

1) para realizar esto primero vamos a simplificar un poco al determinante

$$\begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix}$$

para esto le restamos la segunda fila a la tercera columna, y luego le restamos la primera fila a la segunda columna,

obteniendo

$$\begin{vmatrix} 1 & p_x & p_y \\ 0 & q_x - p_x & q_y - p_y \\ 0 & r_x - q_x & r_y - q_y \end{vmatrix}$$

y si eso lo desarrollamos obtenemos que es

$$(q_x - p_x)(r_y - q_y) - (q_y - p_y)(r_x - q_x)$$

Ahora veamos algo de como usar vectores y productos punto

para ver si un punto esta de un lado o de otro de un punto

Entonces usaremos los mismos puntos p, q y r , entonces ahora

tenemos el vector de p a q como $\vec{pq} = (q_x - p_x, q_y - p_y)$

ahora usaremos el vector ortogonal a \vec{pq} para poder saber el

lado de r , entonces podemos usar que el vector ortogonal sea

$$\vec{o} = (- (q_y - p_y), q_x - p_x) \quad (\text{ya que si tenemos un vector } v = (a, b))$$

sus vectores ortogonales, pueden ser $u = (-b, a)$ o $z = (b-a)$)

entonces ahora usamos el vector de qr como $\vec{qr} =$

$(rx-qx, ry-qy)$, por ultimo usamos el producto punto de

los vectores \vec{qr} y \vec{o} para saber de que lado esta r

(ya que el producto punto nos dice que tanto apuntan dos vectores en la misma dirección), entonces tenemos que si el producto

punto da 0 entonces son vectores ortogonales y lo que significa

que p, q y r son colineares, si el producto punto es negativo

entonces esta de un lado del vector \vec{pq} y si es positivo esta

del otro lado, recordando la definición de producto punto

$(a \cdot b = axbx + ayby)$ tenemos que $\vec{o} \cdot \vec{qr} = -(qy-py)(rx-qx) +$

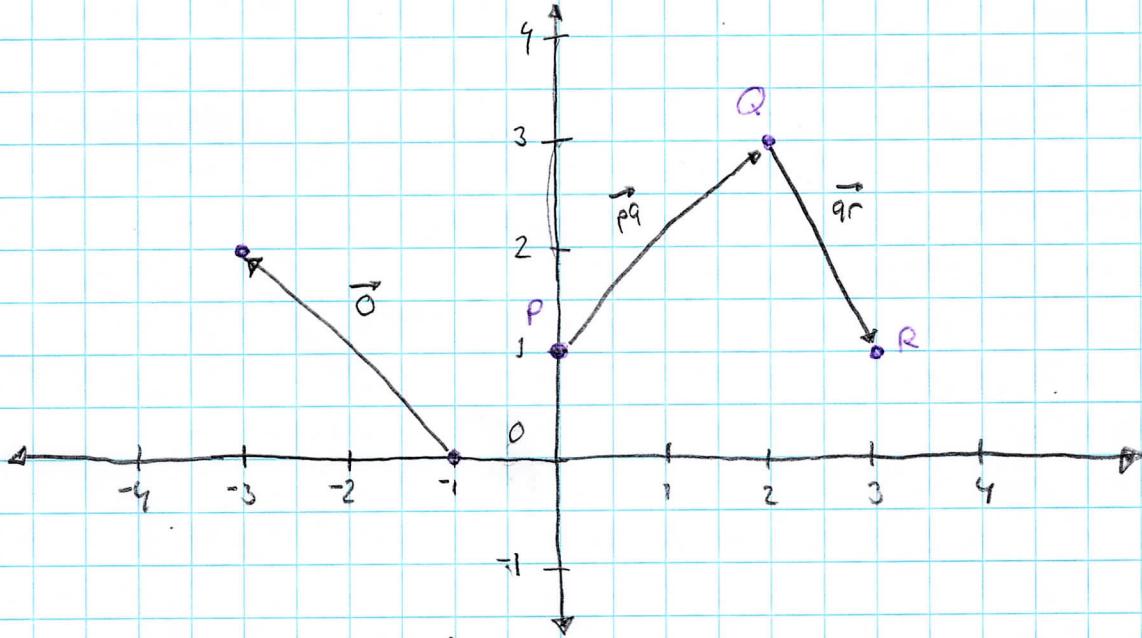
$(qx-px)(ry-qy)$ y reordenando un poco $\vec{o} \cdot \vec{qr} = (qx-px)$

$(ry-qy) - (qy-py)(rx-qx)$, que es lo mismo que obtuvimos de

la determinante, por eso podemos usar el simbolo de esa

determinante para saber de que lado esta r

Veamos un pequeño ejemplo con $P=(0,1)$, $Q=(2,3)$ y $R=(3,1)$



Aquí tenemos que el producto punto de \vec{qr} y \vec{o} es -6

$$(2-0)(1-3) - (3-2)(3-1) = -6$$

y notemos que $\begin{vmatrix} 1 & 0 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 1 \end{vmatrix} = -6$, entonces si sigue
(si es negativo está a la derecha)

Ahora notemos que las coordenadas de los puntos pueden ser reales ya que lo que nos sirve ese signo del determinante entres podemos calcular el determinante ya sea usando los enteros o reales, esto no nos afectara a nuestro resultado, el signo seguira apareciendo

2) Para este ejercicio supondremos que ningún par de triángulos comparten un vértice y que los vértices de cada triángulo conocen a los vértices del triángulo

Ahora, el algoritmo será algo así

Primero, para cada triángulo, identificamos el vértice más alto (con respecto al eje y, y si dos están iguales entonces tomamos el menor de los dos con respecto al eje x) para que sea el vértice de inicio del triángulo, y el vértice más bajo (con respecto al eje y, y si dos están iguales entonces tomamos el mayor de los dos con respecto al eje x) para que sea el vértice de fin del triángulo

Luego, vamos a ordenar los puntos de inicio y fin de todos los triángulos (con respecto al eje y y usando el eje x, cuando dos estén igual en y), de esta manera tendremos los cuadros de nuestro borrado de linea

Después, nuestra linea de borrado tendrá un estado donde se "guardaran" triángulos, para esto usaremos un árbol AVL usando la posición del vértice de inicio para ordenarlos, notemos que podemos

usar el rango en x que abarca el triángulo como factor de ordenamiento para hacer este proceso más eficiente pero no lo hago ya que esa estrategia es algo complicada y no lo entiendo del todo, por lo que en nuestro estado tendremos los triángulos activos. Posteriormente, vamos a procesar los eventos en orden si el evento es un punto de fin de un triángulo entonces sacamos ese triángulo del estado.

si el evento es un punto de inicio de un triángulo, digamos triángulo t , entonces obtenemos los puntos de t , digamos P_1, P_2 y P_3 , ahora por cada triángulo t_i en el estado revisamos si t está dentro de t_i , esto lo hacemos obteniendo las aristas de t_i (usando sus vértices para "crearlas") y luego revisamos si todos los puntos P_1, P_2 y P_3 cumplen que están a la derecha de cada arista de t_i (esto usando la determinante y con el sentido de las aristas de t_i en sentido del reloj, similar a lo visto en clase), si P_1, P_2 y P_3 cumplen que están dentro de t_i entonces avisamos que t está dentro de t_i y

por ultimo , luego de checar con todos los triangulos del estado ,
- agregamos el triangulo t al estado
Al final de procesar todos los eventos sabremos que triangulos estan
dentro de otros

Notemos que esto funciona ya que al detectar que inicia un
triangulo entonces revisamos si esta dentro de alguno de los
triangulos que ya iniciaron pero no terminan aun (el estado) pero
notemos que este algoritmo no usa todo el poder del estado ya que
revisamos a todos los del estado , pero aun asi nos sirve para
detectar los triangulos dentro de otros

Ahora veamos la complejidad de todo , notemos que identificar
los vertices de inicio y fin de cada triangulo nos toma tiempo lineal
 $O(n)$ esto ya que tenemos n triangulos y cada uno tiene 3 puntos/vertices,
por lo tanto para encontrar el mayor usamos 3 comparaciones , es
decir usamos 6 comparaciones para encontrar el de inicio y fin ,
entonces en total usamos $6n$ operaciones lo cual es lineal ,

Luego tenemos que ordenar los puntos de inicio y fin de todos los triángulos, esto nos toma $O(n \log n)$, ya que ordenamos $2n$ elementos (dos puntos por triángulo), después usamos la linea para recorrer los eventos (puntos) los cuales vemos que son $2n$ entonces esto nos toma $O(n)$ por la complejidad de procesar cada evento, posteriormente vemos que tan complejo o tardado es procesar un evento., si el punto es de fin entonces solo sacamos al triángulo de ese punto del estado lo cual nos toma $O(\log n)$ ya que usamos un AVL y es una eliminación, ahora vemos un punto de inicio, aquí primero por cada triángulo en el estado revisamos si el triángulo del punto de inicio esta dentro de ellos, esto se hace viendo direcciones de puntos respecto a aristas lo cual sale en $O(1)$, usando la determinante, entonces ver si el triángulo esta dentro de otro es $O(1)$ y como en el peor caso tenemos n triángulos en el estado nos toma $O(n)$ y luego se agrega el triángulo al estado nos toma $O(\log n)$, una inserción en un AVL

entonces procesar un evento en total nos toma $O(n)$,

como procesamos $2n$ eventos, todos el proceso de eventos nos toma

$O(n^2)$, por lo tanto juntando todo el algoritmo nos toma

$O(n^2)$ lo cual tal vez se pueda mejorar, tal vez una posible

mejora sea usando la idea vista en la acondición del B/3/2S

y usar transitividad para saber si triángulos están dentro de otros,

pero este algoritmo nos toma en total $O(n^2)$, ya que lo

más tardado es procesar a los eventos

3) El algoritmo que usaremos sera el siguiente (usaremos barrido angular)

Primero vamos a ordenar los puntos de los n segmentos de recta, esto lo hacemos con un ordenamiento angular con respecto a p (usando el angulo formado por un rayo de p a los puntos) pero notemos que si dos puntos tienen el mismo angulo entonces diremos que el punto menor es el que esta más cerca de P (usando la distancia euclidea entre los puntos y p)

Luego de ordenar los puntos (los cuales seran los eventos del barrido), vamos a identificar el punto de inicio y fin de cada segmento, el punto de inicio sera menor que el definido en el ordenamiento del paso anterior

Despues, revisamos el primer evento (el menor del ordenamiento angular) si el primer evento es un punto de fin entonces ya no sera el primer evento y lo ponemos al final de los eventos,

repetimos esto hasta que el primer evento sea un punto de inicio

Posteriormente, para el estado de la linea de barrido usaremos un arbol minheap, donde guardaremos los segmentos de recta y seran almacenados usando de criterio de que tan cerca esta esa recta de p en cierto angulo, entonces si vamos a agregar la recta r_1 en el evento del punto v , lo que vamos a hacer es encontrar el lugar de r_1 comparando con los segmentos en el estado (como funciona en minheap), pero como comparamos los segmentos? para esto seguiremos usando r_1 y v , compararemos r_1 con un segmento r_2 , lo primero que hacemos es trazar la recta que pasa por los puntos p y v (notemos que es la linea de barrido), la llamaremos a la recta h , ahora usamos la recta h y el segmento r_1 para encontrar su punto de intersección al cual llamaremos u_1 y usando la recta h y el segmento r_2 para encontrar su

punto de intersección al cual llamaremos u_2 , ahora calculamos la distancia de p al punto u_1 y la distancia de p al punto u_2 , esas distancias las usaremos para comparar los segmentos r_1 y r_2 , de esta manera tendremos a la raíz del minheap al segmento más cercano a p en el angulo en el cual está la linea de barrido

Luego, empezamos a procesar los eventos (los puntos de los segmentos)

Si el evento es un punto de inicio entonces metemos el segmento de recta perteneciente a ese punto a nuestro estado y revisamos cual es el segmento en la raíz del minheap, digamos que es el segmento s , entonces marcamos al segmento s como visto por p

Si el evento es un punto de fin entonces el segmento de recta perteneciente a ese punto de nuestro estado, ahora revisamos si el siguiente evento tiene el mismo angulo que este

evento, si lo tienen entonces seguimos el proceso del proximo evento, pero si el proximo evento no tiene el mismo angulo (los angulos que usamos para ordenar) entonces revisamos cual es el segmento en la raiz del min heap, digamos que es el segmento s , por lo que marcamos al segmento s como visto por P , pero notemos un posible caso es que al intentar eliminar la recta perteneciente al punto (de fin) de nuestro estado falle (es decir que el segmento no esta en el estado) entonces vamos a tener que poner el evento actual y todos los eventos procesados hasta el momento (si es que no estan) de regreso a los eventos pero al final (de esta manera el evento donde falló la eliminación será el nuevo ultimo estado), ademas todas las rectas relacionadas a los eventos que pusimos al final los marcamos como no visto por P

Al final de procesar todos los eventos vamos a saber los

segmentos que podemos ver desde p

Notemos que esto funciona ya que al usar el min heap podemos saber cual es el segmento mas cercano a p (en cierto angulo) y por lo tanto el que podemos ver desde p, esto gracias a la manera de comparar nuestros segmentos, solo hay que tener cuidado en que evento iniciar el proceso para no causar cosas extrañas

Ahora veamos la complejidad de todo, ordenar a los puntos de los n segmentos nos toma $O(n \log n)$ ya que ordenamos $2n$ puntos y para las comparaciones del ordenamiento angular las podemos hacer en tiempo constante, luego marcamos los puntos de inicio y fin, esto se puede hacer en $O(n)$ ya que reusamos $2n$ puntos (suponemos que cada punto conoce al otro punto del segmento) - despues tenemos que asegurarnos que el primer evento sea un punto de inicio para esto en el peor caso tendremos que "desechar" (pasar al final) a n puntos

(ya que hay n puntos de fin) por lo tanto nos toma $O(n)$,

luego vamos a procesar los eventos entonces veamos cuento

nos toma procesar un evento, si el punto es de inicio entonces

tenemos que agregar su segmento al estado que es en min

heap, lo cual sabemos que nos toma $O(\log n)$, pero notemos

que la comparacion entre los segmentos nos va a tomar

tiempo lineal ya que buscamos la intersección entre la linea

de barrido y los dos segmentos a comparar lo cual lo

podemos hacer en tiempo constante (son puras operaciones y

aplicar formulas) y de los dos puntos de intersección usamos

su distancia hacia p para hacer la comparacion lo cual

tambien es constante, por lo tanto procesar un punto de

inicio nos toma $O(\log n)$ ya que ver la raiz del min heap

es constante, para procesar un punto de fin borramos el

segmento del min heap y eso nos toma $O(\log n)$, notemos

que en los puntos de fin podemos regresar eventos ya procesados,

al final de los eventos para que se vuelvan a procesar, entonces esto aumentaría la cantidad de eventos a procesar, notemos que esto solo puede pasar la primera vez que vemos un punto de fin ya que si lo vemos otra vez entonces ya pudimos ver el punto de inicio relacionado a ese punto de fin, así que en el peor caso vamos a regresar a todos los eventos al final (solo regresando los al final una vez), pero regresando a procesar un punto de fin nos toma $O(\log n)$, solo es una eliminación del min heap y revisar el angulo del proximo evento (lo cual es constante) y revisar la raiz del min heap (tambien constante), ahora veamos cuantos eventos vamos a procesar sabemos que tenemos $2n$ eventos (ya que cada segmento tiene un punto de inicio y uno de fin) pero en el proceso de los puntos de fin podemos regresar eventos para que vuelvan a ser procesados (esto solo puede ser la primera vez que vemos un punto de fin) por lo tanto a lo mas vamos a procesar $4n$ eventos (cada evento a lo mas lo

vamos a procesar dos veces), entonces en total procesamos $O(n)$ eventos y procesar un evento nos toma $O(\log n)$ entonces procesar a los eventos nos toma $O(n \log n)$ y por lo tanto, juntando las complejidades podemos notar que todo nos toma $O(n \log n)$ en total

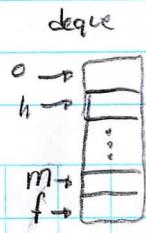
4) Para esto usaremos el algoritmo de Melkman, veamos como funciona

Primero tomamos los tres primeros puntos de la poligonal simple los cuales son p_1, p_2 y p_3 , ahora revisamos si p_3 está a la derecha del segmento de recta de p_1 a p_2 , si está a la derecha entonces metemos p_1 en el tope de un deque (cola doblemente terminada) y luego metemos a p_2 en el tope de la deque, pero si p_3 no está a la derecha entonces metemos a p_2 en el tope del deque y luego metemos a p_1 en el tope del deque

Luego, vamos a procesar los demás puntos de la poligonal,

cuando no tengamos puntos que procesar terminamos

Entonces sea v el siguiente punto de la poligonal a procesar, sea o el elemento del tope del deque, sea h el segundo elemento del tope de la deque (el siguiente de o), sea f el elemento del fin de la deque y sea m el elemento anterior al



elemento del final de la deque , algo así

$m \rightarrow f$

Despues empezamos a procesar al punto V , primero revisamos

si está dentro de la envolvente convexa que llevamos de momento

(la cual es formada por los elementos de la deque) esto lo hacemos

revisando si se cumple que el punto m esta a la izquierda

del segmento de recta del punto v al punto f o que el

punto v esta a la izquierda del segmento de recta del punto

h al punto o , si nada de lo anterior se cumple entonces

el punto v esta dentro del envolvente y terminamos de

procesar al punto v , pero si se cumple al menos uno entonces

seguimos el proceso , ahora revisamos si se cumple

que el punto v esta a la derecha del segmento de recta del

punto h al punto o , si no se cumple sacamos el elemento

del topo de la deque , y volvemos a revisar si se cumple lo anterior

(se repite esto hasta que se cumpla) , pero si se cumple o

cuando se cumpla entonces agregamos el punto v en el topo

de la deque, despues revisamos si se cumple que el punto m esta a la derecha del segmento de recta del punto v al punto f , si no se cumple sacamos el elemento del fin de la deque y volvemos a revisar si se cumple lo anterior (se repite esto hasta que se cumpla), pero si se cumple o cuando se cumpla entonces agregamos el punto v en el fin de la deque, y con esto terminamos de procesar al punto v y seguimos procesando el resto de los puntos Al final de procesar todos los puntos tendremos los puntos que forman la envolvente convexa dentro de la deque y solo los sacamos

Notemos que esto funciona ya que se van agregando puntos a la envolvente convexa poco a poco , por eso ignoramos los que ya estan dentro , ademas al momento de agregar nuevos puntos tambien revisamos los ultimos puntos agregados al deque para sacar los que ya no van a formar parte del

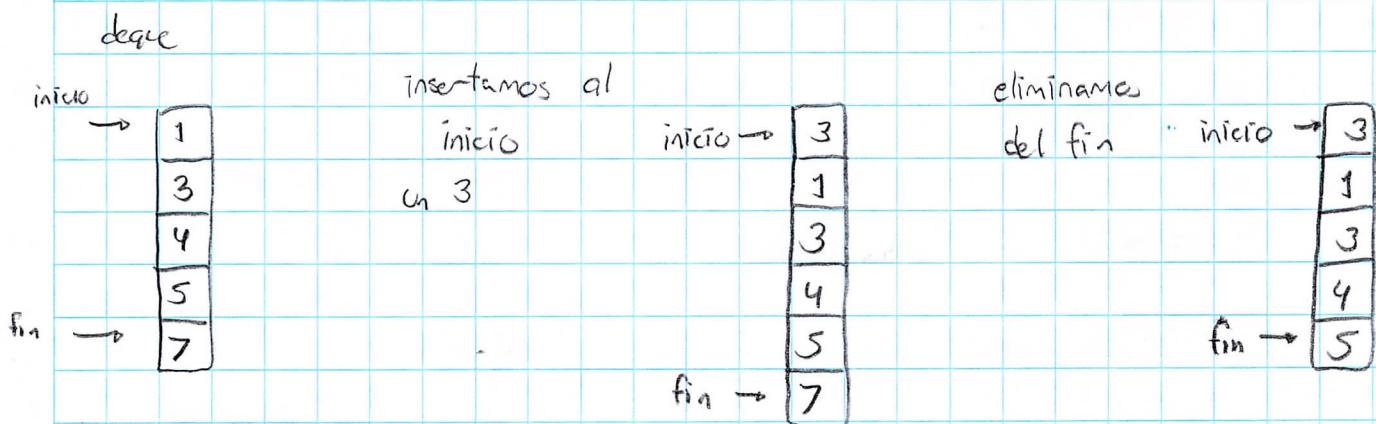
envolvente , esto para que se cumpla que si viajamos por los vértices/puntos de la envolvente en una dirección solo hacemos giros a la derecha y en la otra dirección solo giros a la izquierda , por esto nos sirve este algoritmo

Ahora veamos la complejidad de todo, notemos que ver la dirección de un punto con respecto de un segmento de recta lo podemos hacer en tiempo constante , tambien la operación de agregar o sacar el tope o fin de la deque es en tiempo constante y ver el segundo elemento del tope o del fin tambien es constante , por lo tanto procesar cada punto lo podemos hacer en tiempo constante , por lo que el proceso de los tres primeros vértices lo hacemos en $O(1)$, luego el proceso de los demás vértices es $O(n)$, ya que cada vértice/punto lo metemos al deque al más 2 veces y lo sacamos al más 2 veces , por lo cual en total procesar los vértices nos toma $O(n)$ en total (cada punto entra y sale al más dos

veces de la deque) y al final sacar todos los puntos de la deque para conocer el enunciado tambien es $O(n)$, asi que todo nos toma $O(n)$ y se cumple lo que buscamos

Notemos que lo importante para este algoritmo es que usamos una deque, la cual es una queue donde se puede insertar y eliminar al inicio o final

por ejemplo



5) Observemos que el algoritmo visto en clase consiste en dos algoritmos, el primero es convertir un polígono simple en piezas monótonas (clase 17/2/25) y el segundo es triangular un polígono monótono (clase 19/2/25), por lo tanto vamos a modificar el primer algoritmo ya que tiene complejidad $O(n \log n)$. (Todo lo siguiente lo haremos con y -monótono pero es igual para x y α)

Entonces lo primero que haremos es recorrer todos los vértices del polígono simple y marcar los que son vértices no regulares, esto lo podemos hacer en $O(n)$ ya que solo recorremos todos los vértices

Ahora, por cada vértice no regular vamos a procesarlo si el vértice es de inicio o fin no los procesamos, veamos el caso para un vértice divisor, entonces vamos a recorrer los vértices para encontrar al vértice v que cumpla lo siguiente: que v esté arriba (en el eje y) que el vértice divisor y se pueda poner una diagonal entre v y el vértice

divide (sin causar intersecciones con los segmentos y diagonales del polígono), además que sea el vértice más cercano al vértice divide que lo cumpla, esto lo podemos hacer en tiempo $O(n)$ ya que recorremos a todos los vértices viendo cuáles cumplen lo que queremos y guardando el más cercano que lo cumple mientras revisamos los puntos, luego ponemos la diagonal entre el vértice divide y el vértice encontrado, logrando quitar un vértice divide de nuestro polígono

para el caso de vértice une va a ser similar al de divide, solo vamos a recorrer los vértices buscando al vértice z que cumple: el vértice z está abajo (en el eje y) que el vértice une, se pide poner una diagonal entre el vértice z y el vértice une y que el vértice z es el vértice más cercano al vértice une de todos los vértices que cumplen las primeras dos condiciones, esto lo podemos hacer en $O(n)$ de igual manera que el caso del vértice divide (recorremos todos los vértices)

buscando) , luego ponemos la diagonal entre el vértice one
y el vértice encontrado , logrando quitar un vértice divide
de nuestro polígono

Notemos que llego de procesar nuestros vértices no regulares
vamos a quitar los vértices one y divide por lo tanto
logramos que el polígono se vuelva monótono pero tenemos
varios polígonos monótonos por las diagonales lo cual es el
resultado de usar el primer algoritmo (contenemos el polígono
simple en piezas monótonas) , entonces como procesamos $O(1)$
vértices regulares y cada procesamiento toma $O(1)$, entonces
todo esto nos toma $O(n)$ y generamos a lo más $O(1)$
piezas monótonas , ya que agregamos $O(1)$ diagonales y
cada una puede dividir el polígono en dos (es decir cada
diagonal agrega una pieza)

Ahora a cada una de las piezas monótonas les aplicamos el
segundo algoritmo para triangularlas en tiempo $O(n)$ y como

son $O(1)$ piezas entonces triangulamos en $O(n)$

y por ultimo unimos todas las piezas para tener la triangulación del polígono simple original (conservando las diagonales que agregamos y las de la triangulación) y como sabemos que una triangulación de un polígono simple tiene $n-3$ diagonales esto nos toma $O(n)$

Y en total todo el proceso nos toma $O(n)$ para triangular un polígono simple con $O(1)$ vértices no regulares. Notemos además que los vértices que buscamos para que los vértices divide y une (al poner diagonales) van a existir ya que sabemos que un polígono simple se puede partitionar en piezas monótonas, como vimos la clase del 17/02/25 donde ponemos diagonales entre los vértices une y divide, y los vértices del estado, entonces nosotros en vez de hacer barrido de linea y tener los puntos en el estado lo que hacemos es buscarlos

6) En clase (14/2/25) vimos un teorema que dice "todo polígono simple de n vértices se puede triangular y tendrá $n-2$ triángulos y $n-3$ diagonales", notemos que ese teorema se demostró en clase entonces para este ejercicio veremos que todo polígono simple con h hoyos y n vértices se puede triangular, usaremos inducción sobre el número de hoyos (h), supondremos que $n \geq 3$ y los polígonos son simples.

- Caso base ($h=0$)

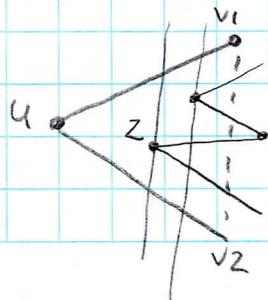
Si un polígono simple P no tiene hoyos entonces sólo es un polígono simple y como tiene n vértices entonces usamos el teorema visto en clase y sabemos que P se puede triangular y por lo tanto admite una triangulación.

- Hipótesis de inducción

Supongamos que un polígono simple con menos de h hoyos y cualquier cantidad de vértices puede ser triangulado y admite una triangulación.

- Paso inductivo

Sea P un polígono simple de n vértices y h hoyos, ahora recordemos que un polígono con hoyos es un polígono con uno o varios polígonos dentro los cuales no intersectan entre ellos o con el polígono exterior, ahora usaremos una técnica similar a la usada en clase y usaremos una diagonal que existe dentro del polígono, veamos que esta diagonal debe existir para esto tomamos al vértice más a la izquierda de P , digamos que es u y sea v_1 y v_2 los dos vecinos de u , ahora revisamos si el segmento de recta del punto v_1 al punto v_2 es una diagonal y no intersecta a ninguna arista de P , si esto se cumple entonces encontramos la diagonal pero si no pasa esto entonces encontramos al vértice más cercano a u usando rectas paralelas al segmento del punto v_1 al punto v_2 , algo así (como lo vimos en clase) y llamamos a ese punto



como z , notemos que si la recta de v_1 a v_2 no fue una diagonal entonces z

debe existir, si no la recta si fuera una diagonal (interna), entonces el segmento de recta del punto u al punto z sera nuestra diagonal, ahora veamos algunos casos

caso 1) la diagonal tiene un punto que pertenece a un hoyo entonces, sean a y b los puntos de la diagonal entonces vamos a partir al punto a en dos puntos a' y a'' (los cuales a' tendrá las aristas a la derecha de a y a'' las de la izquierda) a' y a'' no tienen aristas entre ellos), igual partimos b en b' y b'' y agregamos diagonales entre a' y b' y otra entre a'' y b'' , de esta manera quitamos el hoyo (lo conectamos con el polígono grande o exterior), por lo que P ahora tiene $h-1$ hoyos y $n+2$ vértices, usamos la hipótesis de inducción y sabemos que admite una triangulación, se puede triangular, solo notemos que cuando tengamos la triangulación tomaremos a los puntos a y a' como uno, de igual manera para los puntos b y b' y para el segmento de a a b y el segmento de b' a a'

caso 2) la diagonal no tiene ningún punto que pertenezca a un

hoyo, entonces la diagonal nos va a partir P en dos polígonos

$P_1 \cup P_2$ (como lo visto en clase) donde P_1 tiene n_1 vértices,

y h_1 hoyos y P_2 tiene n_2 vértices y h_2 hoyos, con $n_1 < n$,

$n_2 < n$ y $h_1, h_2 \leq h$, ahora tenemos dos subcasos

caso 2.1) $h_1 < h$ y $h_2 < h$, entonces usamos la hipótesis de

inducción a P_1 y P_2 , entonces admite y se pueden triangular,

si unimos sus triangulaciones obtenemos la triangulación de P , entonces

admite y se puede triangular

caso 2.2) $h_1 = h$ y $h_2 = 0$, entonces podemos aplicar la

hipótesis de inducción a P_2 y sabemos que admite y se puede

triangular, ahora como P_1 tiene h hoyos y n_1 vértices podemos

repetir todo el proceso de encontrar una diagonal (lo que hemos

visto en el paso inductivo) hasta lograr triangularlo al dividirlo

en dos polígonos con menor cantidad de hoyos (caso 2.1) o quitar

un hoyo (caso 1), y así poder aplicar la hipótesis de

inducción y esto es posible ya que si vanas veces caemos

en el caso 2.2 (o caso 2.3) entonces vamos a estar disminuyendo la cantidad de vértices por lo tanto en algún momento terminaremos y como los hoyos no tocan al polígono exterior entonces en algún momento los podremos "desaparecer", entonces en algún punto P_1 admite y se puede triangular, por lo que juntando la triangulación de P_1 y P_2 obtenemos la triangulación de P , por lo cual P admite y se puede triangular.

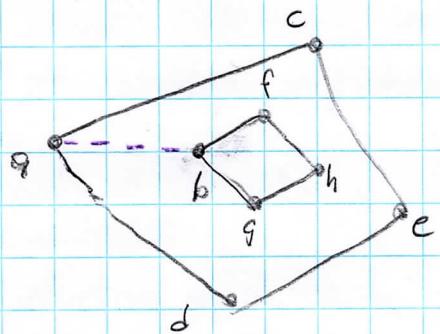
caso 2.3) $h_1 = 0$ y $h_2 = h$, este caso es análogo al caso 2.2

Entonces podemos notar que en todos los casos logramos ver que P admite una triangulación y se puede triangular.

Por lo que podemos concluir, gracias a la inducción, que un polígono simple con h hoyos y n vértices admite una triangulación y se puede triangular. \square

* explicación un poco mayor de lo de quitar el hoyo

lo que tenemos en el caso f es algo así (un ejemplo)

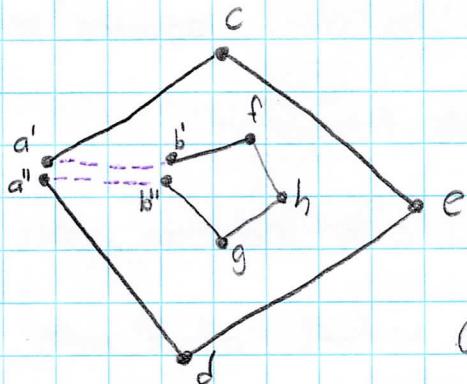


entonces tenemos la diagonal entre a y b

por lo tanto ponemos a a en a' y a'' (y las ponemos muy

juntas - por no decir en la misma posición) , igual ponemos a bien

b' y b'' , y agregamos las diagonales entre a' y b' y entre



a'' y b' , de esta manera

quitamos el hoyo

como se muestra en la imagen

(un poco exagerada para que se note el

cambio) , tendremos que a' tiene arista

con c y b' (que es la diagonal agregada) , a'' tiene arista

con d y con b'' (la otra diagonal agregada) , b' tiene arista con f

y a' , y b'' tiene arista con g y a'' , así el polígono ya no tiene

hoyo

7)

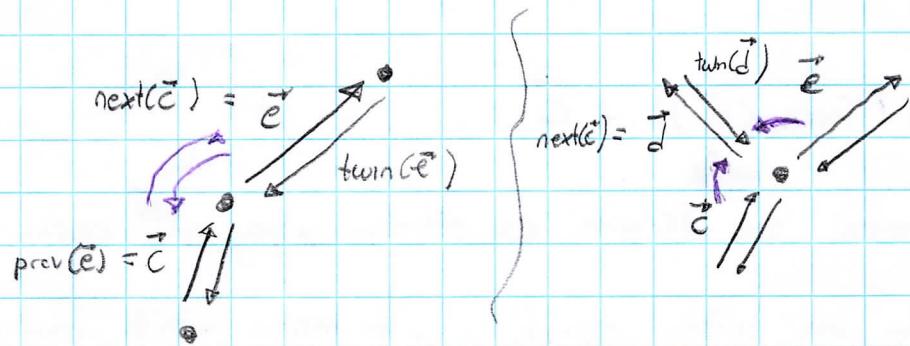
$$a) \text{twin}(\text{twin}_a(\vec{e})) = \vec{e}$$

esto tenemos que siempre es cierto , ya que cada arista solo tiene una arista gemela , la misma en la otra dirección por ejemplo si \vec{e} es la arista de a a b , entonces $\text{twin}(\vec{e})$ es la arista de b a a y $\text{twin}(\text{twin}(\vec{e}))$ seria la arista de a a b y eso es \vec{e} , si esto no se cumple seria que una arista tiene más de una gemela,

$$b) \text{Next}(\text{Prev}(\vec{e})) = \vec{e}$$

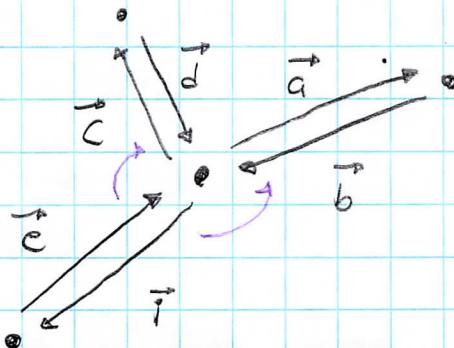
esto tambien siempre es cierto , ya que para ver quien es $\text{prev}(\vec{e})$ usamos la arista más cercana en contra de las manecillas del reloj y para $\text{next}(\vec{e})$ usamos la arista más cercana en las manecillas del reloj entonces usamos las más cercanas , si tuvieramos que $\text{prev}(\vec{e}) = \vec{c}$ y , que $\text{next}(\vec{c}) = \vec{d}$ entonces notaríamos que $\text{prev}(\vec{e})$ no podría ser \vec{c} ya que $\text{twin}(\vec{d})$ estaría más cerca a \vec{e} en contra de las manecillas del reloj

algo así



$$c) \text{Twin}(\text{Prev}(\text{Twin}(\vec{e}))) = \text{Next}(\vec{e})$$

esto no siempre pasa, veamos un contra ejemplo



aquí tenemos que

$$\text{twin}(\vec{e}) = \vec{i},$$

$$\text{prev}(\vec{i}) = \vec{b},$$

$$\text{twin}(\vec{b}) = \vec{a} \text{ y}$$

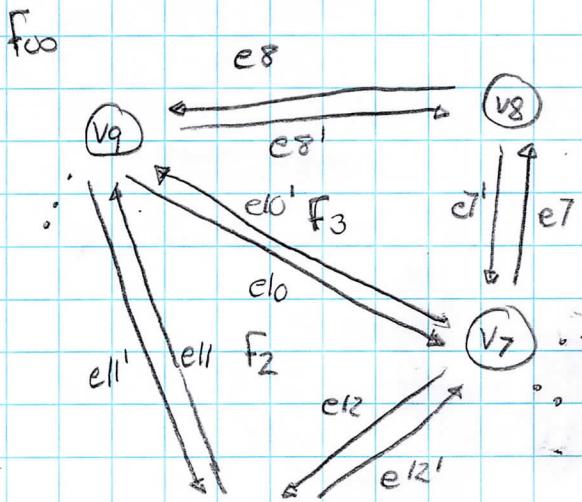
$$\text{next}(\vec{e}) = \vec{c}$$

$$\text{entonces } \text{twin}(\text{prev}(\text{twin}(\vec{e}))) = \vec{a}, \text{ next}(\vec{e}) = \vec{c} \text{ y}$$

$$\vec{a} \neq \vec{c}, \text{ por lo que no se cumple}$$

d) $\text{IncidentFace}(\vec{e}) = \text{IncidentFace}(\text{Next}(\vec{e}))$

esto no siempre pasa, veamos un contra ejemplo (usaremos una fracción del DCEL visto la ayudantía del 27-02-25)



notemos que

$$\text{IncidentFace}(e_8) = F_0,$$

$$\text{Next}(e_8) = e_{10} \quad \times$$

entonces tenemos que

$$\text{IncidentFace}(e_8) = F_0,$$

$$\text{IncidentFace}(\text{next}(e_8)) = F_2$$

y $F_0 \neq F_2$ - por lo que no se cumple (en el dibujo de la ayudantía está el DCEL completo)

8) El algoritmo que usaremos sea el siguiente (supondremos que las aristas de la subdivisión conocen las caras que están a sus lados) , y que S es una subdivisión plana conexa)

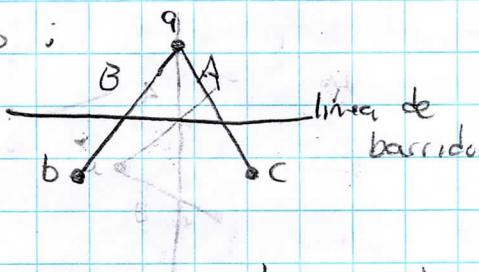
Primero vamos a ordenar los puntos de P y los puntos de S , esto lo haremos con respecto al eje y (es decir el punto con coordenada y mayores sera el menor punto) , si dos puntos tienen la misma coordenada y entonces el menor sera el punto con coordenada x menor , tambien notemos que supondremos que ningun punto de S tiene la misma posición que cualquier punto de P (en caso de que permitamos esto entonces necesitaríais hacer que si un punto de S , otro punto de P tienen la misma posición poner que sea menor el punto de S en el ordenamiento , luego poner que el punto de P es especial , y en el barrido de linea procesarlo como más nos convenga)

Luego de ordenar a los puntos de S , P (los cuales seran los encabezados del barrido de linea) , vamos a identificar los puntos

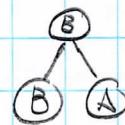
de inicio y fin de cada arista de la subdivisión, notemos que un punto puede ser tanto el punto de inicio de una arista y el punto de fin de otra arista por lo tanto marcamos los puntos de inicio y/o final así como la arista relacionada (notemos que una arista tiene dos puntos entonces el de inicio sera el menor en el ordenamiento del paso anterior y el punto de fin sera el otro)

Después, veremos que para el establecimiento de la linea de barrido vamos a usar un árbol AVL, donde guardaremos aristas de la subdivisión, y el orden que mantendrán sera su posición en la linea de barrido por ejemplo si tenemos

esto:



así sera el árbol AVL



de esta manera las aristas más a la izquierda de la linea de barrido serán las aristas menores en el AVL

Posteriormente, empezamos a recorrer los eventos

si el evento es un punto de S y solo es un punto de inicio entonces sean a_1, \dots, a_i las aristas relacionadas a ese punto (las aristas donde es un punto de inicio) vamos a meter esas aristas al estado de la linea de barrido , notemos que esas aristas tienen el mismo punto de inicio entonces cuando las insertemos en el arbol usaremos la posición del punto de fin de cada arista para compararlas y ver cuales estan más a la izquierda en la linea de barrido , de esa manera insertamos las aristas a_1, a_i en el estado (en el vista tiene las aristas activas y estan ordenadas conforme que tan a la izquierda de la linea de barrido estan)

si el evento es un punto de S y solo es un punto de fin entonces sean a_1, \dots, a_r las aristas relacionadas a ese punto (las aristas donde es un punto de fin) vamos a eliminar esas aristas del estado de la linea de barrido

si el evento es un punto de S y es un punto de inicio y fin

entonces sean a_1, \dots, a_i las aristas donde el punto es un punto de fin para las aristas y sean b_1, \dots, b_j las aristas donde el punto es un punto de inicio para las aristas entonces vamos a eliminar las aristas a_1, \dots, a_i del estado de la linea de barrido y luego agregamos las aristas b_1, \dots, b_j al estado (procurando hacerlo como mencionamos antes y hemos hecho en clase, en particular como cuando agregábamos aristas al estado durante el barrido de linea para encontrar intersecciones) y así podremos tener en nuestro estado las aristas activas (las aristas activas son las aristas que ya vimos su punto de inicio pero todavía no el de fin) ordenadas sobre que tan a la izquierda de la linea de barrido están si el punto es un punto de P entonces vamos a usar el estado para encontrar entre que aristas, para esto nos ponemos en la raíz del arbol y vamos viendo si el punto esta a la izquierda o derecha de cada recta para

as poder encontrar entre que aristas se encuentra (aristas del estacado), notemos que tenemos tres posibles casos

caso 1) que al intentar buscar el lugar del punto pase que el estacado este vacío, entonces diremos que la cara que contiene al punto es la exterior, ya que no está junto a ninguna arista,

caso 2) que al buscar el lugar del punto encontramos que solo esta a lado de una sola arista, entonces diremos que la cara que contiene al punto es la exterior, ya que como solo resta a lado de una arista entonces debe estar en el exterior

caso 3) que al buscar el lugar del punto pase que el punto esta sobre una arista, entonces le preguntamos cuales son las caras que están a lado de la arista y decimos que esas dos caras contienen al punto (notemos que podemos modificar como manejamos esto para que sea como lo que esperamos)

caso 4) que al buscar el lugar del punto pase que el punto este entre la arista c y la arista d, entonces le preguntamos

a la arista c y d cuales son las caras estan a sus lados , nos van a decir cuatro caras pero vamos a tener que ambas aristas van a decir una misma cara (esto ya que esas aristas estan juntas o a lado en el estado entonces deben compartir una cara) entonces diremos que la cara que comparten las aristas c y d es la cara que contiene al punto

Al final de procesar todos los eventos , sabremos que caras contienen a cada punto de P

Notemos que esto funciona ya que usaremos el estado (el arbol AVL) para mantener ordenado las aristas de la subdivision que estan activas y de esta manera cuando nos toque ver que cara contiene un punto de P , solo buscamos entre las que aristas activas se encuentra y listo

Ahora veamos la complejidad de todo , primero ordenar los puntos de P y S nos toma $O(n+m \log(n+m))$ esto debido a

que ordenamos n puntos de S junto con m puntos de P , luego se identifican los puntos de inicio y fin bacial nos toma $O(n)$ ya que revisamos todos los vértices/puntos de S , luego notemos que insertar aristas o eliminar aristas de nuestro estado (Árbol AVL) nos va a tomar $O(\log n)$ esto ya que es la complejidad de un árbol AVL y los elementos que tiene el árbol son aristas de las cuales, usando lo visto en la clase del 24/2/25 sabemos que la subdivisión plana conexa (por esto suponemos que S es conexa para poder usar este resultado y saber el número de aristas) tiene que la cantidad de aristas es la lo más $3v - 6$ con v la cantidad de vértices, por lo tanto tendremos a lo más $3n - 6$ aristas es decir $O(n)$ aristas, después la búsqueda del lugar de un punto de P en el estado nos toma $O(\log n)$ ya que realizamos una búsqueda por el árboliendo si el punto está a la izquierda o derecha de las aristas para encontrar su lugar

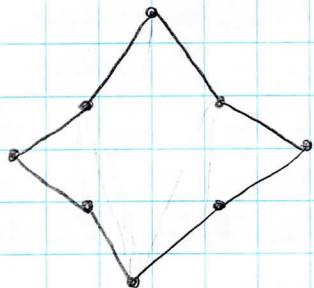
y por lo tanto las comparaciones serán constantes, ahora veamos que tenemos $O(ntm)$ eventos pero al estado solo se insertará una vez cada arista y solo se sacará una vez, por lo tanto procesar a los puntos de S nos va a tomar $O(n \log n)$ (ya que tenemos que insertar y eliminar una vez cada una de las $O(n)$ aristas) y por cada punto de P tenemos que buscar en el estado entonces esto nos toma $O(m \log n)$ (ya que buscamos el árbol que tiene a lo más $O(n)$ elementos/ aristas activas y por cada punto de P buscamos), por lo tanto procesar nuestros eventos nos toma $O((ntm) \log n)$, entonces juntando todas las complejidades podemos ver que todo nos toma $O(ntm) \log(ntm))$, esto ya que ordenar tiene que "más tomo" y de esta manera se cumple lo que pide el ejercicio

Observemos que si no suponemos que las aristas no conocen las caras

que estan a sus lados , lo que podemos hacer es construir la DCEL de la subdivision y usar esto para ver la cara de la arista y de su gemelo , entonces para esto tendriamos que construir el DCEL ; luego revisar la tabla correcta , sabemos que la subdivision plana conexa tiene complejidad geometrica de $O(n)$ (clase 24/02/25) entonces para construir la DCEL tendriamos que recorrer varias veces la subdivision lo que nos toma $O(n)$ entonces podemos construir la DCEL en $O(n)$ (talvez , no estoy muy seguro) y luego usar la DCEL para ver las caras a lado de una arista , entonces la complejidad de todo se sigue respectando

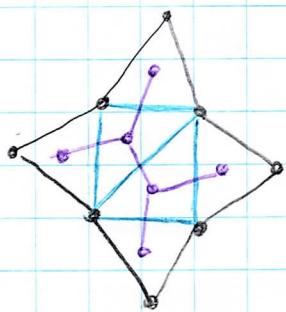
9) Esto no siempre se cumple veamos un ejemplo

tendremos este polígono γ -monotono

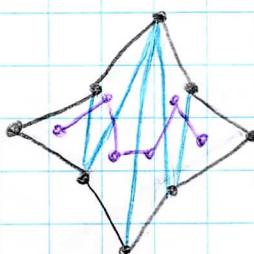


ahora notemos estas dos posibles triangulaciones
→ graficas diales de las triangulaciones

triangulacion 1



triangulacion 2



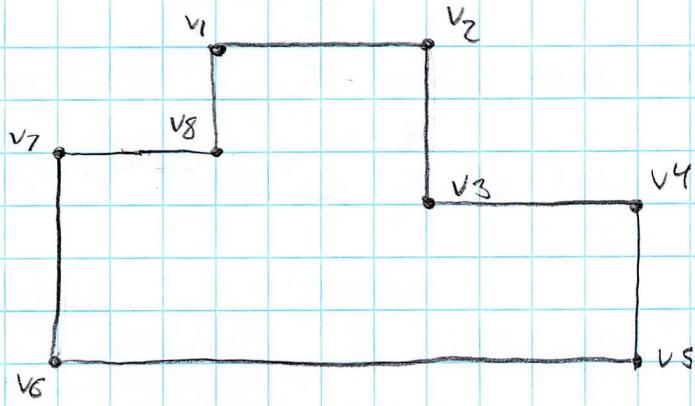
- triangulacion

- grafica dia!

notemos que en la triangulacion 1 , la grafica dual tiene un nodo con grado tres (en particular dos nodos con grado tres) por lo cual no se cumple lo que sugiere el ejercicio ya que la grafica dual de la triangulacion de un polígono monotono tuvo un nodo con grado tres , pero notemos que en la triangulacion 2 si se cumple que sea una cadena , esto solo pasa si cada

triángulo de la triangulación del polígono tiene una arista que forma parte del polígono original (como en la triangulación 2) de esta manera solo tiene dos posibles caras con las cuales estar en contacto con otros triángulos y así procurar que cada nodo tenga a lo más grado dos, por lo cual no todas las triangulaciones cumplen b el ejercicio

10) tendremos este polígono



ahora notemos que si ponemos lámparas en v_6 y v_2 podemos iluminar a todo el polígono , esto ya que v_6 puede ver a todos los vértices excepto a v_1 , por eso la otra lámpara en v_2 (que si puede ver a v_1) , $n=8$ entonces $\lfloor \frac{n}{4} \rfloor = \lfloor \frac{8}{4} \rfloor = 2$ por lo que se cumple

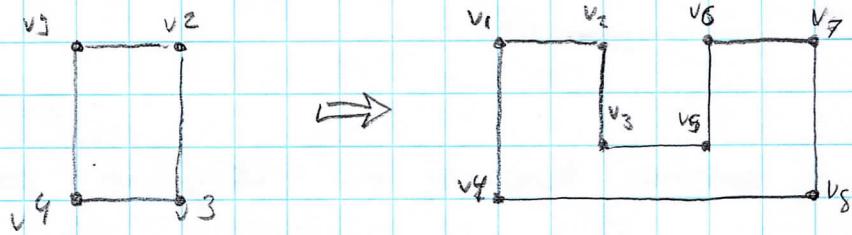
(posible)

Ahora veamos una explicación de porque se cumple lo de $\lfloor \frac{n}{4} \rfloor$ lámparas, notemos que el polígono rectilíneo más simple es un rectángulo , del cual sabemos que una lámpara si no el rectángulo tiene cuatro vértices entonces se cumple , ahora notemos que si le agregamos dos vértices nuevos obtenemos como una letra "L" y aun podemos iluminar todo el polígono , por lo tanto

notamos que al agregarle cuatro nuevos vértices al cuadrado

podemos crear una nueva zona donde nuestra lámpara ya no pueda ver, sera al rectángulo original pegarle una "L"

algo así

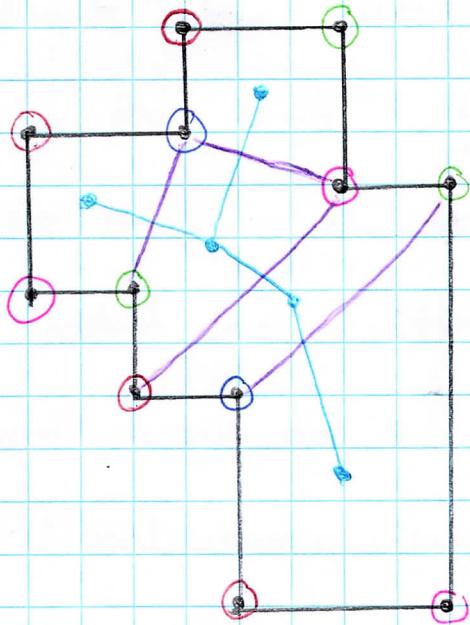


por lo tanto podemos notar que al agregar cuatro vértices al rectángulo necesitamos una nueva lámpara. esto se puede repetir varias veces para crear polígonos rectilíneos, y esa es la razón por la que creo que se usan $\lceil \frac{n}{4} \rceil$ lámparas, ya que necesitamos cuatro vértices nuevos para crear una nueva área que las lámparas actuales no iluminen (o un reacomodo de las lámparas actuales, no ilumine)

Otra posible explicación de este es que como sabemos que solo tenemos aristas horizontales y verticales podríamos decir que

estos polígonos ya son y-monotones (esto depende de si las curvas horizontales cambian o no el sentido, esto se puede ver como que si lo cambia o que no) por lo tanto podríamos dividir el polígono en piezas monótonas o ya dejarlo así, pero lo importante es que podemos triangularlo pero también lo podemos partir en cuadriláteros, esto ya que solo tenemos ángulos internos de 90° y 270° (?), entonces podemos hacer la gráfica dual de esos cuadriláteros y así poder tener que cada nodo de la gráfica tiene cuatro puntos del polígono (dos de los cuales comparte con sus nodos vecinos) entonces podríamos colorear los puntos de cada nodo usando uno de cuatro colores y así poder pintar los puntos del polígono usando cuatro colores y solo elegir un color para poner las lámparas en los puntos de ese color, y obtener el L^4_4 , algo similar a lo visto en clase pero en vez de triangular aquí dividimos en cuadriláteros, no estoy tan seguro de esta explicación pero

veamos un ejemplo



- división en cuadriláteros
- gráfica dual usando los cuadriláteros
- color 1
- color 2
- color 3
- color 4

de esta manera vemos que pidimos usar la gráfica dual de los cuadriláteros y pintar los puntos del polígono, muy similar a lo visto en clase

y el ejemplo se podría iluminar con lámparas en los puntos verdes, logrando $\lfloor \frac{n}{4} \rfloor$ lámparas