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: Q2 2012–2013 (Final)
Fecha: 5 de Junio 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)** Se dice que un intervalo $I = [a, b]$ *cubre* un punto $x \in \mathbb{R}$ si $a \leq x \leq b$. Un intervalo de longitud 1 ($b = a + 1$) se dice que es *unitario*. Diseña un algoritmo voraz que dado un conjunto $X = \{x_1, \ldots, x_n\}$ de puntos, determine el mínimo número necesario de intervalos unitarios para cubrir todos los puntos de X. Calcula el coste del algoritmo en función de n y justifica su corrección.

SOLUCIÓN:
El algoritmo voraz ordena en primer lugar el conjunto X de menor a mayor, de manera que $x_1 < x_2 < \cdots < x_n$. A continuación recorre el vector de izquierda a derecha, creando un nuevo intervalo sólo cuando el punto x_i en curso queda fuera del último intervalo.

```
procedure RECUBRIRVORAZ(X)
    Ordenar X en orden creciente
    R ← ∅; i ← 1; n ← |X|
    while i ≠ n + 1 do
        a ← x_i; b ← a + 1; R ← R ∪ {[a, b]}
        while i ≠ n + 1 ∧ x_i ≤ b do
            i ← i + 1
        end while
    end while
    return R
```

end procedure

El coste del algoritmo viene dominado por el coste de la ordenación $\Theta(n \log n)$; el bucle principal tiene sólo coste $\Theta(n)$.

En primer lugar, resulta obvio que la solución retornada por el algoritmo voraz recubre todos los puntos de X. Para argumentar que el conjunto de intervalos retornado es de cardinalidad mínima, supongamos que

$$R^{\text{opt}} = \{I_1^{\text{opt}}, I_2^{\text{opt}}, \ldots\}$$

es una solución óptima, y sin pérdida de generalidad, supongamos que los intervalos están ordenados de menor a mayor según su extremo inferior ($a_1 < a_2 < \ldots$). La solución voraz

$$R^{\text{greedy}} = \{I_1^{\text{greedy}}, I_2^{\text{greedy}}, \ldots\}$$

cumple también esta propiedad. Supongamos que $R^{\text{opt}} \neq R^{\text{greedy}}$ (en otro caso, la demostración estaría completada). Si recorremos R^{opt} y R^{greedy} en paralelo, habrá un cierto j tal que

$$I_j^{\text{opt}} = [a_j^{\text{opt}}, b_j^{\text{opt}}] \neq I_j^{\text{greedy}} = [a_j^{\text{greedy}}, b_j^{\text{greedy}}].$$

Por construcción, los primeros $j-1$ intervalos de las dos soluciones cubren los mismos puntos de X, digamos $\{x_1, \ldots, x_p\}$ y x_{p+1} es el menor punto de X todavía sin recubrir. Entonces $I_j^{\text{greedy}} = [x_{p+1}, x_{p+1}+1]$. Por otro lado, para $I_j^{\text{opt}} = [a_j^{\text{opt}}, b_j^{\text{opt}}]$ tiene que cumplirse necesariamente que $a_j^{\text{opt}} < x_{p+1}$, porque si $a_j^{\text{opt}} = x_{p+1}$ entonces el intervalo j-ésimo sería igual para ambas soluciones y si $a_j^{\text{opt}} > x_{p+1}$ entonces el punto x_{p+1} no quedaría recubierto en R^{opt}. Dado que $a_j^{\text{opt}} < a_j^{\text{greedy}} = x_{p+1}$, es obvio que la solución óptima queda "rezagada" respecto a la voraz y no podrá tener menos intervalos. Dicho de otro modo, podemos modificar la solución óptima de manera que no cambie el número de intervalos, pero sea igual a la voraz hasta el intervalo j-ésimo (substituyendo I_j^{opt} por I_j^{greedy}) y entonces volvemos a aplicar el mismo razonamiento hasta que finalmente tenemos $R^{\text{opt}} = R^{\text{greedy}}$.

2. **(5 puntos)** El *diámetro* de un grafo no dirigido conexo es la distancia máxima entre dos de sus vértices. La distancia entre dos vértices u y v del grafo es la longitud del camino más corto entre u y v. Escribe un algoritmo que calcule el diámetro de un grafo conexo $G = \langle V, E \rangle$ dado. Calcula el coste del algoritmo en función de $n = |V|$ y $m = |E|$ y justifica su corrección.

SOLUCIÓN:

Denotemos $\text{dist}(u, v)$ la distancia entre dos vértices u y v en G, es decir, el número de aristas en el camino de menor longitud entre u y v. Puesto que G es conexo, $\text{dist}(u, v) < +\infty$ para todo par de vértices. Por definición

$$\text{diam}(G) = \max_{u,v \in V} \text{dist}(u, v).$$

Sea $\Delta(u)$ la mayor distancia en el grafo entre el vértice u y otro de los vértices:

$$\Delta(u) = \max_{v \in V} \text{dist}(u, v).$$

Entonces

$$\text{diam}(G) = \max_{u \in V} \Delta(u).$$

La solución que proponemos utiliza un recorrido en anchura para calcular $\Delta(u)$ con cada uno de los vértices u del grafo, y se queda con el máximo, que será el diámetro del grafo. Puesto que el coste del recorrido en anchura en $\Theta(n+m)$ y hacemos n recorridos, el coste total es $\Theta(n(n+m))$. Como el grafo es conexo tenemos $m \geq n-1$ y por lo tanto el coste del algoritmo es $\Theta(nm)$.

```
procedure DIAM(G)
    maxD ← 0
    for u ∈ V(G) do
        D ← MAXDIST(G, u)
        if D > maxD then
            maxD ← D
        end if
    end for
    return maxD
end procedure
procedure MAXDIST(G, u)
▷ retorna Δ(u)
    for v ∈ V do
▷ dist[v] = +∞ indica que la distancia entre u y v
▷ es desconocida
        dist[v] ← +∞
    end for
    Q ← ∅; Q.PUSH(u);
    dist[u] ← 0
    while Q ≠ ∅ do
        v ← Q.POP()
        for w ∈ ADYACENTES(G, v) do
            if dist[w] = +∞ then
                dist[w] ← dist[v] + 1
                Q.PUSH(w)
            end if
        end for
    end while
    return max_{v∈V} dist[v]
end procedure
```

3. **(5 puntos)** Dado un string de longitud n en el que sólo aparecen letras, escribe un algoritmo de programación dinámica que determine si constituye un texto válido si se insertaran espacios en los lugares apropiados, es decir, si el string es una concatenación de palabras existentes en un cierto diccionario dado. Por ejemplo, para el string

"conciencañonesporbandavientoenpopa"

el algoritmo debería respondernos que es un texto válido (en castellano). Se puede asumir que tenemos una función `dict(s,i,j)` que devuelve cierto si y sólo si el substring $s[i..j]$ es una palabra del diccionario. El coste de la llamada es $\Theta(1)$. Calcula el coste del algoritmo en función de n. Si el string es válido, indica cómo se puede reconstruir el texto original.

SOLUCIÓN:

El primer paso es hallar una recurrencia que exprese la solución al problema. Sea $V_{i,j} =$ cierto si y sólo si el substring $s[i..j]$ es un texto válido (insertando los espacios necesarios, si fuera el caso). La base de la recurrencia es $V_{i,i-1} =$ cierto, ya que la cadena vacía es siempre un texto válido. Si $i \leq j$ entonces $V_{i,j}$ será cierto si $s[i..j]$ es una palabra en el diccionario (`dict(s,i,j)` retorna cierto) ó si existe alguna k, $i \leq k < j$, tal que $V_{i,k} =$ cierto y $V_{k+1,j} =$ cierto, es decir, si insertando un espacio tras la letra k-ésima los textos a izquierda y derecha de ese espacio son válidos. Simbólicamente:

$$V_{i,j} = \begin{cases} \text{cierto}, & \text{si } j < i, \\ \texttt{dict(s,i,j)} \vee \exists k : i \leq k < j : (V_{i,k} \wedge V_{k+1,j}), & \text{si } i \leq j. \end{cases}$$

El algoritmo de programación dinámica simplemente rellena la matriz V en base a la recurrencia anterior: hay que rellenar de abajo ($i = n$) a arriba ($i = 1$) y de izquierda a derecha. La casilla $V[1, n]$ contiene la respuesta buscada. El coste en espacio es claramente $\Theta(n^2)$ y el coste en tiempo es en caso peor $\Theta(n^3)$, pues para cada entrada tendremos el bucle sobre la k, que va de i a $j-1$, aunque "cortocircuitamos" el bucle tan pronto hallásemos un valor de k tal que $V_{i,k} \wedge V_{k+1,j} =$ cierto.

```
procedure EsTextoValido(s)
▷ V[0..n, 0..n] es una matriz de booleanos
    n ← s.length()
    for i ← 1 to n do
        V[i, i − 1] ← true
    end for
    i ← n
    while i ≥ 1 do
        for j ← i to n do
            V[i, j] ← dict(s, i, j)
            k ← i
```

```
            while ¬V[i,j] ∧ k ≠ j do
                V[i,j] ← V[i,k] ∧ V[k+1,j]
                k ← k + 1
            end while
        end for
        i ← i − 1
    end while
    return V[1,n]
end procedure
```

Para poder reconstruir la solución (si el texto es válido), bastará anotar para cada i y j, qué valor de k es el que divide el string en dos subtextos válidos. Esto se hace en la matriz E. Si $s[i..j]$ es de hecho una palabra entonces no hay que insertar ningún espacio, y lo indicaremos con $E[i,j] = -1$. El coste en espacio y tiempo del algoritmo de programación dinámica no cambian. Por su parte, el coste del algoritmo de reconstrucción es $\Theta(n)$.

```
procedure EsTextoValido(s, E)
▷ Se muestran sólo los cambios necesarios
▷ para guardar la matriz E
...
    V[i,j] ← dict(s,i,j); E[i,j] ← −1
    k ← i
    while ¬V[i,j] ∧ k ≠ j do
        V[i,j] ← V[i,k] ∧ V[k+1,j]
        if V[i,j] then E[i,j] ← k
        end if
        k ← k + 1
    end while
...
end procedure
procedure ReconstruirTexto(s, E, i, j)
▷ Se asume que s es un texto válido.
▷ Llamada inicial: ReconstruirTexto(s, E, 1, n)
    if i > j ∨ E[i,j] = −1 then
        return s[i..j]
    else
        t1 ← ReconstruirTexto(s, E, i, E[i,j])
        t2 ← ReconstruirTexto(s, E, E[i,j] + 1, j)
        return t1 + " " + t2
    end if
end procedure
```

Una solución más eficiente en tiempo y espacio se basa en definir W_j como la posición de inicio de la última palabra en una descomposición en palabras válida si $s[1..j]$ es un

texto válido. Tomaremos la definición por convenio $W_j = -1$ para indicar que $s[1..j]$ no es un texto válido y $W_0 = 0$, ya que el prefijo vacío es válido, pero no se descompone en palabras.

Entonces W_j satisface la siguiente recurrencia

$$W_0 = 0$$

$$W_j = \begin{cases} -1 & \text{si } s[1..j] \text{ no es válido,} \\ k+1 & \text{la menor } k,\ 0 \le k < j, \text{ tal que } W_k \neq -1 \text{ y } \texttt{dict}(s, k+1, j). \end{cases}$$

Es decir, para $j > 0$, el prefijo $s[1..j]$ es válido si y sólo si $s[1..j]$ es una palabra del diccionario, o existe una k, $0 \le k < j$, tal que $s[1..k]$ es un texto válido y $s[k+1..j]$ es una palabra del diccionario. Notad que $W_j = 1$ si $\texttt{dict}(s, 1, j)$.

A partir de esta recurrencia, obtener el algoritmo es casi automático; el coste en espacio será $\Theta(n)$ y el coste en tiempo $\Theta(n^2)$. La reconstrucción del texto es también muy simple, utilizando la tabla W directamente, y tiene coste lineal.

```
procedure EsTextoValido(s)
▷ W[0..n] es un vector de booleanos
    n ← s.length()
    W[0] ← 0
    for j ← 1 to n do
        W[j] ← −1
        k ← 0
        while W[j] = −1 ∧ k ≠ j do
            if W[k] ≠ −1 ∧ dict(s, k + 1, j) then
                W[j] ← k + 1
            end if
            k ← k + 1
        end while
    end for
    return W[n] ≠ −1
end procedure
procedure ReconstruirTexto(s, W, j)
▷ Se asume que s es un texto válido.
▷ Llamada inicial: ReconstruirTexto(s, W, n)
    if W[j] = 1 then
        return s[1..j]
    else
        t ← ReconstruirTexto(s, W, W[j] − 1)
        return t + " " + s[W[j]..j]
    end if
end procedure
```

4. **(5 puntos)** Se nos da una secuencia de elementos muy larga (de longitud N) en la que hay muchas repeticiones; sólo hay $n \ll N$ elementos distintos en la secuencia, pero el valor de n es desconocido, aunque sabemos que no excederá cierto valor. Para estimar el número n de elementos distintos, sin usar excesiva memoria auxiliar, se utiliza un vector de M bits y una función de hash h como sigue: para cada elemento de la secuencia, aplicamos la función h y el bit de la posición correspondiente se pone a 1. Si $n = \alpha M$ para algún $\alpha < 1$ entonces después de procesar la secuencia entera quedarán (con alta probabilidad) algunas posiciones del vector de bits que están a 0. Sea Z el número de bits a 0. Justifica porqué, si $Z > 0$, la fórmula

$$M \ln \frac{M}{Z}$$

da una aproximación del valor n.

Pista: Calcula el valor esperado de Z después de "insertar" n elementos en el vector de bits. Las repeticiones de un cierto elemento son irrelevantes: con cada una de las repeticiones, la función de hash nos llevará a la misma posición del vector de bits y se pondrá el bit 1.

SOLUCIÓN:
La probabilidad de que la posición i, $0 \leq i < M$, esté a 0 después de una inserción es

$$1 - \frac{1}{M},$$

y si hacemos n inserciones de elementos distintos, la probabilidad de que la posición i permanezca a 0 es

$$\left(1 - \frac{1}{M}\right)^n,$$

pues podemos asumir que son independientes. Por tanto el número esperado de posiciones que estarán a 0 es

$$\mathbb{E}\{Z\} = M \cdot \left(1 - \frac{1}{M}\right)^n.$$

Si $n = \alpha M$ es suficientemente grande

$$\mathbb{E}\{Z\} = M \cdot \left(1 - \frac{1}{M}\right)^{\alpha M} \sim M \cdot e^{-\alpha} = M \cdot e^{-n/M}.$$

Despejando n,

$$n \sim M \cdot \ln\left(\frac{M}{\mathbb{E}\{Z\}}\right).$$

El estimador del valor n simplemente se obtiene reemplazando en la fórmula el valor esperado $\mathbb{E}\{Z\}$ por el valor "observado" Z.

5. **(5 puntos)** Dado un árbol binario de búsqueda aleatorizado (RBST), escribe un método en C++ que elimina el elemento con menor clave del árbol. Indica cuál es el coste promedio

de la operación en función del tamaño n del árbol. Justifica que el árbol resultante de la eliminación es un árbol aleatorizado.

Utiliza las siguientes definiciones en C++:

```
class RBST {
private:
   struct node_rbst {
      Clave k;
      Valor v;
      node_rbst* left;
      node_rbst* right;
      int size; // tamaño del subárbol del cual
                // es raíz el nodo
   };
   node_rbst* root;
   ...
public:
   ...
   void elim_min();
   ...
}
```

SOLUCIÓN:

La eliminación del mínimo en un RBST es prácticamente igual que en un BST ordinario, la única diferencia es que debemos mantener actualizada la información sobre la talla de cada subárbol. Usaremos un método de clase privado `elim_min(node_rbst* p)` que dado un puntero p a la raíz de un árbol no vacío, elimina el mínimo del árbol y nos devuelve un puntero a la raíz del árbol resultante.

```
class RBST {
private:
   ...
// método privado
static node_rbst* elim_min(node_rbst* p);
   ...
public:
   ...
}

void RBST::elim_min() {
     if (root == NULL) return; // el árbol es vacío, no hacemos nada
     // en otro caso, invocamos al método privado
     root = elim_min(root);
}
```

```
// p != NULL
static RBST::node_rbst* RBST::elim_min(node_rbst* p) {
       if (p -> left == NULL) {
          // p apunta al mínimo del árbol
          node_rbst* result = p -> right;
          delete p; // devolvemos la memoria dinámica de p
          return result;
       } else {
          --(p -> size);
          p -> left = elim_min(p -> left);
          return p;
       }
}
```

El procedimiento `elim_min` baja por la rama izquierda del árbol (recursivamente) hasta hallar el primer nodo que no tiene subárbol izquierdo: dicho nodo contiene la menor clave del árbol, y es reemplazado por su subárbol derecho. Según descendemos por la rama izquierda, sabemos los sucesivos antecesores del nodo con clave mínima van a tener un descendiente menos y por ello vamos actualizando `p->size` en cada llamada recursiva.

El coste de la operación es, por lo tanto, proporcional a la longitud del camino entre la raíz del árbol y el nodo que contiene la mínima clave; para un RBST de tamaño n, la longitud promedio de dicho camino es $\Theta(\log n)$ y por consiguiente ese es también el coste esperado de la operación.

Para demostrar que el árbol resultante de eliminar el mínimo en un RBST es también un RBST, basta recordar que el subárbol derecho del nodo eliminado es un RBST (todo subárbol en un RBST es un RBST). El RBST cuya raíz es el nodo a eliminar, su subárbol izquierdo vacío y su subárbol derecho R son reemplazdos por el RBST R y por lo tanto el árbol resultante de la eliminación es un RBST.