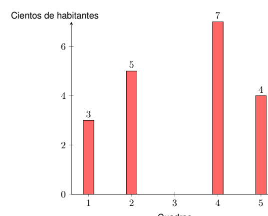
Segundo Parcial de Programación 3

30 de noviembre de 2015

Ejercicio 1 (20 puntos)

En la administración municipal se consideran las solicitudes para construir edificios en una nueva avenida de largo n cuadras, numeradas de 1 a n. Cada edificio estaría en el centro de una cuadra diferente. El edificio de la cuadra i-ésima alojaría h_i habitantes y si no hay una solicitud para esa cuadra se define $h_i = 0$. Tal vez no se puedan aceptar todas las solicitudes porque por razones de infraestructura (saneamiento, transporte, etc.) la distancia entre dos edificios, medida en cuadras, debe ser por lo menos d, con $1 \leq d \leq n$. Se pretende aceptar solicitudes de tal manera que se maximice la cantidad de habitantes en la avenida.

Ejemplo:
Sea n = 5, $h_1 = 300$, $h_2 = 500$, $h_3 = 0$, $h_4 = 700$, $h_5 = 400$.
Si $d = 3$ la solución óptima es $t = \langle 1, 4 \rangle$ que permite un total de $T = 1000$ habitantes.
Si $d = 1$ una solución óptima es $\langle 1, 2, 4, 5 \rangle$.



1. Se propone el siguiente algoritmo, en el que los edificios se consideran en orden decreciente de la cantidad de habitantes: aceptar el primero y a cada uno de los otros aceptarlo si cumple la restricción de la distancia con los edificios ya aceptados. En el ejemplo anterior, el orden en que se considerarían es $4, 2, 5, 1$.

   a) ¿De qué técnica vista en el curso es un ejemplo este algoritmo? Explique brevemente.

   b) Demuestre que siempre se puede obtener una solución óptima o exhiba un contraejemplo, lo más sencillo posible, en el que no ocurre.

2. De manera independiente a la respuesta del punto anterior, se debe resolver el problema mediante Programación Dinámica asumiendo que se cumple el Principio de Optimalidad. Se define $T(i)$ como el total de habitantes de una solución que optimice el subproblema definido para las cuadras $1, \ldots, i$. Se puede calcular $T(i)$ mediante una recurrencia.

   a) Exprese la recurrencia.

   b) Implemente la recurrencia para el problema $T(n)$ corrigiendo el siguiente algoritmo:

```
1  int total (int * h, int n) {
2     int T[n+1]; // El resultado del subproblema T(i) se mantiene en T[i]
3     T[0] = 0;
4     T[1] = h[1];
5     for (int i = 1; i <= n; i++)
6         if (h[i] > T[i-1])
7            T[i] = h[i] + T[i-d];
8         else
9            T[i] = T[i-1];
10    return T[n];
11 }
```

   Solamente puede agregar, suprimir o modificar líneas entre las líneas 3 y 9 incluidas. No puede usarse recursión ni estructuras de datos auxiliares.

Solución

1. *a*) Es un ejemplo de técnica Greedy. En cada paso se elige el edificio con más habitantes que sea compatible con el estado del problema.

   b) En general este algoritmo no encuentra una solución óptima. Contraejemplo: sea $d = 2$, $n = 3$, $h_1 = 2, h_2 = 3, h_3 = 2$. La solución del algoritmo es $\langle 2 \rangle$, que genera un total 3. Pero con $\langle 1, 3 \rangle$ se puede llegar a 4.

2. *a*) La recurrencia que se pide se puede expresar con

$$T(i) = \begin{cases} 0 & \text{si } i = 0 \\ \text{máx}\{h_i + T(\text{máx}\{0, i-d\}), T(i-1)\} & \text{si } i \geq 1. \end{cases} \quad (1)$$

b) La implementación iterativa de esta recurrencia, asumiendo que está definida la función máx que devuelve el máximo de sus parámetros, es:

```
int Total (int * h, int n)  {
  int T[n+1];
  T[0] = 0;
  for (int i = 1; i <= n; i++)
    T[i] = max (h[i] + T[max (0, i - d)], T[i-1]);
  return T[n];
}
```

De manera alternativa en la recurrencia se pueden separar los casos en los que $i \leq d$:

$$T(i) = \begin{cases} 0 & \text{si } i = 0 \\ \text{máx}\{h_i, T(i-1)\} & \text{si } 1 \leq i \leq d \\ \text{máx}\{h_i + T(i-d), T(i-1)\} & \text{si } i > d. \end{cases}$$

Y otra variante puede ser tener como caso base $T(1) = h_1$ en lugar de $T(0) = 0$.

La recurrencia (1) y sus variantes se obtienen tras la aplicación del Principio de Optimalidad.

Sea P_i el subproblema correspondiente a hallar la solución óptima para los edificios en las cuadras $\{1, 2, \ldots, i\}$. La solución, que suponemos óptima, tiene la forma $s = (s_1, s_2, \ldots, s_f)$, con $1 \leq s_1 < s_2 < \ldots < s_f \leq i$, en donde cada s_i representa la cuadra en que está un edificio incluido en la solución. De manera trivial, la solución de P_1 es (1). Para $i > 1$ la última decisión tomada al resolver P_i es si se inclye o no el edificio de la cuadra i-ésima, o sea si $s_f = i$ o $s_f < i$. Llamemos $H_i(t)$ al total de habitantes de una solución t para el problema P_i, esto es $H_i(t) = \sum_{j=1}^{i} h_j$.

- Si se decide no incluir i, esto es si $s_f < i$, la solución de P_i sólo contiene edificios en el conjunto $\{1, \ldots, i-1\}$ por lo que es también solución de P_{i-1} y se cumple $H_{i-1}(s) = H_i(s)$. Y esta solución tiene que ser óptima para P_{i-1} porque cualquier solución r para P_{i-1} no puede superar la cantidad de habitantes de s. Si no fuera así, si $H_{i-1}(r) > H_{i-1}(s)$, se tendría $H_i(r) = H_{i-1}(r) > H_{i-1}(s) = H_i(s) = T(i)$, donde la última igualdad se deriva de la hipótesis de optimalidad de s para el problema P_i. Y esto es una contradicción porque r es también solución para P_i y permitiría alojar una cantidad de habitantes mayor que el óptimo.
- Si la decisión es incluir i entonces en la solución no puede estar ninguno de los edificios anteriores cuya distancia al edificio i-ésimo sea menor que d, lo cual implica que o bien $i \leq d$, lo que implica que en s sólo está i ($f = 1$), o bien $s_{f-1} \leq i - d$.
  - En este último caso la subsolución $(s_1, \ldots, s_{f-1})$ es solución para el problema P_{i-d} y se cumple $H_{i-d}(s) + h_i = H_i(s)$. Y no puede haber una solución para P_{i-d}, $r = (r_1, \ldots r_f)$, mejor que $(s_1, \ldots, s_{f-1})$. Supongamos que r fuera mejor, o sea que $H_{i-d}(r) > H_{i-d}(s)$. Al inluir i en la solución r se obtiene $r' = (r_1, \ldots, r_f, i)$, cuya cantidad de habitantes sería $H_{i-d}(r) + h_i > H_{i-d}(s) + h_i = H_i(s) = T(i)$. De esta forma se llega a la contradicción de que r', que es solución para P_i, permite alojar más habitantes que la solución óptima.
  - En el primer caso el subproblema P_{i-d} no está definido, por lo que de manera trivial no puede haber una mejor solución para P_{i-d} que la subsolución obtenida de excluir i de s.

Por lo tanto se cumple el Principio de Optimalidad.

Como caso base, de manera alternativa a P_1, con $T(1) = h_1$, se puede definir $T(0) = 0$.

Además, para cada cuadra i se define u_i, la última cuadra que cumple la restricción de distancia:

$$u_i = \begin{cases} 0 & \text{si } i \leq d \\ i - d & \text{si } i > d \end{cases}$$

lo que es lo mismo que $u_i = \text{máx}\{0, i-d\}$.

Con esto se justifica la recurrencia (1).

Ejercicio 2 (20 puntos)

Un canal de televisión desea planificar el orden y contenido de las tandas publicitarias para el programa con más rating del canal. Para esto cuenta con un conjunto de reclames a trasmitir. La información se mantiene en el arreglo *reclames* entre los índices entre 1 y n. Los atributos de cada reclame son:

- *identificador*, que es la posición en el índice, está entre 1 y n;
- *duracion*, en cantidad de segundos, no mayor a 1 minuto;
- *minimo*, cantidad mínima de apariciones en todo el programa;
- *maximo*, cantidad máxima de apariciones en todo el programa, menor que 10.

La planificación debe cumplir lo siguiente:

A) Debe haber 5 tandas en el trancurso de todo el programa.

B) Cada tanda tiene que durar entre 4 y 5 minutos.

C) La cantidad total de minutos destinados a las tandas durante el programa no pueden superar los 23 minutos.

D) No puede emitirse el mismo reclame dos veces consecutivas en la misma tanda.

E) En cada tanda se deben emitir los reclames en orden decreciente (no estricto) de duración.

1. Defina una forma de tupla para el problema, considerando que cada componente represente una tanda.

2. Defina por extensión el dominio de las componentes de la tupla (liste los elementos que pertenecen al dominio), dado que los reclames son los siguientes:

		Apariciones	
Identificador	Duración (seg)	Minimo	Máximo
1	50	2	5
2	58	1	5
3	40	2	6
4	45	2	5
5	60	1	5
6	55	2	5

   Por ejemplo, ¿puede el reclame 2, seguido del 5, seguido del 4 ser una tanda válida?

3. Indique para cada una de las expresiones *A-E* si se trata de una Restricción Explícita, una Restricción Implícita, un Predicado de Poda o si queda considerada dentro de la forma de la solución. Exprese cada una en lenguaje formal.

4. Enuncie otras restricciones implícitas o explícitas y expréselas en lenguaje formal. Indique qué tipo de restricción es.

5. Proponga una función objetivo que maximice la facturación generada por el canal, considerando que el mismo cobra una cantidad F por segundo y que realiza descuentos según la cantidad de apariciones del reclame. El descuento para la aparición i-ésima es $(i-1) \times 10\,\%$ (por ejemplo, si un reclame aparece tres veces por la primera vez paga el 100 %, por la segunda 90 % y por la tercera 80 %).

Solución

1. La forma de la solución es una tupla T de largo fijo 5, en la cual cada componente representa cada una de las 5 tandas publicitarias del programa. Las tandas se representan con vectores de largo variable cuyos componentes representan los identificadores de los reclames en el orden en que se van a emitir. El largo del vector i-ésimo se representa con k_i.

$$T = \{t_1, t_2, t_3, t_4, t_5\} \text{ tal que } t_i = (r_{i1}, \dots, r_{ik_i}) / r_{ij} \in \{1 \dots n\}$$

2. Antes de definir el dominio veremos las restricciones que nos ayudan a acortar las tuplas posibles:

   - Dado que la suma de las duraciones de los 4 reclames más largos no alcanza a 4 minutos se concluye que cada tanda debe estar compuesta por más de 4 reclames. Y dado que la suma de las duraciones de 6 reclames supera los 5 minutos la tanda debe tener menos de 6 reclames. Por lo tanto todas las tandas deben estar compuestas por 5 reclames.
   - Al no haber reclames con igual duración y como deben estar en orden decreciente de duración y no se puede emitir de manera consecutiva el mismo reclame, en una misma tanda no se pueden repetir reclames.

   Dicho lo anterior, los vectores posibles son los siguientes:

$$D = \{\langle 5,2,6,1,4\rangle, \langle 5,2,6,1,3\rangle, \langle 5,2,6,4,3\rangle, \langle 5,2,1,4,3\rangle, \langle 5,6,1,4,3\rangle, \langle 2,6,1,4,3\rangle\}$$

   y no hay otros vectores posibles.

3. Para las siguientes formalizaciones asumiremos que S es el espacio de soluciones.

   Expresión A: Forma de la solución

$$\#T = 5, \forall T \in S$$

   Expresión B: Restricción explícita

$$\forall i \in \{1, \dots, 5\}, \qquad 240 \le \sum_{j=1}^{k_i} reclames[r_{ij}].duracion \le 300.$$

   Expresión C: Restricción implícita

$$\sum_{i=1}^{5} \sum_{j=1}^{k_i} reclames[r_{ij}].duracion \le 1380.$$

   Expresión D: Restricción explícita

$$\forall i \in \{1, \dots, 5\}, \forall j \in \{1, \dots, k_i - 1\}, \quad r_{ij} \ne r_{ij+1}.$$

   Expresión E: Restricción explícita

$$\forall i \in \{1, \dots, 5\}, \forall j \in \{1, \dots, k_i - 1\}, \quad reclames[r_{ij}].duracion \ge reclames[r_{ij+1}].duracion.$$

4. Tenemos una restricción implícita sobre los reclames que indica que no se pueden repetir menos de *mínimo* ni más de *máximo*.

   Definimos las funciones $son_iguales$ que devuelve 1 si sus dos argumentos son iguales, $apariciones$ que cuenta la cantidad de veces que un reclame aparece en una tanda y $conteo$ que devuelve la cantidad de apariciones de un reclame en todo el programa:

$$son_iguales(i, j) = \begin{cases} 1 & \text{si } i = j, \\ 0 & \text{en otro caso.} \end{cases}$$

$$apariciones(t_i, j) = \sum_{h=1}^{k_i} son_iguales(r_{ih}, j).$$

$$conteo(j) = \sum_{i=1}^{5} apariciones(t_i, j).$$

Entonces, se debe cumplir que:

$$\forall j \in \{1, \ldots, n\}, \quad reclames[j].minimo \leq conteo(j) \leq reclames[j].maximo.$$

5. Definiremos la función objetivo f, definida sobre todas las tuplas solución al problema, de la siguiente manera:

$$f(S) = \{T | ganancia(T) = max(ganancia(U)/U \in S)\}$$

La función $ganancia(T)$ se define así:

$$ganancia(T) = F \times \sum_{j=1}^{n} reclames[j].duracion \times \sum_{h=1}^{conteo(j)} (1 - 0,1 \cdot (h-1)).$$

Ejercicio 3 (20 puntos)

1. Explique cómo se construye el árbol de decisión de un algoritmo.
2. ¿Qué representan las hojas de un árbol de decisión de un algoritmo dado?
3. ¿Cuál es la altura mínima que puede tener un árbol de decisión que tiene m hojas?
4. ¿Qué representa un camino hasta una hoja en un árbol de decisión de un algoritmo dado?
5. ¿Qué representa la altura de un árbol de decisión de un algoritmo dado?
6. Dado un problema y todos los árboles de decisión correspondientes a los algoritmos que lo resuelven, ¿qué representa la mínima de las alturas de los árboles?

A partir de ahora se consideran solamente algoritmos de ordenación de n elementos no repetidos que comparan elementos de a pares.

7. Probar que para cualquier árbol de decisión de m hojas asociado a un algoritmo de ordenación se cumple: $m \geq n!$
8. Probar que cualquier algoritmo de ordenación debe en el peor caso hacer al menos $\lceil \log n! \rceil$ comparaciones.
9. ¿Cuál es la altura mínima de un árbol de decisión de un algoritmo que ordene 5 elementos?

Solución

1. Un árbol de decisión es una forma de representar el funcionamiento de un algoritmo para todos los datos posibles de un tamaño n dado. Se trata de un árbol binario con raíz. Cada nodo interno contiene una pregunta que se aplica a los datos. Los hijos representan las diferentes posibilidades de respuesta, según los datos que se tengan en la entrada.
2. Cada hoja representa una salida posible del algoritmo para el dato de entrada.
3. La forma más compacta de generar un árbol binario que tenga m hojas, es teniendo todas las hojas al mismo nivel (o en dos niveles consecutivos).

   Si el árbol es completo hasta la profundidad k, entonces en el nivel j (con $1 \leq j \leq k$) hay 2^j nodos (es sencillo probarlo por inducción completa). Por lo tanto, para tener m hojas, éstas deben estar a una profundidad r tal que $2^r \geq m$.

   Por lo tanto, la altura mínima se obtiene cuando $r = \lceil \log m \rceil$.

4. Un camino de la raíz a una hoja representa una ejecución del algoritmo. Empezando por la pregunta realizada en la raíz, el camino contiene la información del resultado de todas las preguntas realizadas, hasta llegar a una hoja en donde se tiene el resultado del algorimo para la entrada dada.

5. Como consecuencia de la pregunta anterior, la altura de un árbol de decisión representa la ejecución de un peor caso del algoritmo (una ejecución que hace la máxima cantidad de preguntas antes de dar el resultado).

6. Un árbol de altura mínima representa la ejecución de un algoritmo que minimiza el peor caso para resolver un problema dado. En este sentido, esta altura es una cota inferior para el peor caso de ejecución de cualquier algoritmo que resuelva el problema dado.

7. Dados n elementos diferentes, hay $n!$ maneras distintas de ordenarlos. Por tal motivo, cualquier algoritmo que ordene n elementos diferentes usando comparaciones entre ellos, debe tener la posibilidad de devolver cada una de las $n!$ permutaciones de salida. Como consecuencia, la cantidad de hojas m de cualquier árbol de decisión para ordenar n elementos debe cumplir con $m \geq n!$.

8. Por la parte 7 sabemos que cualquier algoritmo que ordene n elementos, debe generar un árbol de decisión con al menos $n!$ hojas. Por la parte 3, sabemos entonces que cualquier árbol de decisión debe tener al menos altura $\lceil \log n! \rceil$, y por la parte 6 sabemos que minimizar la altura de un árbol de decisión es equivalente a minimizar el costo del peor caso. Por tal motivo, cualquier algoritmo de ordenación bajo las condiciones dadas en la letra debe realizar al menos $\lceil \log n! \rceil$ comparaciones en el peor caso.

9. Tenemos $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. Por otro lado, sabemos que $2^6 = 64 < 120 < 2^7 = 128$, y que por lo tanto $\lceil \log 5! \rceil = 7$. Entonces, la altura mínima de cualquier árbol de decisión de un algoritmo que orden 5 elementos debe ser 7. En este sentido, cualquier algoritmo que ordene 5 elementos, debe realizar al menos 7 preguntas en el peor caso. Notar que un árbol óptimo con exactamente 120 hojas, tiene 8 hojas en el nivel 6 y 112 en el nivel 7.

   Es interesante notar que en las páginas 183 y 184 del volumen 3 de la segunda edición de la colección ”The Art of Computer Programming”de Donald E. Knuth, hay un algoritmo que permite ordenar 5 elementos usando 7 comparaciones en el peor caso. Este método fue presentado por primera vez en la tesis de doctorado de H. B. Demuth (páginas 41 a 43) de la Universidad de Stanford en 1956.