UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ciencias

Autor: Adrián Aguilera Moreno

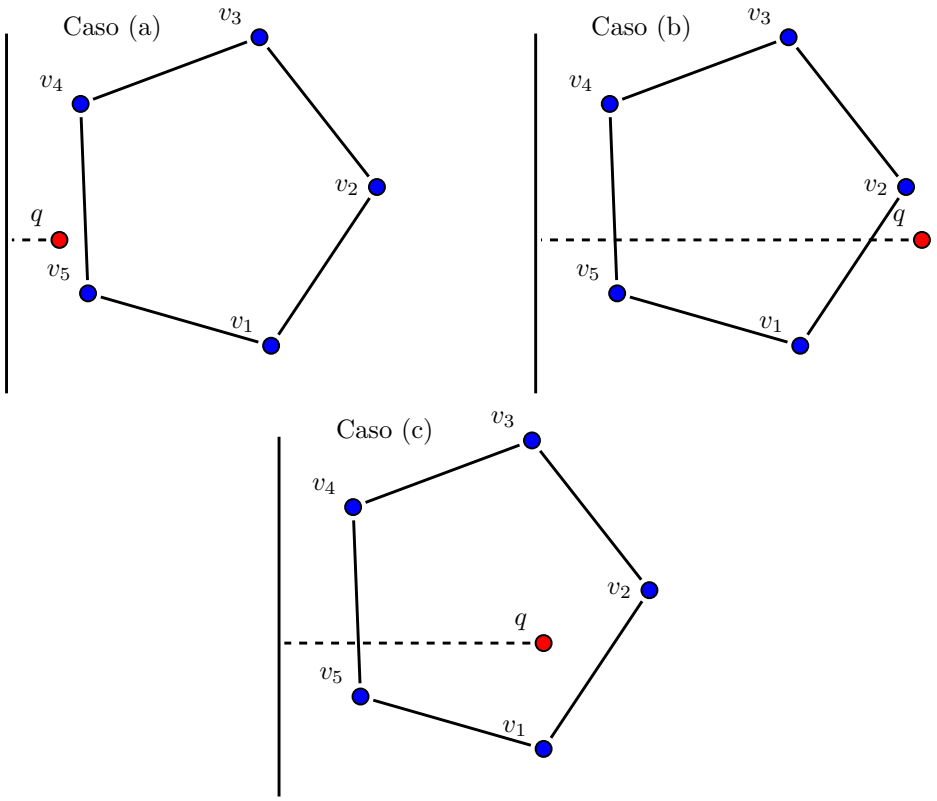

# Geometría Computacional

# Tarea 02

**1.** Sea $P$ un polígono convexo de $n$ vértices. Suponga que los vértices de $P$ son dados en un arreglo ordenado. Muestra que, dado unpunto $q$, podemos detectar si $q \in P$ en tiempo $\mathcal{O}(\log n)$.

***Demostración:*** Exhibamos un algoritmo en tiempo $\mathcal{O}(\log n)$ para encontrar que $q \in P$. Esto lo podemos hacer de varias maneras, en partícular podemos usar una de las estructuras de datos vistas en clase o podemos usar una técnica basa en búsqueda binaria. Por simplicidad usaré la técnica basada en búsqueda binaria, a continuación se muestra el algoritmo deseado:

1. Encontremos el extremo "más alto" (digamos $a$) y "más bajo" (digamos $b$) en $P$ en $\mathcal{O}(1)$, pues tenemos los puntos ordenados en un arreglo.
2. Verificamor que $b.y \leq q.y \leq a.y$ en $\mathcal{O}(1)$. Si la condición anterior se cumple, entonces seguimos con el algoritmo. Si no se cumple la condición anterior, entonces concluimos que $q$ no se encuentra destro de $P$.
3. Ahora tracemos una recta $\ell$ a partir de $q$ tal que intersecte al polígono $P$ respecto a una proyección, digamos la proyección hacia la izquierda. Ahora, tendremos tres posibles opciones:
   (a) $\ell$ intersecta en 0 puntos (Caso análogo a 2). Entonces $q \notin P$.
   (b) $\ell$ intersecta en 2 puntos (Caso análogo a 1). Entonces $q \notin P$.
   (c) $\ell$ intersecta en 1 punto, entonces $q \in P$.




**Obs.** Encontrar $q$ respecto a los puntos de $P$ se realiza en $\mathcal{O}(\log n)$. Producir las recta y encontrar las intersecciones lo realizamos en $\mathcal{O}(1)$.

¿Por qué será cierto que lo anterior es suficiente? R./ Sabemos que $P$ es convexo, entonces toda línea trazada puede cortar a $P$ en a lo más 2 puntos. Además, como la proyección siempre es a la izquierda podemos tomar como referencia los puntos de corte respecto al polígono[1]. En otro caso, podemos considerar una orientación distinta para la línea en la que proyectaremos $\ell$.

*Análisis de Complejidad.* La complejidad del algoritmo esta contenida en

$$\mathcal{O}(1) + \mathcal{O}(\log n) \in \mathcal{O}(\log n)$$

□

**2.** Muestra que si un polígono tiene $\mathcal{O}(1)$ vértices no regulares, entonces el algoritmo de triangulación visto en clase se puede adaptar para usar tiempo $\mathcal{O}(n)$.

▷ **Solución:** Recordemos que el algoritmo visto en clase toma $\mathcal{O}(n + k \cdot \log n)$, dónde $n$ es lo que nos cuesta realizar el barrido punto a punto (estaciones por eventos) y $k \cdot \log n$ son los $k$ vértices une y divide (vértices no regulares), y la búsqueda que se debe hacer en el estado de la línea (cada vértice une y divide) es exactamente una búsqueda binaria en $\log n$.

Ahora, observemos que $k \in \mathcal{O}(1)$, lo que implicaría que

$$\begin{aligned} k \cdot \log n \approx \log n \quad &\Rightarrow \quad \mathcal{O}(n + k \cdot \log n) \approx \mathcal{O}(n + \log n) \\ &\Rightarrow \quad \mathcal{O}(n + \log n) = \mathcal{O}(n). \end{aligned}$$

**Obs.** [Acerca del orden] Se considera que tenemos un orden y que nuestro polígono no es regular. Sin embargo, se me hace curioso saber ¿Cómo obtenemos un orden en tiempo $\mathcal{O}(n)$? Sabemos que no podemos ordenar por comparaciones en menos que $\mathcal{O}(n \log n)$, sin embargo, en un barrido de línea no necesitamos ordenar con prioridades respecto a $(x, y)$ y de hecho basta con tener un orden en $x$ o en $y$. Así, podemos ordenar basándonos en una proyección, digamos que proyectamos cada punto al eje $X$, esto nos tomará $O(n)$, luego basta recorrer el eje $X$, en $\mathcal{O}(n)$, y tenemos un orden (No angular), este orden es suficiente para nuestro barrido de línea. La construcción de la estructura para el estado de la línea se hará en tiempo $\mathcal{O}(n)$, pues son $n$ puntos y nuestras consultas son $k \in \mathcal{O}(1)$ sobre el estado de la línea que sabemos tardan $\log n$.

De esta manera, nuestra complejidad queda contenida en $\mathcal{O}(n)$. ◁

[1] Es simple plantear un pequeño sistemas de ecuación que me indique en cuántos puntos se corta respecto a $\ell$, todo esto en $\mathcal{O}(1)$.

**3.** Demuestra que cualquier polígono admite una triangulación, incluso si el polígono tiene hoyos.

***Demostración:*** Para este ejercicio basta exhibir un algoritmo que genere la triangulación del polígno con hoyos. La idea será separar nuestro polígono en polígonos sin hoyos y triangularlos por separado. A continuación se exhibe el algoritmo deseado:

1. Ordenemos nuestros puntos. Esto lo realizamos en $\mathcal{O}(n \log n)$.

2. Realizamos un barrido de línea para encontrar los hoyos. Este barrido nos tomará $\mathcal{O}(n)$, y detectaremos hoyos cuando el estado de la línea nos indique que hay al menos 4 aristas en su posición y conforme el barrido avance esto se sigue cumpliendo hasta que eventualmente llegamos a que 2 de los (al menos) 4 segmentos que siempre estuvieron en el estado de la línea (no necesariamente los mismos segementos), siempre fueron "internos" de los otros segmentos (siempre estuvieron "entre ellos").

   Una manera de hacer esto sería coloreando las aristas, si en un punto el estado de la línea me indica que las trayectorias formadas hasta el momento (con el avance de la línea) y coloreadas de distinto color se unen, entonces cambio el color a la trayectoria más pequeña. Si una trayectoria núnca se unió y además forma un ciclo interno al polígono, entonces este es un hoyo. **Obs.** Tomar esta idea sin mayor cuidado nos puede llevar a una complejidad cuadrada (aquí podemos hacer uso de la estrategia usada en el agoritmo de Borůvka-Kruskal para encontrar ciclos y agrupar).

3. Ahora que hemos encontrado los ciclos, podemos con gran facilidad segmentar nuestro polígono en nuevos polígonos sin hoyos. Para esto;

   (a) tomamos una arista del hoyo $O_i$, digamos $e$, entonces unimos los extremos de $e$ con los vértices más cercanos en el polígono y verificamos si estas nuevas aristas son "estrictamente internas al polígono" o no. En caso de que sí, entonces serán parte de nuestra subdivisión, en otro caso la descartamos. Esto lo podemos hacer con ayuda de un nuevo barrido y nos toma $\mathcal{O}(n)$ barrer nuestros puntos y $\mathcal{O}(n)$ realizar las comparaciones para encontrar las nuevas aristas para nuestra subdivisión. Si el número de aristas de $O_i$ es lineal respecto al número de vértices en el polígono, entonces tendremos que hacer $\mathcal{O}(n^2)$ comparaciones. De lo anterior tenemos una subdivisión.

4. Ahora aplicamos el algoritmo de triangulación visto en clase a cada parte de la subdivisón encontrada.

*Análisis de complejidad.* La complejidad de este algoritmo esta contenida en

$$\mathcal{O}(n + k \cdot \log n) + \mathcal{O}(n) + \mathcal{O}(n \log n) + \mathcal{O}(n^2) = \mathcal{O}(n^2).$$

Como el algoritmo anterior nos regresa una triangulación de nuestro polígno con hoyos, entonces hemos mostrado lo requerido. □

**4.** Un árbol de rango en un conjunto de $n$ puntos en el plano requiere $\mathcal{O}(nlogn)$ de almacenamiento. Uno podría reducir los requisitos de almacenamiento almacenando estructuras asociadas solo con un subconjunto de los nodos en el árbol principal.

- Supongamos que sólo los nodos con profundidad $0, 2, 4, 6, \cdots$ tienen una estructura asociada. Muestre cómo se puede adaptar el algoritmo de consulta para responder consultas correctamente.
- Analice los requisitos de almacenamiento y el tiempo de consulta de dicha estructura de datos.

▷ **Solución:** Para este problema dividamos la solución en dos posibles opciones:

1. ¿Cómo adaptamos nuestro árbol de rangos para que cumpla lo requerido en el primer punto? Como sabemos que un árbol, al menos, tendrá una raíz y las estructuras estarán "colgadas" de niveles pares. Entonces, la información respecto de $Y$ de los niveles impares estarán contenidas en el árbol colgado en su nivel par, inmediato, anterior.

   Así, las consultas para los niveles pares se quedan exactamente iguales. Para los niveles impares consultamos respecto de $X$ y hacemos "bracktraking" al nivel par, inmediato, anterior para terminar la consulta en la estructura colgada en el nodo de ese nivel.

2. *Análisis de almacenamiento.* El almacenamiento, aunque a primera vista se reduce en la cantidad de niveles pares, realmente las estructuras que estaban colgadas en estos niveles ahora formarán parte de las estructuras en niveles pares. El ahorro de almacenamiento es $\mathcal{O}(1)$, pues solo nos ahorramos las raíces de los niveles impares (que son, normalmente, hojas de los niveles pares).

   *Análisis de tiempo de consulta.* El tiempo de consulta para un elemento en un nivel par es el mismo, para un elemento en un nivel impar es el mismo en términos globales, pues el "backtraking" lo realizamos en tiempo $\mathcal{O}(1)$ y la consulta en la estructura "colgada" sigue siendo $\mathcal{O}(\log n) + k$. Como la consulta en árbol respecto a $X$ (árbol grande) es $\mathcal{O}(\log n)$, concluimos una complejidad de consulta en $\mathcal{O}(\log^2 n)$

◁