Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Apellidos, Nombre D.N.I.

Titulación: Grau d'Enginyeria Informàtica **Curso:** Q2 2011–2012 (Final)
Asignatura: Algorísmia **Fecha:** 6 de Junio de 2012
Duración: 2h 30m

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 punto $p' = (x', y')$ **domina** a otro punto $p = (x, y)$ si $x \leq x'$ e $y \leq y'$. Dado un conjunto de n puntos $F = \{(x, y) \mid x, y \in \mathbb{N}\}$ con coordenadas enteras positivas, diseñad un algoritmo eficiente para encontrar los *óptimos de Pareto* del conjunto F. Los óptimos de Pareto en F son los puntos del conjunto que **no** están dominados por ningún otro punto del conjunto. Vuestro algoritmo debe tener coste necesariamente inferior a cuadrático (es trivial resolver el problema con coste $\Theta(n^2)$).

SOLUCIÓN:
Se ordenan los puntos por x creciente, con un algoritmo eficiente de ordenación, p.e., quicksort, heapsort o mergesort en tiempo $\Theta(n \log n)$. En caso de empate, se ordenan por y creciente. Una vez ordenados, digamos que en un vector $F[1..n]$, se recorre el vector de derecha a izquierda, cada máximo local en y que encontramos es un óptimo de Pareto. Por ejemplo, $F[n]$ no está dominado por ningún otro punto puesto que su x es máxima, y si hay algún otro punto con la misma x, entonces su y es máxima. En general, supongamos que hemos recorrido el subvector $F[i+1..n]$ y hallado todos los óptimos de Pareto hasta ese momento. Sea $y_{\max}$ el valor más alto de y visto en dicho subvector, digamos para el punto j, $i < j \leq n$. Si $F[i]$ tiene una $y \leq y_{\max}$ entonces está dominado por el punto $F[j]$ ya que $F[i].x \leq F[j].x$ y $F[i].y \leq F[j].y = y_{\max}$. Por el contrario, si $F[i].y > y_{\max}$ entonces $F[i]$ es un óptimo de Pareto ya que ningún otro punto le domina, y $y_{\max} := F[i].y$.

```
void OptimosPareto(vector<punto>& F,
                   list<punto>& opt_Pareto) {
     if (F.size() == 0) return;
     sort(F.begin(), F.end());
     // la comparacion entre puntos
     // es (x,y) < (x',y') ssi x < x' o (x=x' e y<y')
     opt_Pareto.push_back(F[F.size()-1]);
     int ymax = F[F.size()-1].y;
     for (int i = F.size() - 2; i >= 0; --i)
         if (F[i].y > ymax) {
            opt_Pareto.push_back(F[i]);
            ymax = F[i].y;
         }
}
```

El coste total del algoritmo es obviamente $\Theta(n \log n)$, pues la última parte tiene coste $\Theta(n)$.

2. **(5 puntos)** Tenemos una *skip list* S de tamaño n donde cada uno de los elementos incluye su *rango* o *número de orden*. Así, por ejemplo, si `p` apunta al primer nodo de la skip list entonces `p ->rank` vale 1.

   Escribid un algoritmo (preferentemente en C++) eficiente que dada la skip list de tamaño n y un rango i, $1 \leq i \leq n$, nos devuelve el i-ésimo elemento de la skip list. Naturalmente lo que no tiene ningún mérito es recorrer la lista de nivel 1 secuencialmente hasta llegar al i-ésimo, debe sacarse partido de la estructura de datos.

   Utilizad las siguientes declaraciones:

```
struct node {
  Elem info;
  int rank;
  vector<node*> next; // next[i] = siguiente del nodo en
                      // la lista de nivel i
};

struct skiplist {
  int size;           // = num. de elementos
  int altura;         // = num. de niveles
  vector<node*> ini; // ini[i] = apuntador al primero de
                     //          la lista de nivel i
}

// 1 <= i <= S.size
Elem i_esimo(const skiplist& S, int i)
```

   Describe brevemente (no hace falta escribir el código C++) cómo debe modificarse el algoritmo de inserción para mantener actualizada la información el rango. ¿Puedes

señalar un defecto grave de esta solución? ¿Porqué guardar el rango de cada elemento no es una buena idea? El espacio de memoria extra **no** es un problema.

SOLUCIÓN:
El algoritmo de búsqueda del i-ésimo es extraordinariamente simple:

```
Elem i_esimo(const skiplist& S, int i) {
  if (i < 1 or i > S.size) return ... // un Elem ficticio
  int l = S.altura;
  node* p = S.ini[l];
  while (p -> rank != i)
    if (p -> rank < i)
        p = p -> next[l];
    else
        --l;
  return p -> info;
}
```

Su coste es igual al de una búsqueda de un elemento x entre el $(i-1)$-ésimo y el i-ésimo. En promedio, será $\Theta(logn)$.

En el algoritmo de inserción se necesita determinar cuáles son los predecesores, en cada uno de los niveles, del nuevo elemento x a insertar. Si el predecesor de x en el nivel 1 es el elemento de rango i entonces x será el elemento de rango $i+1$; ¡y aquí reside el problema! los elementos mayores que x tienen que ser actualizados, su rango se incrementa en 1 para todos ellos. Por lo tanto el coste de una inserción pasa a ser $O(n)$. Otro tanto ocurre con los borrados. La solución a este problema consiste en almacenar otro tipo de información.

Consideremos un nodo p de la skip list y un cierto nivel $i \leq \text{nivel}(p)$. El nivel de un nodo es el número de listas enlazadas en las que está incluido. Dado un par (p,i), $\text{sz}(p,i)$ denotará el número de nodos accesibles desde (p,i), esto es, todos los nodos alcanzables en la skip list desde (p,i): esto incluye al propio nodo p y todos los nodos a su "derecha" hasta llegar al primer nodo cuyo nivel sea mayor que i. Para la skip list completa, $\text{sz}(S, S.\texttt{altura}) = S.\texttt{size} = n$. Supongamos que almacenamos en cada nodo p el valor $\texttt{sz} = \text{sz}(p, \text{nivel}(p))$. Al iniciar el algoritmo examinamos el primer nodo accesible desde el inicio de la skip list a la máxima altura, y supongamos que $\texttt{sz} = k$ para dicho nodo. Entonces eso significa que hay $n-k$ nodos que le preceden y que a continuación están los k restantes (incluído el nodo en cuestión). Si $i \leq n-k$ entonces debemos buscar el i-ésimo descendiendo al nivel inferior y repetir el procedimiento. Si $i = n-k+1$ el nodo buscado es el que estamos viendo. Si $i > n-k+1$ entonces el nodo que buscamos se encuentra a la derecha, y habrá que repetir el procedimiento, arrancando desde dicho nodo y siguiendo a la máxima altura, pero ahora buscaremos el elemento cuyo rango (relativo) es $i' = i-(n-k+1)$.

Este nuevo algoritmo es un poquitín más complejo que la solución dada más arriba,

pero elimina el problema de la ineficiencia en inserciones y borrados. Efectivamente, cuando buscamos el punto de inserción de un nuevo elemento (o el elemento a borrar) pasamos por una serie de nodos en el camino de búsqueda y esos y sólamente esos son los únicos cuyo valor `sz` debe incrementarse (o decrementarse) pues un nuevo elemento será accesible (o dejará de serlo) desde ellos. Con este nuevo esquema todas las operaciones (inserciones, borrados y búsqueda del i-ésimo) tienen coste promedio $O(\log n)$.

3. **(5 puntos)** En el problema de la *circulación con demandas* tenemos un grafo dirigido G con capacidades c_e en los arcos y *demandas* en los nodos d_v. Para simplificar, las capacidades y las demandas son números enteros. Un nodo v con demanda $d_v > 0$ es un "sumidero" en el que deseamos recibir d_v unidades de flujo. Un nodo v con demanda $d_v < 0$ es una "fuente" que puede generar $-d_v$ unidades de flujo. Los restantes nodos tendrán $d_v = 0$. Cuidado: puede haber arcos entrantes en las "fuentes" y arcos salientes de los "sumideros".

   Una solución factible al problema es un flujo f tal que, para todo arco e, $0 \leq f(e) \leq c_e$ y para todo vértice v se cumple $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$. Es fácil demostrar que sólo puede existir solución factible al problema si la suma D de las demandas positivas iguala a la suma de las demandas negativas (cambiadas de signo). Si no fuera así, o bien no habría suficiente oferta (demanda negativa) para satisfacer la demanada (positiva), o bien habría un exceso de oferta.

   Diseñad un algoritmo, basado en la maximización de flujos, que determine el flujo que debe circular en cada arco en una solución factible del problema de la circulación con demandas, si tal solución existe.

SOLUCIÓN:
Denotemos S el subconjunto de los vértices de G con demanda negativa (= oferta), y T el de los vértices con demanda positiva. Añadimos un vértice s^* y arcos que unen a s con cada uno de los vértices de S. De manera similar, se añade un vértice t^* y arcos que van de cada uno de los vértices de T a t^*. Los arcos de la forma (s^*, v) tienen capacidad $-d_v$ y los arcos de la forma (v, t^*) tienen capacidad d_v. La nueva red G' así formada es una red s-t. Sea

$$D = \sum_{v \in T} d_v = -\sum_{v \in S} d_v.$$

Si consideramos el corte $\langle \{s^*\}, V' - \{s^*\} \rangle$ su capacidad es D y por lo tanto todo flujo en G' necesariamente ha de tener valor menor o igual a D. Ahora bien, si la red G' admite un flujo máximo de valor D entonces todos los arcos que salen de s^* y todos los que llegan a t^* han de estar saturados. Como el arco (s^*, v) no existe en la red original, tiene que cumplirse que la diferencia entre el flujo entrante y el saliente, descontado el que llega a través de (s^*, v) es $-d_v$. De igual manera, para los vértices de T, la diferencia entre el flujo entrante y el saliente, descontado el que sale por el

arco (v, t^*) es d_v. Si el flujo máximo sobre G' tiene valor inferior a D entonces algunas demandas no podrán ser satisfechas. Así pues, la red original admite una solución factible si y sólo si la red G' tiene un flujo máximo de valor D, para

$$D = \sum_{v \in T} d_v = -\sum_{v \in S} d_v.$$

Podemos usar cualquier algoritmo de maximización de flujo, como por ejemplo el de Ford y Fulkerson. El coste adicional de incluir los vértices s^* y t^* y los arcos entre éstos y los vértices de S y de T, respectivamente, es $O(n)$.

4. **(5 puntos)** Una empresa tiene que acometer proyectos durante n semanas, cada proyecto necesita una semana para su ejecución. Hay dos perfiles de proyectos: proyectos de perfil bajo y proyectos de alto stress. Para cada una de las semanas i, $1 \leq i \leq n$, podemos decidir realizar un proyecto de perfil bajo con beneficio ℓ_i, un proyecto de alto stress con beneficio h_i (típicamente $h_i > \ell_i$) o no hacer nada y nuestro beneficio será 0. Naturalmente queremos maximizar el beneficio total en las n semanas, pero el problema estriba en que si queremos realizar un proyecto de alto stress en la semana $i > 1$ (y obtener gracias a ello un beneficio de h_i euros), la semana anterior $i - 1$ no podremos realizar ningún proyecto, pues será necesario prepararse para la ejecución del proyecto estresante de la semana i ...

   Diseñad un algoritmo de programación dinámica que dados los beneficios de los proyectos $\ell = (\ell_1, \ldots, \ell_n)$ y $h = (h_1, \ldots, h_n)$ nos devuelve una planificación de proyectos $P = (p_1, \ldots, p_n)$ que maximiza el beneficio total. Cada p_i puede ser L, H, N, según que en la semana i se realice un proyecto de perfil bajo (L), uno de alto stress (H) o no se haga nada (N). Calculad su coste en tiempo y espacio en función de n.

SOLUCIÓN:
Sea B_i el beneficio máximo alcanzable en las primeras i semanas, con $B_0 = 0$ y $B_1 = \text{máx}(\ell_1, h_1)$. En la semana $i \geq 2$ tenemos dos opciones, y habremos de quedarnos con la opción que maximice el beneficio:

$$B_i = \text{máx}(B_{i-1} + \ell_i, B_{i-2} + h_i).$$

La opción de no hacer nada la última semana (la i) no tiene sentido, y no tenemos que considerarla. La estructura del plan óptimo también es fácil: si $B_i = B_{i-1} + \ell_i$ entonces $P_i = [P_{i-1}, L]$ y si $B_i = B_{i-2} + h_i$ entonces $P_i = [P_{i-2}, N, H]$. El algoritmo de programación dinámica resulta casi inmediato una vez tenemos las recurrencias anteriores.

```
...
vector<double> B(n+1);
vector<char> plan(n+1);
B[0] = 0.0; B[1] = max(l[1], h[1]);
for (int i = 2; i <= n; ++i)
```

```
  if (B[i-1]+l[i] > B[i-2]+h[i]) {
      B[i] = B[i-1]+l[i];
      P[i] = 'L';
  } else {
      B[i] = B[i-2] + h[i];
      P[i] = 'H'; P[i - 1] = 'N';
  }
...
```

El coste del algoritmo del algoritmo es $\Theta(n)$ tanto en espacio como en tiempo.

5. **(5 puntos)** En un contexto de rápida inflación debemos adquirir n items cuyo precio inicial $C > 0$ es para todos el mismo; sin embargo, sólo podemos comprar un item cada semana y el precio del i-ésimo item se incrementa en progresión geométrica con tasa $\tau_i > 1$. Así, si compramos el item i en la semana t, $0 \leq t \leq n-1$, su precio será $C \cdot \tau_i^t$. Diseñad un algoritmo voraz que, dadas las tasas $\tau_0, \ldots, \tau_{n-1}$ y el valor C, determine en qué orden deben hacerse las compras y qué valor total tendrán las n compras de tal modo que dicho valor sea mínimo; en concreto, para cada semana t, el algoritmo debe devolver un vector *item* tal que $item[t]$ es el índice del item a comprar en la semana t. Calculad el coste del algoritmo en función de n y demostrad su corrección.

   Por ejemplo, si $n = 3$, $C = 100$ y $\tau = [3, 2, 4]$, y compramos los items en el orden 0, 1, 2 entonces el valor total de las compras es

$$100 \cdot 3^0 + 100 \cdot 2^1 + 100 \cdot 4^2 = 100 \cdot (1 + 2 + 16) = 1900 \quad \text{euros}$$

   El orden de las compras es relevante: en efecto, podemos comprar los 3 items del ejemplo con un gasto de 800 euros.

SOLUCIÓN:

Tras una breve experimentación llegamos a la conclusión de que los items deben comprarse por orden decreciente de su tasa. De modo que el algoritmo voraz consiste sencillamente en ordenar los items y obtener mediante un simple cálculo el valor. La ordenación toma tiempo $\Theta(n \log n)$; el cálculo del valor de las compras puede llevarnos tiempo $O(n^2)$ salvo que utilicemos un algoritmo astuto para calcular las potencias de las τ_i's. Podemos conseguir que dicho coste sea también $\Theta(n \log n)$.

```
void orden_compras(vector<double>& tasa, double C,
                   vector<int>& item, double& val) {

    for (int t = 0; t < tasa.size(); ++t)
        item[t] = t;
    ordenar_por_tasa_decr(item,tasa);
    // se ordena el vector item de tal modo
    // tasa[item[t]] >= tasa[item[t+1]] para toda t=0..n-2
```

```
        // Coste: O(n log n)

        // pot(x,k) calcula x^k con coste O(log k)
        val = 0.0;
        for (int t = 0; t < item.size(); ++t)
            val += pot(tasa[item[t]], t);
        return C * val;
}
```

Para argumentar la corrección (optimalidad) del algoritmo voraz supongamos que los items están de hecho ordenados por tasa decreciente: $\tau_{i+1} \geq \tau_i$ para toda i, $0 \leq i < n-1$. En tal caso, el algoritmo voraz devuelve como solución que en la semana t hay que comprar el item t.

Consideremos una solución óptima y supongamos que en dicha solución los items no se compran por orden decreciente de tasa. Denotemos, para simplificar, $\pi(t)$ el índice del item comprado en la semana t en la solución óptima. Por lo tanto, estamos suponiendo que existirá una semana t tal que $\tau_{\pi(t)} < \tau_{\pi(t+1)}$, es decir, la tasa del item comprado en la semana t es menor que la del item comprado en la semana $t+1$. Si construímos un nuevo orden de compras π' intercambiando los items comprados en las semanas t y $t+1$, entonces el valor V' de π' será igual al valor V del orden óptimo π, excepto por el cambio derivado del intercambio del orden de compra de las semanas t y $t+1$:

$$\begin{aligned} V = \text{valor}(\pi) &= C \cdot \left(1 + \tau_{\pi(1)} + \tau^2_{\pi(2)} + \ldots + \tau^{n-1}_{\pi(n-1)}\right) \\ V' = \text{valor}(\pi') &= C \cdot \left(1 + \tau_{\pi'(1)} + \tau^2_{\pi'(2)} + \ldots + \tau^{n-1}_{\pi'(n-1)}\right) \\ &= V - C(\tau^t_{\pi(t)} + \tau^{t+1}_{\pi(t+1)}) + C(\tau^t_{\pi'(t)} + \tau^{t+1}_{\pi'(t+1)}) \end{aligned}$$

Debemos ver qué sucede con la diferencia d entre V y $V' = V + d$:

$$d = -C(\tau^t_{\pi(t)} + \tau^{t+1}_{\pi(t+1)}) + C(\tau^t_{\pi'(t)} + \tau^{t+1}_{\pi'(t+1)})$$

Como $\pi'(t) = \pi(t+1)$ y $\pi'(t+1) = \pi(t)$ tenemos

$$\begin{aligned} d &= -C(\tau^t_{\pi(t)} + \tau^{t+1}_{\pi(t+1)}) + C(\tau^t_{\pi(t+1)} + \tau^{t+1}_{\pi(t)}) \\ &= C \cdot \left(\tau^t_{\pi(t)}(\tau_{\pi(t)} - 1) - \tau^t_{\pi(t+1)}(\tau_{\pi(t+1)} - 1)\right) \end{aligned}$$

Ahora bien, como todas las τ_i's son > 1 y $\tau_{\pi(t)} < \tau_{\pi(t+1)}$ vemos que

$$\tau^t_{\pi(t)}(\tau_{\pi(t)} - 1) < \tau^t_{\pi(t+1)}(\tau_{\pi(t)} - 1) < \tau^t_{\pi(t+1)}(\tau_{\pi(t+1)} - 1),$$

y se deduce que $d < 0$. Hemos llegado a una contradicción, pues $V' = V + d < V$, y V era óptima. Así que nuestro supuesto de que en la solución óptima existe una semana t tal que $\tau_{\pi(t)} < \tau_{\pi(t+1)}$ ha de ser falso y la solución voraz es óptima.