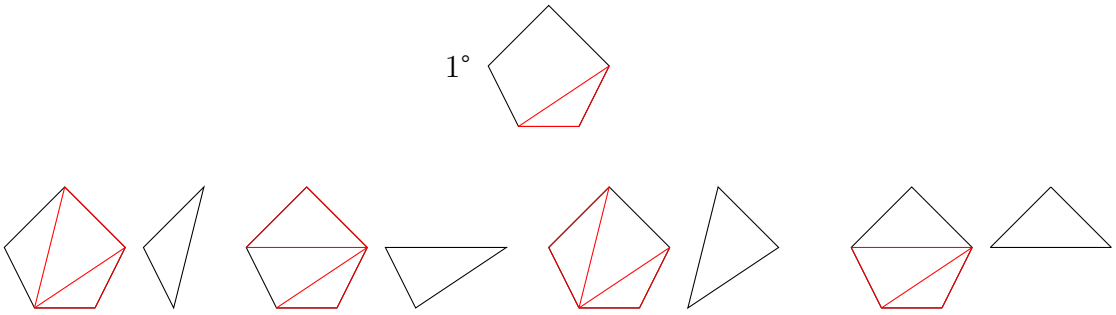
Supongamos ahora que queremos triangular un polígono de 5 vértices mediante el algoritmo propuesto anteriormente. Inicialmente tenemos el polígono sin triangular



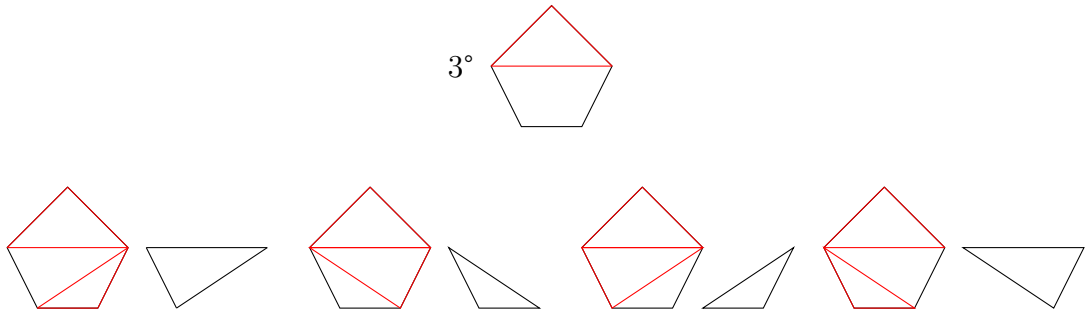
En una primera iteración el algoritmo forma todas las "triangulaciones parciales" de 1 triángulo formado por 3 vértices contiguos, junto con sus subpolígonos



Luego para cada una de las triangulaciones parciales se le agrega un triángulo formado por 3 vértices contiguos del subpolígono, junto con el nuevo subpolígono. Por ejemplo por la primera de todas se agregan 4 triangulaciones pues dentro del subpolígono tenemos 4 triángulos posibles



Análogamente para la segunda triangulación de 1 triángulo se agregan 4 triangulaciones. Sin embargo para la tercer triangulación de 1 triángulo se agregarán solo 3 pues vemos que una de las 4 posibles triangulaciones ya fue agregada previamente



ya fue construida

De la misma forma, hay varias triangulaciones parciales que se repiten durante la ejecución del algoritmo, sin embargo solo se agregan aquellas que no están repetidas.

Una vez que están formadas todas las triangulaciones de 2 triángulos, por cada una de ellas se agrega solo una triangulación completa, pues como el subpolígono es un triángulo éste contiene un único triángulo a agregar. El número de triangulaciones completas para un polígono de n vértices está dado por

$$\frac{1}{n-1}\binom{2n-4}{n-2}$$

En este caso para un polígono de 5 vértices ($n = 5$) resultan 14 triangulaciones completas. De entre las 14 se elige aquella con menor peso.

2.2.6. Complejidad

Para calcular la complejidad temporal tenemos en cuenta que para cada vértice inicializamos una triangulación de un triángulo, lo cual tiene un costo de $O(n)$. Para cada una de las triangulaciones de un triángulo se crean $n-1$ triangulaciones de 2 triangulos, resultando en un total de $n(n-1)$ triangulaciones de 2 triángulos, lo cual cuesta $O(n(n-1)) = O(n^2)$. Se repite este proceso hasta llegar a una triangulación de $n-2$ triángulos. En cada iteración se revisa si el triángulo ya fue construido ($O(1)$) y se crea el subpolígono correspondiente $O(n-1)$. Juntando todo llegamos a una ecuación de la forma

$$O(n) + O(n(n-1)) + O(n(n-1)(n-2)) + ... + O(n(n-1)(n-2)...(2)) \approx O(n!)$$

Con lo cual se concluye que la complejidad temporal del algoritmo es $O(n!)$ lo cual no muestra una mejora en cuanto a la solución por fuerza bruta.

Por otro lado, durante la ejecución del algoritmo se almacenan todas las triangulaciones parciales junto con sus subpolígonos. En el árbol de recursión podemos

ver que en el segundo nivel tenemos n triangulaciones parciales, en el tercer nivel tenemos $n(n-1)$ y así sucesivamente, lo cual nos lleva a pensar que almacenamos $n!$ triangulaciones parciales, sin embargo hay muchas que se repiten las cuales son ignoradas por el algoritmo. Como no sabemos cuántas triangulaciones parciales se almacenan por nivel, $O(n!)$ funciona para acotarla superiormente, aunque creemos que existe una mejor cota.

2.2.7. Ejecución de la solución

Para ejecutar el programa simplemente se debe correr `python3 main.py poligono.txt` por línea de comandos.

3. Teoría

3.1. Descripción y propiedades de las metodologías presentadas en el trabajo

Los algoritmos de tipo **Greedy** se utilizan como metodología de resolución de problemas de optimización, ya sea para maximizar o minimizar una variable. La propuesta es dividir el problema principal en subproblemas, y que estos últimos tengan una jerarquía entre ellos, para ir resolviendo cada subproblema de manera iterativa, tomando la mejor decisión para la situación local. A medida que se resuelven los subproblemas de acuerdo a la heurística (procedimiento) elegida, se habilitan nuevos subproblemas a resolver.

Para que la solución de un algoritmo greedy sea óptima, el problema a resolver debe cumplir ciertas propiedades:

- Subestructura óptima: La solución global al problema contiene dentro de su interior las soluciones óptimas de sus subproblemas.
- Elección greedy: Se elige la mejor solución local en cada paso para acercarse a la solución óptima global.

La metodología **División y Conquista** trata sobre dividir el problema principal que se presenta en menores subproblemas que puedan resolverse independientemente entre si. Cada uno de estos subproblemas conserva la naturaleza y características del problema original. Entonces se produce un proceso recursivo donde cada uno de estos subproblemas generados se divide hasta llegar a un caso base donde el problema tenga un tamaño apto para que pueda resolverse de manera trivial. De esta forma, en la çonquista", se van resolviendo los subproblemas hasta tener todas las soluciones y finalmente son combinadas para obtener una solución global al problema con el que se comenzó.

Al igual que la metodología Greedy, no todo los problemas pueden ser resueltos mediante División y Conquista. Otra propiedad que comparten es la de subestructura óptima. Además para que este tipo de algoritmos sea eficiente, es conveniente que la resolución y combinación de las soluciones de los subproblemas se realice de forma eficaz para que la complejidad sea razonable al utilizarlo en la práctica.

Por otro lado en el caso de **Programación Dinámica** al igual que los algoritmos Greedy, es una estrategia que resuelve problemas de optimización donde se necesite minimizar o maximizar algún resultado. Utiliza la división del problema en subproblemas de menor tamaño para solucionarlos de manera recursiva, donde en general se comeinza por resolver los problemas más pequeños, y las soluciones pueden ser reutilizadas en otros subproblemas de mayor tamaño. Esta propiedad es llamada memorización, que es una técnica en la que se almacenan los resultados en, por ejemplo, una tabla, para su posterior uso. Esto permite evitar hacer cálculos innecesarios para mejorar la eficiencia del algoritmo en cuestión.

En este caso las propiedades que debe cumplir el problema para ser resuelto mediante esta metodología es la de subestructura óptima, y la de subproblemas

superpuestos. Esto último significa que a la hora de resolver un problema, exista la posibilidad de reutilizar los resultados que se almacenaron al resolver ciertos subproblemas (de caso base). En otras palabras que se compartan resultados.

3.2. Algoritmo Gale-Shapley y metodologías vistas

El algoritmo de Gale-Shapley plantea un problema en el cual hay grupos de solicitantes y requeridos, cada uno con una lista de preferencias sobre el otro conjunto. Y el objetivo es emparejar a los participantes de manera que no haya parejas inestables.

A continuación analizaremos si pertenece a alguna de las metodologías comentadas anteriormente.

- **Greedy:** En cada paso del algoritmo, se realizan asignaciones basadas en preferencias. Estas elecciones que se realizan en cada paso no garantizan que el resultado final sea óptimo en términos globales, porque el resultado depende del orden en que se realizan las solicitudes y aceptaciones, y diferentes ordenamientos pueden llevar a diferentes emparejamientos estables. Aunque el algoritmo tiene alguna característica greedy, como por ejemplo que los elementos eligen las opciones que mejor les parecen en cada paso, no se ajusta a la definición, y por lo tanto no se lo puede considerar parte de esta metodología.

- **División y Conquista:** El algoritmo de Gale-Shapley no divide el problema en subproblemas más pequeños para resolverlos de forma recursiva, si no que opera de forma iterativa en varios pasos de propuestas y aceptaciones, por lo que tampoco podemos decir que utilice este tipo de estrategia.

- **Programación Dinámica:** Como se mencionó en el caso anterior, tampoco se divide el problema en subproblemas más pequeños y además no se basa en la resolución de subproblemas solapados, porque los resultados parciales de cada iteración no son reutilizados en ningún momento.

Por lo tanto podemos concluir que el algoritmo Gale-Shapley no pertenece a ninguna de las metodologías anteriormente mencionadas. Utiliza una estrategia propia de propuestas y aceptaciones.

Segunda entrega

1. Preselección de inversiones

1.1. División y Conquista

1.1.1. Complejidad

El algoritmo presentado divide el problema en dos partes para resolverlo. Con cada una de las dos partes teniendo la mitad de los elementos del problema. Adicionalmente por cada elemento del problema izquierdo hay que agregarlo o no a los elementos del problema derecho, por lo que la complejidad de esta operación es $O(n)$. Sabiendo esto la relación de recurrencia del problema es:

$$T(n) = 2T(n/2) + O(n)$$

Aplicando esta relación de recurrencia al teorema del maestro. Tenemos que $a = 2$, $b = 2$ y $f(n) = O(n)$. Calculamos $log_b(a) = log_2(2) = 1$.

Dado que el segundo caso del teorema maestro $f(n) = O(n^c log^k(n))$ se aplica cuando $log_b(a) = c$ y luego $T(n) = O(n^c log^{k+1}(n))$. observamos que para este problema $f(n) = O(n^c log^k(n)) = O(n)$ si $c = 1$ y $k = 0$ y $log_b(a) = 1 = c$ por lo tanto pertenece al caso 2 del teorema del maestro y la complejidad temporal del problema será $T(n) = O(n^1 log^{0+1}(n)) = O(nlog(n))$.

Adicionalmente hay que recordar que la operación de ordenamiento del comienzo es de complejidad $O(nlog(n))$ por lo que no modifica la complejidad de la solución de este problema.

También podemos encontrar esta complejidad desarrollando la recurrencia. Digamos que la cantidad de veces que se divide el problema en subproblemas es k. Sabemos que el problema deja de dividirse cuando se tiene 1 elemento por lo que k sería igual a la cantidad de veces que podemos dividir a n hasta obtener 1. Por lo tanto $n = 2^k$, que despejado es $log_2(n) = k$. Habíamos definido la complejidad dentro de cada subproblema como $O(n)$ y concluimos que el problema se subdivide $log_2(n)$ veces podemos concluir que la complejidad temporal del problema es $O(nlog(n))$.

La complejidad espacial se puede razonar de la misma forma ya que el arreglo se divide en dos y cada mitad se copia a otro subproblema. El arreglo de n elementos será copiado la cantidad de veces que se divida el subproblema por lo que la complejidad espacial del problema es de $O(nlog(n))$.

2. Triangulación de polígonos

2.1. Fuerza bruta

2.1.1. Pseudocódigo

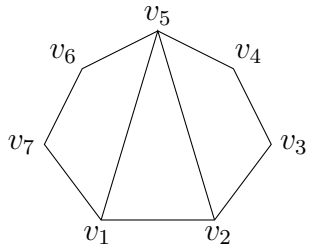
```
triangular(poligono, n):
  Si n == 3:
    retornar calcular_peso(poligono[0], poligono[1], poligono[2])

  peso_minimo = infinito
  Desde i = 1 hasta n
    subpoligono = poligono - {poligono[i+1]}
    peso = triangular(subpoligono, n-1) +
           + calcular_peso(poligono[i], poligono[(i+1)], poligono[(i+2)])
    Si peso < peso_minimo:
      peso_minimo = peso
  Retornar peso_minimo
```

2.2. Programación dinámica

2.2.1. Relación de recurrencia

Según observamos en el árbol de recursión del algoritmo por fuerza bruta, hay ciertas triangulaciones parciales que se repiten a lo largo del cálculo de una triangulación. Tenemos que hallar una forma de memorizar dichos resultados.

Podemos pensar que una triangulación de un polígono de n vértices puede descomponerse en un triángulo y los 2 subpolígonos a ambos lados del triángulo, por ejemplo para un polígono P de 7 vértices



vemos que si tenemos calculadas todas las triangulaciones de los subpolígonos de 4 vértices del polígono original, entonces para calcular la triangulación completa podemos hacerlo hallando el peso mínimo de la suma de dos triangulaciones de 4 vértices y un triángulo.

En tal caso definimos el subproblema $\text{OPT}(i, j)$ como el menor peso para la triangulación de vértices de $\{v_i, v_{i+1}, ..., v_j\}$ (recorriendo los vértices en sentido antihorario), cuya relación de recurrencia está dada por

$$\text{OPT}(i,j) = \begin{cases} \min\limits_{i<k<j} (\text{OPT}(i,k) + \text{OPT}(k,j) + w(i,j,k)), & \text{si } j \geq i+2 \\ 0, & \text{si } j \leq i \end{cases}$$

es decir, el menor peso de la triangulación de vértices $\{v_i, v_{i+1}, ..., v_j\}$ se puede calcular como la suma de los pesos de

- la triangulación del polígono de vértices $\{v_i, v_{i+1}, ..., v_k\}$
- la triangulación del polígono de vértices $\{v_k, v_{k+1}, ..., v_j\}$
- el triángulo de vértices $\{v_i, v_j, v_k\}$

hallando el valor de k que minimice dicha suma.

2.2.2. Pseudocódigo

```
triangular(polígono)
  Sea n la cantidad de vértices del polígono
  Sea OPT una matriz de n*n
  Sea triangulacion una matriz de n*n

  Desde i = 0 hasta n - 1
    OPT[i][i] = 0
    triangulacion[i][i] = {}
    Si i < n - 1
      OPT[i][i+1] = 0
      triangulacion[i][i+1] = {}

  Desde tamaño = 2 hasta n - 1
    Desde i = 0 hasta n - tamaño - 1
      j = i + tamaño
      OPT[i][j] = infinito
      Desde k = i + 1 hasta i + j - 1
        peso = OPT[i][k] + OPT[k][j] + calcular_peso(poligono, i, j, k)
        Si peso < OPT[i][j]
            OPT[i][j] = peso
            triangulacion[i][j] = triangulacion[i][k] + triangulacion[k][j] +
                                  + {poligono[i], poligono[j], poligono[k]}

  Retornar OPT[0][n-1], triangulacion[0][n-1]
```

2.2.3. Estructuras de datos

Durante la ejecución del algoritmo se utilizan 2 matrices de $n \times n$ como estructura de datos. En una de ellas en cada posición (i, j) guardamos el menor peso de la triangulación del polígono de vértices $\{v_i, v_{i+1}, ..., v_j\}$, mientras que en la otra almacenamos los triángulos que forman dicha triangulación como listas de tuplas.

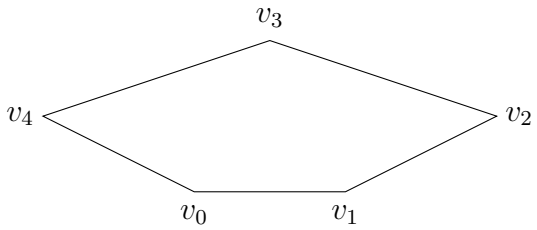
2.2.4. Explicación del algoritmo

El algoritmo consiste en ir calculando iterativamente los pesos de las triangulaciones de los subpolígonos del polígono que queremos triangular. Primero comienza

con subpolígonos de 3 vértices, a partir de los cuales calculamos los pesos de las triangulaciones de subpolígonos de 4 vértices y así sucesivamente hasta llegar a n vértices. En cada iteración, además de calcular los pesos, vamos almacenando las triangulaciones parciales. Para elegir una triangulación para un subpolígono de vértices v_i a v_j el algoritmo calcula los pesos de la siguiente forma: Para cada vértice v_k entre v_i y v_j calcula el peso de las triangulaciones de los subpolígonos de v_i a v_k y de v_k a v_j y a esto le suma el peso del triángulo v_i, v_k, v_j. Luego nos quedamos con el menor de los pesos y el vértice v_k correspondiente. La triangulación del subpolígono de v_i a v_j esta formada por la unión de la triangulación de v_i a v_k, la de v_k a v_j y el triángulo v_i, v_k, v_j.

2.2.5. Ejemplo

Supongamos que queremos triangular el polígono P de vértices $\{v_0 = (2,0)$, $v_1 = (4,0)$, $v_2 = (6,1)$, $v_3 = (3,2)$, $v_4 = (0,1)\}$



El primer paso es inicializar las matrices y llenar los casilleros correspondientes con 0. En este caso la matriz de pesos tiene la forma

	0	1	2	3	4
0	0	0	–	–	–
1	–	0	0	–	–
2	–	–	0	0	–
3	–	–	–	0	0
4	–	–	–	–	0

Luego se comienzan a construir las triangulaciones. Empezamos por las triangulaciones de 3 vértices. La primera es la del subpolígono de vértices v_0 a v_2, es decir, el triángulo v_0, v_1, v_2 cuyo peso es aproximadamente 8.36. La siguiente triangulación es la del subpolígono de vértices v_1 a v_3 cuyo peso es 7.63. De esta forma vamos calculando los pesos de todas las triangulaciones de 3 vértices, con lo cual la matriz queda de la siguiente forma

	0	1	2	3	4
0	0	0	8,36	–	–
1	–	0	0	7,63	–
2	–	–	0	0	12,32
3	–	–	–	0	0
4	–	–	–	–	0

El siguiente paso es calcular las triangulaciones de los subpolígonos de 4 vértices. Comenzamos por la del subpolígono de vértices v_0 a v_3. Según el algoritmo

el peso se calcula como el mínimo para cada k de 1 a 2, de la suma del peso del subpolígono de v_0 a v_k más el peso del subpolígono de v_k a v_3 más el peso del triángulo v_0, v_k, v_3.

Probamos primero con $k = 1$. En este caso tenemos que el peso de v_0 a v_1 es OPT(0,1)=0, el peso de v_1 a v_3 es OPT$(1, 3) = 7{,}63$ y el peso del triángulo v_0, v_1, v_3 requiere un cálculo aparte y es 6,47. Sumando todo tenemos un peso total de 14,1

Repetimos lo mismo con $k = 2$, obteniendo un peso de total de 17,88. Por lo tanto el peso mínimo para la triangulación del subpolígono de vértices de v_0 a v_3 es $min(14{,}1; 17{,}88) = 14{,}1$. De la misma forma calculamos el peso mínimo para los demás subpolígonos de 4 vértices (es decir para el de vértices de v_1 a v_4. Luego la matriz queda de la forma

	0	1	2	3	4
0	0	0	8,36	14,1	–
1	–	0	0	7,63	17,15
2	–	–	0	0	12,32
3	–	–	–	0	0
4	–	–	–	–	0

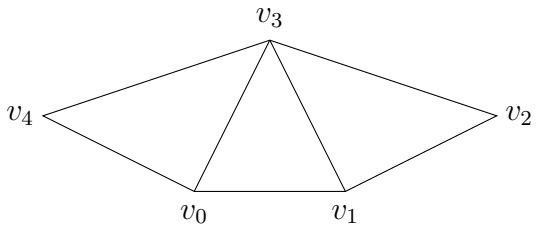
Por último queda calcular el peso mínimo de la triangulación del polígono completo, es decir, el de vértices v_0 a v_4. Análogamente a como hicimos para el caso de 4 vértices, debemos calcular para cada k de 1 a 3 el mínimo de la suma del peso del subpolígono v_0 a v_k más el peso del subpolígono de v_k a v_4 más el peso del triángulo de vértices v_0, v_k, v_4.

Comenzamos con $k = 1$. En este caso el peso de v_0 a v_1 es OPT$(0, 1) = 0$, el de v_1 a v_4 es OPT$(1, 4) = 17{,}15$ y el peso del triángulo v_0, v_1, v_4 es 8,36. Sumando todo nos queda un peso de 25,51.

Continuamos con $k = 2$. El peso de v_0 a v_2 es OPT$(0, 2) = 8{,}36$, el peso de v_2 a v_4 es OPT$(2, 4) = 12{,}32$ y el peso del triángulo v_0, v_2, v_4 es 12,36. En total tenemos un peso de 33,04.

Por último usamos $k = 3$. El peso de v_0 a v_3 es OPT$(0, 3) = 14{,}1$, el peso de v_3 a v_4 es OPT$(3, 4) = 0$ y el peso del triángulo v_0, v_3, v_4 es 7,63. En total nos queda un peso de 21,73.

Luego, el menor peso para la triangulación del polígono de vértices de v_0 a v_4 es 21,73. Por otro lado, si en cada iteración vamos almacenando los triángulos que componen cada triangulación parcial podemos reconstruir la triangulación final. En este caso la triangulación nos queda de la siguiente forma



2.2.6. Complejidad

Para calcular la complejidad temporal analizamos cada parte del algoritmo por separado.

- Inicializar las matrices de tamaño $n \times n$ tiene un costo de $O(n^2)$.
- Setear en 0 las posiciones de la matriz de pesos que no corresponden a un subpolígono cuesta $O(n)$.
- Completar la matriz de pesos requiere hallar el peso mínimo para cada subpolígono. En el peor de los casos, para hallar el peso mínimo debemos probar $n-2$ combinaciones, es decir, tenemos un costo de $O(n)$ (cuando el subpolígono tiene n vértices). Dado que debemos llenar aproximadamente n^2 casilleros, esta parte tiene un costo total de $O(n^3)$.
- Completar la matriz de triangulaciones implica llenar n^2 casilleros con listas de triángulos. En el peor de los casos crear cada lista cuesta $O(n)$ (cuando tenemos $n-2$ triángulos). Por lo tanto como realizamos n^2 veces una operación $O(n)$ el costo de llenar la matriz de triangulaciones es $O(n^3)$.

Sumando los costos de cada parte se concluye que la complejidad total es $O(n^3)$.

Por otro lado para calcular la complejidad espacial vemos que tenemos 2 matrices de $n \times n$, una para almacenar los pesos y otra para las triangulaciones. La matriz de pesos almacena n^2 números, mientras que la de triangulaciones almacena n^2 listas de tuplas. En el peor de los casos cada lista tiene $n-2$ triángulos, cada uno formado por 3 vértices, es decir, un total de $3(n-2)$ vértices. Al tener n^2 listas, esto tiene una complejidad espacial $O(n^2(3(n-2))) = O(n^3)$. Por lo tanto la complejidad espacial es $O(n^3)$.

2.2.7. Ejecución de la solución

Para ejecutar el programa se debe correr `python3 main.py poligono.txt` por línea de comandos.

2.2.8. Complejidad del programa

Para calcular la complejidad del programa podemos ver que la solución en python es prácticamente igual al pseudocódigo. Inicializar las matrices en python puede hacerse en $O(n^2)$ pues si representamos la matriz como una lista de listas, simplemente tenemos que llenar cada una de las n listas con n elementos. Por otro lado, para llenar la matriz de pesos debemos hacer los cálculos correspondientes y una vez que se tiene el peso mínimo guardarlo en la matriz. Todo esto puede realizarse manteniendo la complejidad teórica pues hallar el peso mínimo es $O(n)$ y guardarlo en la matriz es $O(1)$. Por último, completar la matriz de triangulaciones requiere almacenar las listas de triángulos. Esto puede hacerse en python cumpliendo la complejidad ya que unir dos listas de tamaño n y m puede hacerse en $O(n+m)$ simplemente recorriendo los elementos de cada lista y guardándolos en otra. Por lo tanto concluímos que nuestro programa mantiene la complejidad de la propuesta teórica.