Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Apellidos, Nombre | D.N.I.

Titulación: Grau d'Enginyeria Informàtica
Asignatura: Algorísmia
Duración: 2h 30m

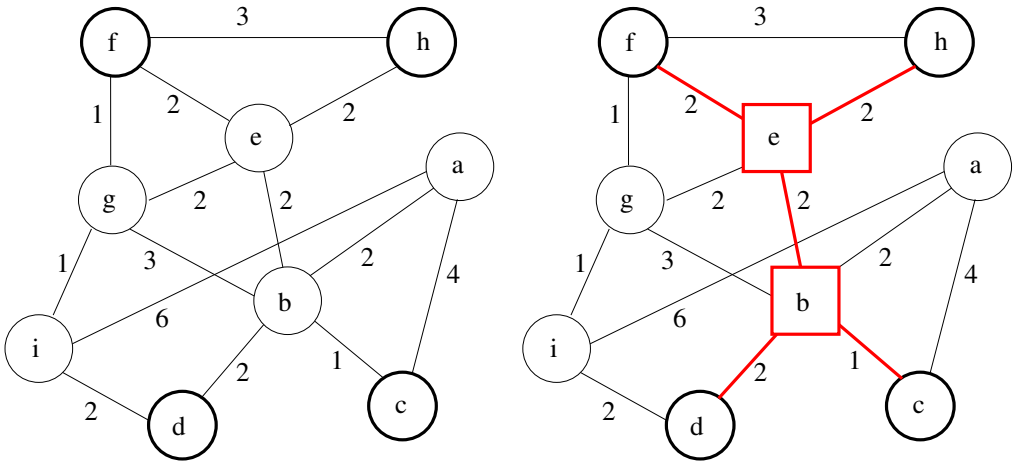
Curso: Q1 2012–2013 (Final)
Fecha: 22 de Enero de 2013

Instrucciones generales:

- Debéis escoger dos de las cinco preguntas. En el examen debéis indicar claramente cuáles son las preguntas elegidas. Si se contestáis más de dos preguntas entonces la nota será la suma de las dos notas máximas y además, si cada una de esas dos calificaciones supera 3 puntos sobre 5 (60% del peso de la pregunta) entonces las calificaciones de las otras preguntas contestadas se sumarán, en caso contrario serán desechadas.
- En cualquier caso, la calificación global de esta prueba escrita no podrá superar el 10, al sumar todos los puntos.

---

1. **(5 puntos)** Dado un grafo no dirigido $G = \langle V, E \rangle$ con pesos positivos $\omega : E \to \mathbb{R}^+$ en las aristas y un subconjunto de vértices $S \subseteq V$, un *árbol de Steiner* para S es un subgrafo conexo de peso mínimo que contiene los vértices en S y eventualmente otros vértices en $V \setminus S$, que se les conoce como vértices de Steiner. La figura siguiente muestra un grafo y a su derecha, resaltado en trazo grueso, el árbol de Steiner para $S = \{c, d, f, h\}$. Los vértices de Steiner, marcados mediante rectángulos, son b y e.



El problema del árbol de Steiner es, en general, NP-duro. Pero hay casos particulares sencillos. Por ejemplo, si $S = V$ entonces un árbol de expansión mínimo de G es un árbol de Steiner para S. Si $S = \{u, v\}$ entonces el árbol de Steiner es el camino de peso mínimo entre u y v.

Diseña un algoritmo voraz que proporcione un árbol que conecta los vértices del subconjunto S dado; dicho árbol no seá necesariamente de peso mínimo y por tanto no será un árbol de Steiner. Pero el algoritmo diseñado debería procurar hallar al menos una buena aproximación al árbol de Steiner. Calcula el coste de tu algoritmo en función de $n = |V|$ y $m = |E|$. Explica el algoritmo con el máximo detalle posible, las estructuras de datos auxiliares necesarias, etc.

No se considera válida una solución que devuelva un árbol de expansión siempre. Un buen punto de partida para una estrategia voraz es comenzar con el camino mínimo entre dos vértices de S.

SOLUCIÓN:

La estructura típica de un algoritmo voraz para obtener una aproximación al árbol de Steiner es la siguiente:

```
procedure GreedySteiner(G, A)
    A ← ∅
▷ A es un subgrafo de G
    eligir dos vértices u y v en S
    añadir las aristas del camino mínimo entre u y v a A
    while hay vértices en S que no están en V(A) do
        elegir un camino mínimo entre un vértice u ∈ V(A) y un vértice
        v ∈ S \ V(A)
▷ S \ V(A) son los vértices de S no conectados todavía
        añadir las aristas del camino mínimo entre u y v a A
    ...
```

Según la manera en que escogamos los vértices u y v al inicio, y en cada iteración del bucle principal, tendremos diferentes aproximaciones. En cualquier caso en cada iteración como mínimo añadiremos un vértice de S a $V(A)$, por lo que a lo sumo se harán $|S| - 2$ iteraciones del bucle principal. Podemos suponer que ya hemos calculado en un paso previo todos las distancias y caminos mínimos entre cada uno de los vértices de S y los restantes vértices del grafo: podemos hacer esto mediante $|S|$ llamadas al algoritmo de Dijkstra, de manera que $D[u, v]$ sea la distancia mínima entre los vértices u y v, y $CAM[u, v]$ nos dé el antecesor de v en el camino mínimo entre u y v (lo que nos permitirá reconstruir el camino mínimo completo). El coste del interior del bucle principal será en caso peor $\Omega(n)$ ya que reconstruir el camino mínimo entre u y v y añadirlo a A tiene coste en caso peor $\Theta(n)$; dependiendo del coste en caso peor del paso en que elegimos u y v, el coste del interior del bucle será lineal o mayor. Así que el coste en caso peor del bucle principal

será $\Omega(n \cdot |S|)$. A ello habremos de añadir el coste $O(|S|m \log n)$ de las $|S|$ llamadas al algoritmo de Dijkstra.

En la elección de u y v tendremos dos subconjuntos de candidatos P y Q: $P \subseteq V(A)$ contiene los vértices de $V(A)$ susceptibles de ser elegidos como u; $Q \subseteq S \setminus V(A)$ contiene los vértices como susceptibles de ser elegidos como v. Y aplicando el espíritu del esquema voraz el par (u, v) elegido debería ser aquél tal que $D[u, v]$ es mínima entre todos los pares en $P \times Q$. Usando estructuras de datos apropiadas, en particular un min-heap, el coste de hallar el par (u, v) será $O(\log(n \cdot |S|)) = O(\log n)$. Mantener actualizada la estructura de datos tras cada iteración tendrá coste global acumulado $O(n \log(n \cdot |S|)) = O(n \log n)$, porque podemos llegar a agregar hasta n vértices a $V(A)$. Teniendo todo esto en cuenta, y puesto que $|S| \leq n$, el coste en caso peor del algoritmo es

$$O(|S|m \log n) + O(n \cdot |S|) + O(n \log n).$$

Parece lógico tomar siempre $Q = S \setminus V(A)$, pero no es así con P. Según nuestra elección del conjunto P tendremos diferentes situaciones; vamos a considerar dos posibilidades:

- $P = V(A)$: el camino elegido arranca de un vértice cualquiera de $V(A)$, lo cual garantiza que A es acíclico en todo momento. En efecto, puesto que los pesos son positivos, si al agregar las aristas del camino mínimo entre u y v a A se creara un ciclo en A, llegaríamos a la conclusión absurda de que el camino agregado no es mínimo.
- $P = S \cap V(A)$: el camino arranca de un vértice de $V(A)$ que también es de S; no se garantiza que el camino elegido no crea un ciclo en A. Si optamos por esta variante, entonces será necesario agregar en cada iteración o después del bucle principal, la detección de ciclos en A y su eliminación, retirando la arista de mayor peso de cada ciclo. La ventaja de esta opción es que intenta incluir el mínimo de vértices que no son de S en A, al no considerar caminos que arrancan desde vértices intermedios (vértices de Steiner), que en definitiva son vértices que nos gustaría no necesitar. Por lo general, obtendrá mejores aproximaciones del árbol de Steiner, pero su coste será mayor, al necesitar detectar ciclos y eliminarlos (se puede hacer con coste $O(m \cdot n)$.

Otras opciones son posibles, naturalmente. Como ya se ha evidenciado con las dos variantes comentadas, tendremos compromisos entre la eficiencia de nuestros algoritmos y la calidad de la solución que nos proporcionan.

2. **(5 puntos)** Tenemos un algoritmo eficiente SELECT que nos permite hallar el k-ésimo menor elemento de un vector A de n elementos. Más específicamente, con $1 \leq i \leq k \leq j \leq n$ y un vector $A[1..n]$ no ordenado, la llamada

$$\text{SELECT}(A, i, j, k)$$

reorganiza el subvector $A[i..j]$ de manera que $A[k]$ contiene el elemento que corresponde si $A[i..j]$ estuviera ordenado (es decir, $A[k]$ contendrá el $(k-i+1)$-ésimo menor elemento de $A[i..j]$), y adicionalmente, todos los elementos en $A[i..k-1]$ son menores o iguales a

$A[k]$ y todos los elementos en $A[k+1..j]$ son mayores o guales que $A[k]$. El número medio de comparaciones que realiza el algoritmo es

$$j - i + 1 + \min(k - i + 1, j - k).$$

Con la llamada inicial SELECT$(A, 1, n, k)$ se obtiene el k-ésimo del vector $A[1..n]$ con coste promedio $\Theta(n + \min(k, n - k))$, que puede demostrarse que es óptimo.

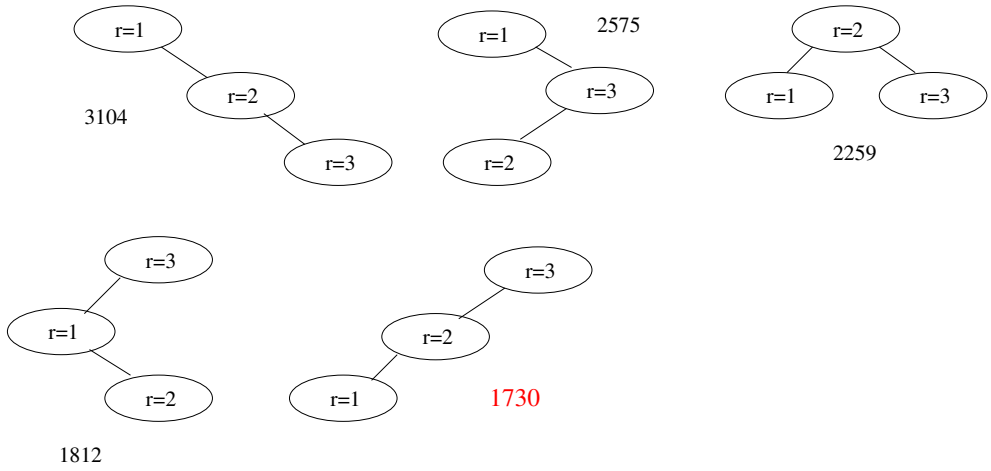
Ahora queremos utilizar SELECT para seleccionar múltiples rangos. Se nos da un vector $K = [k_1, \ldots, k_p]$, con $1 \leq k_1 < k_2 < \cdots < k_p \leq n$, y queremos un algoritmo que nos devuelva el k_1-ésimo menor elemento de A, el k_2-ésimo, etc. Podemos aprovechar que SELECT particiona el vector A en torno al k-ésimo, de manera que la versión recursiva de nuestro algoritmo de selección múltiple tendría la siguiente estructura:

procedure MULTIPLESELECT(A, i, j, K, ℓ, u)
▷ $A[i..j]$ contiene los elementos i-ésimo, …, j-ésimo de A, $1 \leq i \leq k_\ell < \cdots < k_u \leq j \leq n$, $1 \leq \ell \leq u \leq p$
  if $u < \ell$ **then**
▷ No hay rangos que buscar
  else
    $r \leftarrow$ un índice entre ℓ y u
    SELECT$(A, i, j, K[r])$
    MULTIPLESELECT$(A, i, K[r] - 1, K, \ell, r - 1)$
    MULTIPLESELECT$(A, K[r] + 1, j, K, r + 1, u)$

La eficiencia de MULTIPLESELECT dependerá de cómo se elige r en cada etapa recursiva. Si elegimos $r = \ell$ ó $r = u$ siempre, el coste promedio del algoritmo será $\mathcal{O}(pn)$. Si sistemaáticamente elegimos el rango medio ($r = (u + \ell)/2$) entonces el coste promedio será $\mathcal{O}(n \log p)$. Pero no necesariamente tomar el punto medio es lo mejor. En general, la elección de r óptima debe determinarse mediante un algoritmo de programación dinámica.

Por ejemplo, si $n = 1000$, y $K = [20, 117, 239]$ en la primera llamada recursiva tendremos tres elecciones: $r = 1$, $r = 2$ o $r = 3$. Supongamos que elegimos $r = 1$. Entonces hallaremos el vigésimo elemento de A ($K[1] = 20$) con $1000 + 20 = 1020$ comparaciones. A continuación la llamada recursiva por la izquierda no hace nada y en la llamada por la derecha podemos optar por escoger $r = 2$ ó $r = 3$. Si escogemos $r = 2$, necesitaremos $980 + 98 = 1078$ comparaciones y en la llamada recursiva por la derecha tendremos que usar $r = 3$ y haremos $883 + 123 = 1006$ comparaciones. En total, $1020 + 1078 + 1006 = 3104$ comparaciones. Si escogieramos en primer lugar $r = 2$ haríamos $1000 + 117 = 1117$ comparaciones y a continuación hariamos llamadas recursivas por la izquierda (con $r = 1$) y por la derecha ($r = 3$), que harían $116 + 20 = 136$ y $883 + 123 = 1006$ comparaciones respectivamente; en total: $1117 + 136 + 1006 = 2159$ comparaciones.

Las elecciones de las r en cada etapa pueden representarse de manera natural con un árbol binario, y cada uno tiene asociado tiene el número medio de comparaciones que hará el algoritmo de selección múltiple si se hacen las elecciones según indica el árbol binario. En nuestro ejemplo usando la mejor elección necesitaremos sólo 1812 comparaciones.



Diseñad un algoritmo de programación dinámica que dado n y el vector K nos devuelva la forma óptima de escoger r en cada etapa, es decir, el árbol binario con menor número medio de comparaciones asociado.

Determinad su coste en espacio y tiempo en función de p (el tamaño del vector K que se recibe como entrada).

Nota: Conviene pensar que el vector K contiene dos centinelas: $K[0] = 0$ y $K[p+1] = n+1$ para escribir las recurrencias de manera homogénea.

SOLUCIÓN:

Denotamos $C_{\ell,u}$ el número mínimo de comparaciones que se tendrán que usar para seleccionar los rangos en $K[\ell..u]$. Naturalmente nos interesa $C_{1,p}$. Usaremos $R_{\ell,u}$ para el rango que optimiza $C_{\ell,u}$ (el valor de r en la raíz del subárbol óptimo). El coste de seleccionar un rango k_r con $\ell \leq r \leq u$ es

$$k_{u+1} - k_{\ell-1} - 1 + \min(k_r - k_{\ell-1}, k_{u+1} - 1 - k_r)$$

porque (recursivamente) todos los elementos con los rangos buscados $(k_\ell, \ldots, k_u)$ están en el subvector $A[k_{\ell-1}+1..k_{u+1}-1]$.

Veamos la recurrencia para $C_{\ell,u}$.

(a) $C_{\ell,u} = 0$ si $\ell > u$; en efecto, si $\ell > u$ significa que no hay rangos a seleccionar.

(b) Si $\ell \leq u$ entonces habremos de considerar qué valor r, $\ell \leq r \leq u$ minimiza el coste de seleccionar el k_r-ésimo y luego recursivamente seleccionar primero los elementos de rangos $k_\ell, \ldots, k_{r-1}$ y luego los elementos de rangos $k_{r+1}, \ldots, k_u$.

$$\begin{aligned} C_{\ell,u} &= \min_{\ell \leq r \leq u} \Big(k_{u+1} - k_{\ell-1} - 1 + \min(k_r - k_{\ell-1}, k_{u+1} - k_r - 1) + C_{\ell,r-1} + C_{r+1,u} \Big) \\ &= k_{u+1} - k_{\ell-1} - 1 + \min_{\ell \leq r \leq u} \Big(\min(k_r - k_{\ell-1}, k_{u+1} - k_r - 1) + C_{\ell,r-1} + C_{r+1,u} \Big). \end{aligned}$$

Con la recurrencia para $C_{\ell,u}$ el algoritmo es bastante sencillo y se obtiene de manera rutinaria; es muy similar al algoritmo para la construcción de árboles binarios de búsqueda óptimos. En su versión más simple su coste es $\mathcal{O}(p^3)$. El coste en espacio es $\mathcal{O}(p^2)$. Se puede aplicar la optimización de Knuth y reducir el coste del algoritmo a $\mathcal{O}(p^2)$, ya que

$$R_{\ell,u-1} \leq R_{\ell,u} \leq R_{\ell+1,u}.$$

3. **(5 puntos)** Tenemos una red s-t con capacidades en los arcos y nuestro objetivo es hallar un flujo máximo; pero ahora hay *pérdidas* tanto en los arcos como en los vértices. En un arco $e = (u, v)$ con capacidad $c_e > 0$ y coeficiente de pérdida λ_e, $0 < \lambda_e \leq 1$, podrá salir procedente de u un flujo $f(e) \leq c_e$, pero el flujo que llega a v es, a lo sumo, $\lambda_e \cdot f(e)$. Por otro lado, para todo vértice $v \notin \{s, t\}$, el flujo saliente f' de v no será igual al flujo entrante f en v: será a lo sumo $\vartheta_v \cdot f$, donde ϑ_v, $0 < \vartheta_v \leq 1$, es el coeficiente de pérdida del vértice v.

   Escribid un programa lineal para resolver esta variante del problema de máximización del flujo. Los datos del problema son la red, las capacidades c_e y los coeficientes de pérdida λ_e y ϑ_v. Indicad claramente cuántas variables tiene el problema y cuál es su significado, cuál es la función objetivo (¿qué significan los coeficientes en la función?), y cuáles son las igualdades y desigualdades que conforman las restricciones, explicando en palabras lo que significan. ¿Cuáles son los coeficientes $a_{i,j}$ y b_i en la formulación matricial del programa lineal? Si la red tiene m arcos y $n + 2$ vértices (n vértices internos), ¿cuántas ecuaciones/inecuaciones tiene el programa lineal?

SOLUCIÓN:

Tendremos $2m$ variables: para cada arco $e = (u, v)$, x_e denotará el flujo que se inyecta en e desde u, y x'_e el flujo que llega a v.

La función objetivo es la suma de los flujos que llegan al sumidero:

$$x'_{e_1} + \cdots + x'_{e_k}$$

donde $e_1, \ldots, e_k$ son los arcos que inciden en t. Si queremos escribir la función objetivo como producto escalar de dos vectores $\vec{u} = (u_1, \ldots, u_{2m})$ y $\vec{x} = (x_1, \ldots, x_m, x'_1, \ldots, x'_m)$ entonces los primeros m coeficientes $u_i = 0$ y para los siguientes tenemos que $u_{i+m} = 0$ si e_i no es un arco que llega al sumidero t y $u_{i+m} = 1$ si e_i es un arco que llega a t. Entonces la función objetivo es $\vec{u}^{\mathrm{T}} \cdot \vec{x}$.

Después tenemos:

(a) $2m$ desigualdades de positividad: todos los flujos entrantes y salientes en los arcos son positivos,

$$x_e \geq 0, x'_e \geq 0, \qquad e \in E$$

(b) m condiciones de capacidad: los flujos entrantes en los arcos no pueden exceder las capacidades respectivas,

$$x_e \leq c_e$$

(c) m condiciones de pérdida en los arcos:

$$x'_e \leq \lambda_e \cdot x_e \Rightarrow x'_e - \lambda_e x_e \leq 0$$

(d) n desigualdades de flujo con pérdida, una por vértice interno: el flujo saliente del vértice v no excede ϑ_v veces el flujo entrante; el flujo entrante en v es $x'_{e_1} + \cdots + x'_{e_j}$ siendo $e_1, \ldots, e_j$ los arcos que llegan a v; el flujo saliente de v es $x_{w_1} + \cdots + x_{w_\ell}$ siendo $w_1, \ldots, w_\ell$ los arcos que salen de v

$$x_{w_1} + \cdots + x_{w_\ell} \leq \vartheta_v \left(x'_{e_1} + \cdots + x'_{e_j}\right) \Rightarrow$$
$$x_{w_1} + \cdots + x_{w_\ell} - \vartheta_v \left(x'_{e_1} + \cdots + x'_{e_j}\right) \leq 0$$

En la forma matricial tendremos n filas de la forma

$$a_{v,1}x_1 + \cdots + a_{v,m}x_m + a_{v,m+1}x'_1 + \cdots + a_{v,2m}x'_m \leq 0$$

donde, para $1 \leq i \leq m$, $a_{v,i} = 0$ si e_i no sale de v, $a_{v,i} = 1$ si e_i sale de v, $a_{v,m+i} = 0$ si e_i no llega a v y $a_{v,m+i} = -\vartheta_v$ si e_i llega a v.
Si se escribe el programa lineal en forma estándar

$$\begin{aligned} \max \quad & \vec{u}^{\mathrm{T}} \cdot \vec{x} \\ & \mathbf{A} \cdot \vec{x} \leq \vec{b} \\ & \vec{x} \geq \vec{0}, \end{aligned}$$

la matriz $\mathbf{A}$ tiene $2m + n$ filas y $2m$ columnas; sus m primeras filas corresponden a las condiciones de capacidad y el valor $b_i = c_{e_i}$; las siguientes m filas corresponden a las condiciones de pérdida y son de la forma

$$-\lambda_e x_e + x'_e \leq 0$$

Las últimas n filas dan las condiciones de flujo con pérdida en cada vértice y ya se han comentado más arriba.

4. **(5 puntos)** Como es bien sabido el coste en caso peor del algoritmo de ordenación por inserción es $\Theta(n^2)$. Por el contrario, el coste en caso mejor es $\Theta(n)$. ¿Dónde cae el coste promedio? ¿Es lineal? ¿Es cuadrático? ¿Está en un punto intermedio, por ejemplo, $\Theta(n\sqrt{n})$? Veamos cómo dar respuesta a estas preguntas.

   El coste del algoritmo es proporcional al número de *inversiones* en el vector, de manera que para calcular el coste promedio deberemos calcular el número promedio de inversiones. Recordemos que un par (i, j) con $i < j$ es una inversión en el vector A si $A[i] > A[j]$.

   Para determinar el número medio de inversiones en una permutación aleatoria de tamaño n convendrá utilizar las siguienetes variables aleatorias indicadoras:

$$X_{ij} = \begin{cases} 1 & \text{si } (i, j) \text{ es una inversión,} \\ 0 & \text{en caso contrario.} \end{cases}$$

Naturalmente se cumple que la variable aleatoria "número de inversiones"

$$I_n = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

(a) Calculad la probabilidad de que $X_{ij} = 1$ en una permutación aleatoria.

(b) A partir del apartado anterior, hallad $\mathbb{E}\{I_n\}$.

Justificad vuestras respuestas en los dos apartados.

SOLUCIÓN:
Primero vamos a calcular la probabilidad de que en una permutación aleatoria σ el par (i, j) sea una inversión, es decir, que $X_{i,j} = 1$ para $i < j$. Intuitivamente esta probabilidad es $1/2$. En efecto:

$$\begin{aligned}\Pr\{X_{ij} = 1\} &= \sum_{\ell=1}^{n} \Pr\{\sigma(i) = \ell \wedge \sigma(j) < \ell\} \\ &= \sum_{\ell=1}^{n} \Pr\{\sigma(i) = \ell\} \cdot \Pr\{\sigma(j) < \ell\} \\ &= \sum_{\ell=1}^{n} \frac{1}{n} \frac{\ell - 1}{n-1} = \frac{1}{n(n-1)} \sum_{\ell=1}^{n} (\ell - 1) \\ &= \frac{1}{n(n-1)} \sum_{\ell=0}^{n-1} \ell = \frac{1}{n(n-1)} \frac{n(n-1)}{2} = \frac{1}{2}.\end{aligned}$$

Para una variable indicadora X_j,

$$\mathbb{E}\{X_{ij}\} = \Pr\{X_{ij} = 1\} \cdot 1 + \Pr\{X_{ij} = 0\} \cdot 0 = \frac{1}{2}.$$

Y por la linealidad de las esperanzas

$$\begin{aligned}\mathbb{E}\{I_n\} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}\{X_{ij}\} &= \frac{1}{2} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 \\ &= \frac{1}{2} \sum_{i=1}^{n-1} (n - i) = \frac{1}{2} \sum_{i=1}^{n-1} i = \frac{1}{2} \frac{n(n-1)}{2} = \frac{n(n-1)}{4} \sim \frac{n^2}{4}.\end{aligned}$$

5. **(5 puntos)** Tenemos un conjunto de n puntos bidimensionales almacenados en un *K-d tree* T. Se nos da también un polígono convexo P, especificado como una lista de vértices donde el primero es el punto con coordenada x mínima (y en caso de empate de coordenada y mínima) y a continuación aparecen los restantes vértices en el orden horario.

   Escribe una función en C++ que devuelva el subconjunto de puntos de T que están en el interior de P. Utiliza las definiciones que se dan a continuación.

```
struct punto {
   double x, y;
}
typedef list<punto> poligono;

struct nodo {
   punto p;
   int discr;
   nodo* izq;
   nodo* der;
}
typedef nodo* kdtree;

// función que tenéis que implementar
void dentro_poligono(const kdtree& T, const poligono& P,
                     list<punto>& resultado);
```

Escribid en primer lugar la implementación de `dentro_poligono` suponiendo que tenéis una función

```
bool es_interior(const poligono& P, punto q)
```

Escribid después el código de la función `es_interior`. ¿Qué coste tiene `es_interior` para un polígono de m vértices?

¿Qué cambios podemos hacer en la representación de los polígonos que permita mejorar la eficiencia (al menos en promedio) de la función `es_interior`? Escribe la nueva representación del tipo o clase `poligono` en C++. Podemos suponer que tenemos una función

```
void crea_poligono(const list<punto>& L, poligono& P)
```

que a partir de la lista de vértices genera la estructura de datos P, que contendrá la información sobre los vértices, pero también la información adicional que puede ser útil para mejorar la eficiencia del algoritmo `dentro_poligono`. O si se define una clase `poligono`, la constructora de la clase recibiría la lista de los vértice L. ¿Cuál es el coste de la función `crea_poligono` si el polígono tiene m vértices? Una vez creado el `poligono`, la función `dentro_poligono` recibe éste, no una lista de vértices "pelada".

SOLUCIÓN:

Vamos a dar directamente una solución en el que se mejora la representación de los polígonos para hacer más eficiente el algoritmo. Dada la lista de vértices del polígono (con el convenio habitual), calculamos en primer lugar y de manera muy simple el rectángulo envolvente R. La esquina inferior izquierda es $(x_{\min}, y_{\min})$ siendo $x_{\min}$ la menor coordenada x de entre todos los vértices del polígono (de hecho es la coordenada x del primer vértice de la lista!) e $y_{\min}$ la menor coordenada y de entre todas las coordenadas y de los vértices.

De manera similar, la esquina superior derecha $(x_{\max}, y_{\max})$ está determinada por $x_{\max}$, la mayor coordenada x de todos los vértices, e $y_{\max}$, la mayor coordenada y de todos los vértices. El cálculo de R resulta muy útil para mejorar la eficiencia de `dentro_poligono`: si un punto $p = (x, y)$ no está en el interior de R entonces no puede estar en el interior del polígono. Para verificar si p está en el interior o no de R basta con comprobar que $x_{\min} \leq x \leq x_{\max}$ **y** que $y_{\min} \leq y \leq y_{\max}$. Así, por ejemplo, en el código más abajo si `P.xmin > x` entonces el resto de condiciones ya no se comprueba y el test retorna falso de inmediato.

Pero además el rectángulo envolvente nos servirá para guiar la búsqueda en el K-d tree y obtener una mejora **substancial** en el coste promedio de `dentro_poligono`. Supongamos que la raíz del subárbol que estamos explorando contiene $p = (x, y)$ y que discrimina respecto a la coordenada x. Entonces si $x_{\max} < x$ sólo hace falta explorar el subárbol izquierdo, puesto que ningún punto del subárbol derecho puede estar en el interior de R. Y si $x < x_{\min}$ entonces sólo hace falta explorar el subárbol izquierdo y no el derecho. Si el nodo discrimina respecto a y entonces podemos razonar de manera análoga. En definitiva el algoritmo es exactamente el de búsqueda en rangos ortogonales (el rectángulo es R), pero en vez de chequear si la raíz está o no dentro de R, chequeamos si está o no dentro del polígono. Como se vió en clase, el coste promedio va a ser sólo levemente superior a $\Theta(\sqrt{n})$ (para ser más precisos, será $\Theta(n^{0.56})$ en K-d trees estándar).

```
struct poligono {
   list<punto> vertices; // segun el convenio usual
   double xmin, xmax, ymin, ymax; // coordenadas del
                                  // rectangulo envolvente

   // constructora para crear poligonos
   poligono(const list<punto>& L, poligono& P) {
     vertices = L;
     ... // calculo trivial con coste O(|L|)
         // de xmin, xmax, ymin e ymax
   }
};

void dentro_poligono(const kdtree& T, const poligono& P,
                     list<punto>& resultado) {
     if (T == NULL) return;
     if (T -> discr == 0) { // discriminamos por x
        double x = T -> p.x;
        if (P.xmin <= x and x <= P.xmax
            and es_interior(P.vertices, T -> p))
           resultado.push_back(T -> p);
        if (P.xmin <= x)
           dentro_poligono(T -> izq, P, resultado);
        if (x <= P.xmax)
```

```
            dentro_poligono(T -> der, P, resultado);
        } else { // T -> discr == 1
                 // codigo analogo con las y's
           ...
        }
}
```

En vez del test

```
if (P.xmin <= x and x <= P.xmax
    and es_interior(P.vertices, T -> p))
```

podemos escribir

```
if (es_interior(P, T -> p))
```

si la función `es_interior` se aplica sobre la nueva representación de los polígonos y ya incorpora la mejora que usa el rectángulo envolvente.

Si el rectángulo envolvente contiene al punto tendremos que afinar para saber si el punto realmente está en el interior del polígono o no. Puede hacerse de varias formas: en definitiva, se trata de comprobar que la secuencia de ángulos que subtiende el punto p con los sucesivos vértices del polígono completa un giro de 360º. Asumimos que la función `angulo`(p, q) nos devuelve el ángulo que forma el segmento que va de q a p con respecto al eje de abcisas, estando dicho ángulo entre $-\pi$ y π.

```
bool es_interior(const poligono& P, punto q) {
  if (q.x < P.xmin or P.xmax < q.x) return false;
  if (q.y < P.ymin or P.ymax < q.y) return false;
  double sumang = 0.0
  double angprev = angulo(P.vertices[0], q);
  for (int k = 1; k < P.vertices().size(); ++k) {
    double ang = angulo(P.vertices[k], q);
    sumang += abs(ang-angprev); // abs(x) = valor absoluto de x
    angprev = ang;
  }
  return sumang == 2 * pi;
}
```

El coste en caso peor de la función sigue siendo $\Theta(|P|)$, done $|P|$ denota el número de vértices del polígono. Pero muchos puntos quedarán excluídos gracias a los dos tests sencillos del comienzo, ya que si un punto cumple alguno de ellos significa que el punto **no** está en el interior del rectángulo envolvente y en consecuencia **no** está en el interior del polígono.