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 2013–2014 (Final)
Fecha: 20 de Enero de 2014

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)** Escribe un algoritmo de divide y vencerás que dado un vector $A[1..n]$ y un valor k, $1 \leq k \leq n$, nos devuelve el subvector central de tamaño k con los elementos correspondientes ordenados ascendentemente. Por ejemplo, si

$$A = [30, 81, 95, 63, 54, 17, 1, 46, 22]$$

y $k = 5$ entonces el algoritmo debe devolver

$$A = [?, ?, 22, 30, 46, 54, 63, ?, ?],$$

esto es, $A[3..7]$ contiene el tercero, cuarto, ..., elementos en orden, los restantes elementos están en $A[1..2]$ y $A[8..9]$, distribuídos en cualquier orden (lo más probable es que el algoritmo haya movido a los elementos 1 y 17 al subvector $A[1..2]$ y a los elementos 81 y 95 los haya movido al subvector $A[8..9]$). Justificad la corrección del algoritmo. Vuestro algoritmo no debe ordenar el vector A; en particular su coste debe ser inferior a $\Theta(n \log n)$ si $k = o(n)$. Si $n - k$ es impar el subvector a la izquierda del subvector central tendrá un elemento más que el subvector a la derecha.

SOLUCIÓN:
Nuestra solución es una variación del algoritmo de quicksort. Disearemos un algoritmo recursivo

$$\text{ORDENASEGMENTO}(A, i, j, \ell, u)$$

que ordena el segmento $A[l..u]$ de $A[i..j]$, con $1 \leq i \leq j \leq n$; como precondición $A[i..j]$ contiene los elementos i-ésimo, $(i+1)$-ésimo, ..., j-ésimo del vector original.

procedure ORDENASEGMENTO(A, i, j, ℓ, u)

```
    if i > j then return
    end if
▷ si i ≤ j el subvector no es vacío
    PARTICIONAR(A, i, j, r)
▷ A[i..r − 1] ≤ A[r] ≤ A[r + 1..j]
    if ℓ ≤ r then
        ORDENASEGMENTO(A, i, r − 1, ℓ, min(r − 1, u))
    end if
    if r ≤ u then
        ORDENASEGMENTO(A, r + 1, j, max(ℓ, r + 1), u)
    end if
end procedure
```

Sea r la posición del pivot tras la partición. Si $\ell \leq r$ entonces ó todos los elementos del segmento $A[\ell..u]$ se encuentran en $A[i..r-1]$ (si $u < r$) ó una parte del segmento, concretamente $A[\ell..r-1]$, está en $A[i..r-1]$ (si $r \leq u$). Análogamente, algunos de los elementos buscados estarán en el subvector $A[r+1..j]$ si $r \leq u$; todos (si $r < \ell \leq u$) ó una parte (si $\ell \leq r \leq u$). Este algoritmo tiene coste promedio $\Theta(n \log k)$, por razones similares al coste $O(n \log p)$ que se obtiene para el algoritmo de multiselección que selecciona p elementos de rangos dados. En cualquier caso resulta obvio que no ordenará completamente el vector A (salvo que $k \approx n$) y que sólo hace dos llamadas recursivas si es estrictamente necesario.

Para resolver el problema original, bastará hacer la llamada inicial apropiada:

```
procedure ORDENACENTRALES(A, k)
    ℓ := ⌈(n − k)/2⌉ + 1; u := ℓ + k − 1
    ORDENASEGMENTO(A, 1, n, ℓ, u)
end procedure
```

2. **(5 puntos)** Dado un grafo no dirigido completo $G = \langle V, E \rangle$ donde cada arista (u, v) está etiquetada por la *distancia* $d(u, v) \geq 0$ entre los vértices u y v, y un entero $k \ll |V|$, disead un algoritmo voraz que encuentre una partición de V en k subconjuntos disjuntos $C_1, \ldots, C_k$ tal que la *separación* sea máxima. Se denomina clúster a cada uno de los subcojuntos C_i. La distancia inter-clúster $\text{dic}(C_i, C_j)$ es la mínima distancia entre un vértice $u \in C_i$ y un vértice $v \in C_j$:

$$\text{dic}(C_i, C_j) = \min_{u \in C_i, v \in C_j} d(u, v).$$

La separación $\text{sep}(C)$ de una k-partición $C = \{C_1, \ldots, C_k\}$ es

$$\text{sep}(C) = \min_{i,j:i \neq j} \text{dic}(C_i, C_j)$$

Se os pide, pues, la k-partición C^* de los vértices de G cuya separación es máxima, es decir

$$\text{sep}(C^*) = \max_{C \text{ es una } k\text{-partición}} \text{sep}(C).$$

Justifica la corrección de tu algoritmo y calcula su eficiencia.

Pista: puede ayudaros a responder este problema pensar en términos geométricos. Cada vértice es un punto en el plano, y $d(u, v)$ es la distancia (Euclídea) entre los puntos u y v. Se tratará por lo tanto de agrupar los puntos en k subconjuntos de manera que la separación sea máxima.

SOLUCIÓN:

Consideremos un árbol de expansión T del grafo G. Si eliminamos $k-1$ aristas del árbol, tendremos k componentes (cada una de las cuales es un árbol). Es decir, cada árbol de expansión y subconjunto de $k-1$ aristas del árbol define una k-partición del grafo en k clústers/componentes. Si el árbol de expansión es mínimo, entonces cada una las aristas eliminadas nos da la distancia interclúster entre los clústers C_i y C_j que estaban unidos para la arista en cuestión. En efecto, cualquier otra arista que un vértice de C_i con uno de C_j ha de tener una distancia/peso mayor o igual al de la eliminada, porque sino dicha arista sería la que estaría en el árbol de expansión mínimo y no la que hemos eliminado.

Si las $k-1$ aristas que escogemos para eliminar son las de mayor peso de un árbol de expansión mínimo, entonces la separación de la partición corresponde al peso de la menor de las $k-1$ aristas escogidas y por tanto será máxima.

El agoritmo que utilizaremos será una variación simple del algoritmo de Kruskal: en vez de terminar cuando se han encontrado $n-1$ aristas para formar un árbol de expansión, el algoritmo acaba cuando tenemos formadas k componentes. Inicialmente, empezamos con n componentes (cada vértice es una), y en cada iteración en la que encontramos una arista que une vértices en componentes distintas, se unen las dos componentes. El coste del algoritmo vendrá dominado por la fase inicial en la que ordenamos todas las aristas por peso, es decir, con coste $\Theta(|E| \log |V|) = \Theta(n^2 \log n)$, ya que $|E| = \Theta(n^2)$. Alternativamente podemos formar un heap de mínimos con las $|E|$ aristas, aunque el coste en caso peor seguirá siendo el mismo, pues tendríamos que hacer $\Theta(n^2 - k)$ iteraciones y cada una nos costaría $O(\log n)$ (para extraer la arista de peso mínimo del heap).

```
procedure MaxSepParticion(G, k)
    n := |V|
    Ordenar las aristas de G por distancia
    Particion P(n) ▷ crea un mfset del conjunto {1, ..., n}
    while P.NumParts() ≠ k do
        (u, v) := NextEdge()
        if P.Find(u) ≠ P.Find(v) then
            P.Merge(u, v)
        end if
    end while
▷ P contiene la partición
end procedure
```

A la salida del bucle principal, el mfset P representa la partición resultante. Si queremos

el resultado de alguna otra forma, es bastante sencillo pasar de una representación a otra con coste $\Theta(n)$.

3. **(5 puntos)** Uno de los subproblemas que necesitamos resolver en el juego de *Cazatesoros* es el siguiente: tenemos un grafo dirigido $G = \langle V, E \rangle$ donde en cada vértice hay un cofre que puede estar vacío o contener un cierto número de monedas; $t(v) \geq 0$ nos da el número de monedas (0 si el cofre está vacío) en el vértice v, y el objetivo es hallar camino(s) simple(s) que permiten recolectar el máximo número de monedas. Si un camino P pasa por los vértices $v_0, v_1, \ldots, v_k$ se recolectan

$$t(P) = t(v_0) + t(v_1) + \ldots + t(v_k)$$

monedas. A $t(P)$ le llamaremos *valor* del camino. Notad que no pueden repetirse vértices si el camino es simple. Escribid un algoritmo de programación dinámica que nos permita calcular para dos vértices dados u y v, el valor $t(P^*)$ del camino simple P^* de valor máximo entre u y v. Justifica su corrección y calcula su coste en espacio y tiempo. Describe cómo se puede resolver el siguiente problema adicional, usando el algoritmo anterior o una variación simple del mismo:

> Dado un vértice u, cuál es el máximo valor que se podrá recolectar yendo a otro vértice (o quedándose en u) y cuál es el número de arcos mínimo necesario para alcanzar dicho valor, dicho de otro modo, si hay varios caminos simples que parten de u y todos ellos recolectan el mismo valor, queremos saber cuál es la longitud mínima entre todos esos caminos.

SOLUCIÓN:

En esta solución usaremos la hipótesis (tal como se dijo durante el exámen) de que el grafo es acíclico. Esta hipótesis simplifica la tarea, pues todo camino en el grafo será simple. También ocurre que, bajo este supuesto de que el grafo es acíclico, el problema admite una solución algo más eficiente que no requiere la programación dinámica.

Nuestro algoritmo es bastante similar al algoritmo de Floyd para caminos mínimos. Sea $\tau_k(u, v)$ el máximo valor alcanzable en el mejor camino entre u y v que pase exclusivamente por vértices intermedios del conjunto $\{1, \ldots, k\}$.

Consideremos en primer lugar la base de la recursión, cuando $k = 0$. Si $u = v$ entonces $\tau_0(u, u) = t(u)$, el número de monedas en el cofre que hay en u y la mejor y única opción es simplemente el camino vacío. Si $u \neq v$ y entre u y v no hay una arco entonces $\tau_0(u, v) = t(u)$ porque no podemos usar ningún vértice intermedio. Finalmente si $(u, v) \in E$ entonces

$$\tau_0(u, v) = t(u) + t(v)$$

porque podremos usar el arco para ir de u a v y recolectar todas las monedas en ambos vértices.

Para $k > 0$, razonamos así: o bien el mejor camino no pasa por el vértice k y entonces su valor es $\tau_{k-1}(u, v)$, o bien el mejor camino consiste en recoger todas las monedas entre u y k y luego todas las monedas entre k y v, de manera que el valor es $\tau_{k-1}(u, k) + \tau_{k-1}(k, v) - t(k)$. El término $t(k)$ hay que sustraerlo porque si no estaría duplicado (se suma en $t(k)$ en $\tau_{k-1}(u, k)$ y en $\tau_{k-1}(k, v)$). Por lo tanto

$$\tau_k(u, v) = \max\big(\tau_{k-1}(u, v), \tau_{k-1}(u, k) + \tau_{k-1}(k, v) - t(k)\big).$$

Una vez tenemos la recurrencia el resto es bastante simple. Al igual que en el algoritmo de Floyd, $\tau_k(u, k) = \tau_{k-1}(u, k)$ y $\tau_k(k, v) = \tau_{k-1}(k, v)$, lo que nos permite usar una sola matriz T tal que al final de la iteración k-ésima, $T[u, v] = \tau_k(u, v)$ para toda u y v.

```
▷ T[1..n, 1..n] inicializada a 0
for all v := 1ton do
    T[v, v] := t(v)
end for
for all e = (u, v) ∈ E(G) do
    T[u, v] := t(u) + t(v)
end for
for k := 1 to n do
    for u := 1 to n do
        for v := 1 to n do
            if T[u, k] + T[k, v] − t(k) > T[u, v] then
                T[u, v] := T[u, k] + T[k, v] − t(k)
            end if
        end for
    end for
end for
```

El coste del algoritmo es $\Theta(n^3)$ en tiempo y $\Theta(n^2)$ en espacio. Al final del algoritmo $T[u, v]$ contiene la respuesta buscada.

Para la variante que se nos pide, debemos modificar levemente el algoritmo de manera que para cada u y v almacenemos en $L[u, v]$ la longitud del mejor camino entre u y v. $L[u, v]$ se inicializa a 1 para los vértices u y v entre los que hay un arco y a 0 en otro caso. En el bucle principal si $T[u, k] + T[k, v] - t(k) > T[u, v]$ se actualiza $T[u, v]$ y $L[u, v] := L[u, k] + L[k, v]$. Si $T[u, v] = T[u, k] + T[k, v] - t(k)$ entonces deberemos comparar cuál de los dos caminos (de igual valor) usa menos arcos: si $L[u, v] > L[u, k] + L[k, v]$ entonces se actualiza $L[u, v]$ porque el camino pasando por el vértice k tiene menor longitud. Los costes en espacio y en tiempo de esta variante son idénticos a los del algoritmo original. El valor máximo $t^*(u)$ que podemos recolectar en u es el mayor valor en la fila correspondiente de la matriz T,

$$t^*(u) = \max_v T[u, v]$$

y la longitud del camino correspondinte la hallaremos en el lugar correspondiente de la

matriz L. Esa última parte del cálculo tiene coste lineal: es buscar el máximo en una fila de una matriz $n \times n$.

4. **(5 puntos)** Un problema de los filtros de Bloom (BF) convencionales es que no admiten borrados. Explica, en primer lugar, porqué pasa esto.

   A continuación, propón una variante de los BFs que sí admita borrados: la denominaremos BFD. Tal variante utilizará más bits de memoria que el BF convencional. En principio, dado que un BF tiene una probabilidad no nula de falsos positivos, la operación de borrado en el BFD necesitará poder verificar si x está realmente presente o no consultando una estructura externa auxiliar. Podemos suponer que cuando se inserta un nuevo elemento se inserta tanto en el BFD como en la estructura auxiliar, y lo mismo ocurre cuando se borra. En las búsquedas, se chequea si x está en el BFD o no; si la respuesta es negativa, estamos seguros de que x no está, mientras que si la respuesta del BFD es afirmativa, podría tratarse de un falso positivo y tendremos que descartar esta posibilidad haciendo una búsqueda adicional en la estructura de datos auxiliar. Escribid todo el código C++ necesario para definir el BFD, excepto la parte correspondiente al hash: podéis asumir que disponéis de un vector de k funciones de hash. Tampoco hay que escribir el código correspondiente a la estructura de datos auxiliar.

```
template <class Elem>
class Hash {
public:
  int operator()(const Elem& x) const;
  // dado un objeto h de la clase Hash<Elem>,
  // h(x) nos devuelve el valor de hash para x
  ...
};

template <class Elem>
class BFD {
private:
   vector<Hash<Elem> > h;
   // este vector guarda las k funciones de hash del filtro;
   // h[i](x) nos devuelve el valor de hash de la funci\'on i-esima
   // para el Elem x
   ...
public:
   const int DEFAULT_SIZE = ...;
   BFD(int M = DEFAULT_SIZE); // constructora
void insert(const Elem& x);
void remove(const Elem& x);
bool contains(const Elem& x) const;
   ...
};
```

Justificad la corrección de la estructura propuesta para el BFD y calculad el coste de las operaciones de inserción, borrado y consulta en el BFD (no contabilizar el coste de las operaciones que se hagan sobre la estructura auxiliar).

Un análisis del coste en memoria del BFD detallado es complejo y queda fuera del alcance de la asignatura; no obstante, a modo de guía, si para un BF usasemos M bits, para un BFD de prestaciones equivalentes necesitaríamos $O(M \log \log n)$ bits, siendo n el número de elementos en el filtro.

Se valorará positivamente que determinéis la probabilidad de falso positivo en el BFD y que argumentéis intuitivamente porqué se necesitarían $O(M \log \log n)$ bits de memoria para el BFD.

SOLUCIÓN:

El filtro de Bloom convencional no admite borrados, ya que si queremos borrar un elemento x presente en el filtro (y verificado en la ED auxiliar) no podemos poner los bits de las posiciones $h_1(x), \ldots, h_k(x)$ a 0: otros elementos insertados en el BF pueden estar también poniendo a 1 algunos de esos bits, y si ponemos los bits a 0 entonces el BF respondería que dichos elementos no están en el BF, dando falsos negativos (los BF sólo tienen probabilidad no nula de falsos positivos).

La solución es simple: basta utilizar un vector F de enteros (de pocos bits cada uno) en vez de un vector de bits. Así, $F[i] = j$ indicará que j de los elementos insertados "hashean" en la posición i, es decir, existen exactamente j elementos x_1, x_2, ..., x_j insertados en el filtro tal que para cada uno de ellos alguna de las k funciones de hash toma el valor i, esto es, para cada de los x_r, $1 \leq r \leq j$, existe una función de hash h_s, $1 \leq s \leq k$, tal que $h_s(x_r) = i$. Cuando insertamos un elemento nuevo x incrementamos en uno las k posiciones $F[h_1(x)], \ldots, F[h_k(x)]$.

Para borrar un elemento x, comprobaremos si está o no (primero consultando el filtro y luego verificando con la ED auxiliar). Si el elemento está entonces decrementamos en 1 cada uno de los contadores indicados por las k funciones de hash y lo eliminamos también de la ED auxiliar.

```
typedef char byte;
template <class Elem>
class BFD {
private:
   vector<Hash<Elem> > h;
   vector<byte> F;
   BigDataStructure* D;
   ...
// operacion privada que comprueba si x esta en el filtro o no
// tiene probabilidad > 0 de falso positivo
bool lookup_filter(const Elem& x) const {
  int k = h.size();
```

```
  for (int i = 0; i < k; ++i)
      if (F[h[i](x)] == 0) return false;
  return true;
}

public:
void insert(const Elem& x) {
  int k = h.size();
  for (int i = 0; i < k; ++i) {
      ++F[h[i](x)];
  }
  D -> insert(x);
}

void remove(const Elem& x) {
  if (contains(x)) { // x esta seguro en el filtro
                     // ver el comentario en el metodo
                     // BFD::contains
       int k = h.size();
       for (int i = 0; i < k; ++i) {
         --F[h[i](x)];
       }
       D -> remove(x);
     }
  } // else x is not in the BFD
}

// devuelve cierto si y solo si x fue insertado en el BFD (y no
// ha sido eliminado); la llamada a D->contains(x) solo se hace
// si el filtro responde positivamente; entonces lo verificamos
// con la ED auxiliar
bool contains(const Elem& x) const {
  return lookup_filter(x) and D -> contains(x);
}
...
};
```

Si no tenemos en cuenta los costes de las operaciones sobre la ED auxiliar entonces todas las operaciones implementadas tienen coste $\Theta(1)$ respecto al número n de elementos que almacena (realmente) el BFD. Básicamente consiste en iterar sobre las k funciones de hash; tanto k como el coste de evaluar una función de hash son independientes del número de elementos en el BFD.

Desde el punto de vista de la memoria, razonamos así: tenemos una tabla en la que realizamos nk incrementos de contador. En promedio cada contador será incrementado $\alpha = nk/M$ veces y el tamaño del filtro se habrá escogido de manera que dicha α sea con-

stante (como en los BF convencionales). El contador que más incrementos recibe recibirá $\Theta(\log(nk)) = \Theta(\log n)$ incrementos con alta probabilidad (por la misma razón que la cadena más larga de sinónimos en una tabla de hash con encadenamientos separados tiene longitud $\Theta(\log n)$ con alta probabilidad). Eso quiere decir que bastarán contadores de $\Theta(\log \log n)$ con alta probabilidad. Con 5-6 bits por contador tenemos más que suficiente para cualquier finalidad práctica. En la implementación dada hemos usado bytes (= 8 bits) para simplificar.

Para el análisis de falso positivos, en primer lugar debemos tener en cuenta que tal como está diseñada la operación de borrado, el estado en el que queda el filtro tras un borrado es exactamente igual que el estado que tendría el filtro si el elemento en cuestión jamás hubiese sido insertado. Esto quiere decir que de cara al análisis podemos suponer que en el BFD sólo se han hecho n inserciones (y ningún borrado). La probabilidad de falso positivo en el BFD será entonces igual a la probabilidad de que los k contadores que corresponden a un cierto x sean > 0 sin haber insertado ese x; esto es aproximadamente igual a la probabilidad de que tras n inserciones, es decir, nk incremento aleatorios, los contadores tengan un valor > 0. Pero esa probabilidad es la misma que la de que los bits correspondientes estén a 1 en un BF convencional, porque el valor del contador no es relevante, lo que importa es si es 0 o si es > 0. De manera que la probabilidad de falso positivo es idéntica en el BF convencional y en el BFD.

5. **(5 puntos)** Dado un árbol de búsqueda K-dimensional (K-d tree), un índice j, $0 \leq j < K$ y un valor z, escribe un método en C++ que devuelva el número de elementos en el árbol cuya coordenada j-ésima es $\leq z$.

   Utiliza las siguientes definiciones en C++:

```
class kdtree {
private:
   struct node_kdtree {
      vector<double> x; // la clave k-dimensional del elemento
      Info v;           // informacion asociada a la clave
      node_kdtree* left;
      node_kdtree* right;
      int discr;  // coordenada por la cual se discrimina
      int size;   // tamaño del subarbol del cual
                  // es raiz el nodo
   };
   node_kdtree* root;
   int K;     // dimension de los puntos
   ...
public:
   ...
   // T.por_debajo(j, z) devuelve el numero de nodos en T tales que
   // x[j] <= z
   int por_debajo(int j, double z) const;
```

```
    ...
}
```

No se pide que calculéis la eficiencia del método implementado; pero debe evitar visitar más nodos de los necesarios. Aprovechad el atributo `size` que hay en cada nodo para no tener que visitar subárboles a no ser que sea necesario. Justifica su corrección. Argumenta que el coste de la operación `por_debajo` es equivalente al de una operación de *partial match* en la que sólo se especifica un atributo.

SOLUCIÓN:
Emplearemos un método privado que recibe explícitamente un puntero p al nodo raíz del subárbol para el cual se calcula el número de elementos tales que $x[j] \leq z$.

```
class kdtree {
private:
...
// metodo privado de la clase
static int por_debajo(nodo_kdtree* p, int j, double z);
...
public:
...
int por_debajo(int j, double z) const {
  return por_debajo(root, j, z);
};
...
}
```

El algoritmo es muy sencillo y de hecho los nodos que visita son los mismos que visitaría un *partial match* con la *query* $q = (*, *, \ldots, z, *, \ldots, *)$ en la que únicamente se especifica la coordenada j.

Si el subárbol es vacío, el puntero será nulo y el resultado ha de ser 0. Si el subárbol no es vacío, sea i el discriminante del nodo raíz y sea x el punto almacenado en el nodo raíz. Si $i \neq j$ entonces puede haber puntos con coordenada $j \leq z$ tanto en el subárbol izquierdo como en el derecho y tendremos que hacer las llamadas recursivas a uno y otro lado y sumar los conteos obtenidos. Pero si $i = j$ entonces si $z < x_j$ sólo puede haber puntos con coordenada j inferior a z en el subárbol izquierdo. Si $x_j \leq z$ entonces todos los nodos del subárbol izquierdo, el nodo raíz y quizás algunos del subárbol derecho cumplen la condición; el atributo `size` nos dará el número de nodos en el subárbol izquierdo (si no es vacío) y sólo necesitaremos hacer una llamada recursiva en el subárbol derecho para saber cuántos nodos de ese subárbol cumplen la condición.

```
int tam(nodo_kdtree* p) {
    return (p == NULL) ? 0 : p -> size;
}
```

```
int kdtree::por_debajo(nodo_kdtree* p, int j, double z) {
  if (p == NULL) return 0;
  int i = p -> discr;
  if (i != j) {
     int inc = (p -> x[j] <= z) ? 1 : 0;
     return inc + por_debajo(p -> left, j, z)
                + por_debajo(p -> right, j, z);
  }  else {
     if (z < p -> x[j])
        return por_debajo(p -> left, j, z);
     else
        return tam(p -> left) + 1 + por_debajo(p -> right, j, z);
  }
}
```