Estructuras de Datos y Algoritmos – IIC2133
Examen

27 de junio, 2018

1. Algoritmos codiciosos

Tú debes conducir un auto de Santiago a Puerto Montt, suponemos que en línea recta por la autopista. Cuando el estanque de bencina del auto está lleno, te permite viajar k kilómetros sin detenerte. Tú tienes un mapa con las distancias entre bombas de bencina a lo largo de la autopista; las distancias entre bombas consecutivas son todas $\leq k$ kilómetros. Tú inicias el viaje con el estanque lleno, y quieres hacer el menor número posible de detenciones para cargar bencina durante el viaje. Una solución factible es una lista de bombas de bencina en las que tienes que detenerte y que te permiten llegar a Puerto Montt; y una solución óptima es una solución factible con el menor número de bombas.

a) Propón una elección codiciosa —cómo eliges la próxima bomba de bencina en la que tienes que detenerte— que te permita minimizar localmente el número de detenciones [**1.5 pts.**].

Respuesta. Las bombas posibles son aquellas que están a una distancia $\leq k$ km desde mi posición actual, ya que tengo el estanque lleno; elijo codiciosamente la bomba que está más lejos entre éstas (y allí lleno nuevamente el estanque).

b) Da un contraejemplo que muestre que tu elección codiciosa no minimiza el número total de detenciones [**1 pt.**]; o, por el contrario, demuestra que sí lo hace, respondiendo **c**) y **d**).

Me salto **b**): la elección codiciosa de **a**) minimiza el número de detenciones, como se demuestra en **c**) y **d**).

c) Demuestra que hay una solución óptima que se obtiene haciendo la elección codiciosa de **a**) [**2 pts.**].

Respuesta. Supongamos que las próximas bombas son S, …, T, …, U, en orden de cercanía a nuestra posición actual, y que T es la más lejana que no está a más de k km de distancia (es decir, T es la próxima bomba en que deberíamos detenernos según nuestra elección codiciosa).

Y supongamos que hay una solución óptima en que la próxima bomba es S (más cercana que T) y la que le sigue es U (más lejana que T); es decir, la solución óptima es $\{S, U, \ldots\}$. Es fácil ver que en esta solución óptima podemos cambiar la elección de detenernos en S por la de detenernos en T, sin aumentar el número total de detenciones: si desde S se puede llegar a U sin detenerse (entre medio), entonces desde T también se puede llegar a U sin detenerse, ya que T está más cerca de U que S; y, por supuesto, podemos llegar desde nuestra posición actual a T sin detenernos, ya que la distancia no es mayor que k km.

d) Finalmente, demuestra que si combinas la elección codiciosa de **a**) con una solución óptima del subproblema que te queda por resolver (una vez hecha la elección codiciosa), obtienes una solución óptima al problema original [**2.5 pts.**].

Respuesta. Este argumento vale para cualquier solución óptima, independientemente de si la primera elección se hizo codiciosamente o no. En una solución óptima, en que la primera detención es en una bomba R a los j km ($j \leq k$), **el resto del recorrido tiene que ser una solución óptima** (es decir, que minimiza el número de detenciones) **al problema de viajar desde *R* hasta Puerto Montt** [esto que está en *negrita* es lo que hay que demostrar, p.ej., por contradicción, como sigue]. La razón es que si el resto del recorrido no fuera óptimo, entonces podríamos encontrar otra secuencia de detenciones para viajar desde R hasta Puerto Montt con menos detenciones, y podríamos usar esta secuencia en el viaje de Santiago a Puerto Montt, después de detenernos en R, reduciendo el número total de detenciones para este viaje, y contradiciendo así la suposición de que la solución que tenemos es óptima.

2. Programación dinámica

Queremos dar vuelto de S pesos usando el menor número posible de monedas. Si los valores de las monedas, en pesos, ordenados de mayor a menor son $\{v_1, v_2, \ldots, v_n\}$ (es decir, $v_1 > v_2 > \ldots > v_n = 1$), y tenemos una cantidad suficientemente grande de monedas de cada valor, entonces:

a) Muestra que la estrategia codiciosa de dar tantas monedas como sea posible de valor v_1, seguido de dar tantas monedas como sea posible de valor v_2, y así sucesivamente con v_3, etc., no siempre usa el menor número posible de monedas para totalizar S pesos. [**1 pto**]

b) Demuestra, en cambio, que el problema siempre puede resolverse usando programación dinámica. En particular, sea $z(S,n)$ el problema de encontrar el menor número de monedas necesarias para totalizar la cantidad S, con monedas de valor $\{v_1, v_2, \ldots, v_n\}$; entonces:

b1) Dada una solución a $z(S,n)$, identifica subpartes de la solución que sean soluciones óptimas para algunos subproblemas (del mismo tipo, pero más pequeños); o bien, identifica subproblemas cuyas soluciones óptimas puedan ser usadas para construir una solución a $z(S,n)$. [**1.5 pts.**]

b2) Escribe la recurrencia que relaciona la solución a $z(S,n)$ con las soluciones óptimas a los subproblemas; luego, generaliza esta recurrencia de modo que sea aplicable para resolver los subproblemas; y, finalmente, escribe los casos iniciales (es decir, los casos cuyos valores se determinan sin aplicar la recurrencia). [**2 pts.**]

b3) Escribe el algoritmo de programación dinámica, típicamente un algoritmo iterativo. (Si, en cambio, escribes un algoritmo recursivo, asegúrate de incluir los tests necesarios para evitar hacer cálculos redundantes). [**1.5 pts.**]

Respuesta:

a) Basta con dar un contraejemplo donde usar la estrategia greedy no llega al óptimo. Esto se puede hacer cuando los valores de las monedas no son divisores de las monedas más grandes **[1pto]**.

b1) Digamos que la moneda más grande usada en nuestra solución a *z(S,n)* es v_k. Entonces el problema *z(S-*v_k*,n)* fue resuelto de manera óptima para que *z(S,n)* sea óptimo **[1.5pts]**.

b2) Recurrencia **[2pts]**:

$$z(S,n) = \begin{cases} \infty \; si \; S < 0 \\ 0 \; si \; S = 0 \\ \min\limits_{i=1..n} (z(S - v_i, n) + 1) \end{cases}$$

OJO: Hay muchas maneras de escribir esta recurrencia. Esta es solo una manera de hacerla.

b3)

Modo recursivo:

```
int z(S, n, T):
        si T[S] está en la tabla:
                return T[S]
        si S < 0:
                return infinity
        si S = 0:
                return 0

        minimo = infinity

        for i = 1…n:
                result = z(S-v_i, n, T) +1
                si result < minimo:
                        minimo = result
        T[S] = minimo
        return minimo
```

Modo iterativo:

```
int z(S, n):
        T = [-1 for i = 0…S]
        T[0] = 0
        for i = 1...S:
                si tengo una moneda de costo S:
                        T[i] = 1
                        Break
                minimo = infinity
                for j = 1…n:
                        si v_j ≤ S:
                                result = T[S-v_j] +1
                                si result < minimo:
                                        minimo = result
                T[S] = minimo
```

OJO: Hay muchas maneras de escribir estos algoritmos.

3. *Treaps*

Considera un árbol binario de búsqueda (no necesariamente balanceado). Considera que los nodos de este árbol, además de tener una clave, tienen una prioridad, de modo que el árbol está ordenado según las claves de los nodos (como todo árbol binario de búsqueda), pero ahora, además, las prioridades de los nodos satisfacen la propiedad de min-heap. Es decir, si un nodo tiene clave k y prioridad q, entonces los nodos que están en su subárbol izquierdo tienen claves menores que k y los nodos que están en el subárbol derecho tienen claves mayores que k; y, además, las prioridades de los hijos de este nodo son mayores que q.

a) Describe un algoritmo eficiente para insertar un nodo con clave k y prioridad q, y analiza su complejidad.

Respuesta: Se inserta el elemento normalmente como en un árbol binario de búsqueda. Luego se hacen rotaciones subiendo el elemento hasta que su prioridad sea mayor a la de su padre **[3ptos]**.

b) Describe un algoritmo eficiente para eliminar un nodo con clave k, y analiza su complejidad.

Respuesta: Si el elemento eliminado es una hoja no se hace nada especial **[0.5pts]**. Si tiene 1 hijo simplemente se remplaza por su hijo **[0.5pts]**. Si tiene 2 hijos se remplaza por el sucesor o el antecesor y luego se hacen rotaciones hasta que cumple la propiedad de heap **[2pts]**.

4. Algoritmo de Bellman-Ford

a) El algoritmo revisa todas las aristas en cada iteración para ver si es necesario actualizar el costo de llegar a un nodo. Sin embargo, es posible saber cuáles son las aristas que realmente vale la pena revisar en cada iteración (en vez de revisarlas siempre todas). Describe una versión del algoritmo que incorpore este cambio y justifica por qué mejoraría la eficiencia del algoritmo.

b) ¿Cómo se puede hacer para detectar si el grafo contiene un ciclo cuyo costo total es negativo? Juatifica.

Respuesta:

a) Al hacer una iteración actualizando los pesos de algunos nodos solo es posible actualizar los pesos de los nodos vecinos a los recién actualizados **[1pto]**, por lo que se pueden guardar los vecinos de los recién actualizados en una cola y solo se actualizan estos en la iteración siguiente **[2ptos]**.

b) Luego de hacer $|V|$ iteraciones ya no debería haber más actualizaciones de los pesos de los nodos si se sigue iterando a no ser que exista un ciclo negativo. Por lo que si se hace una iteración número $|V|+1$ y se actualiza algún peso entonces existe un ciclo de costo negativo **[3ptos]**.

5. Algoritmos de Dijkstra y Prim

El algoritmo de Prim produce como resultado un árbol que conecta todos los nodos y que tiene costo mínimo. También el algoritmo de Dijkstra, ejecutado hasta llegar a todos los nodos del grafo, da como resultado un árbol de rutas mínimas, con las rutas desde el nodo inicial hasta cada uno de los otros nodos del grafo. La duda que surge es si estos árboles tienen algo en común.

a) Demuestra con un contraejemplo que el árbol producido por Dijkstra no es necesariamente mínimo.

b) Demuestra con un contraejemplo que el árbol producido por Prim no necesariamente tiene las rutas más cortas desde el nodo inicial al resto de los nodos.

Respuesta:

Un posible grafo es V = {A, B, C}, E = {{A,B,4}, {A, C, 2}, {B, C, 3}}. Si partimos haciendo Dijkstra desde A el árbol resultantes tiene las aristas {A, B, 4} y {A, C, 2} y el árbol resultante de hacer Prim desde A tiene las aristas {A, C, 2}, {B, C, 3}.
a) El árbol de Dijkstra no es mínimo ya que tiene costo 6 mientras que el de Prim tiene costo 5 **[3pts].**

b) El árbol de Prim no tiene la ruta óptima a C ya que la ruta tiene costo 5 mientras que en la ruta de Dijkstra tiene costo 4 **[3pts].**

6. Ordenación y estadísticas de orden

Se quiere hacer un estudio de salud sobre la población chilena. Para esto, se registraron varias métricas sobre N personas elegidas aleatoriamente; una de estas métricas es la estatura de las personas. Para eliminar valores muy extremos (*outliers*), se decidió considerar solo las estaturas desde la i-ésima persona más alta hasta la j-ésima persona más alta; lamentablemente, los valores de las estaturas están desordenados.

Dados un arreglo *datos* con las N estaturas y los valores i y j, escribe un algoritmo en pseudocódigo que tenga tiempo esperado O(N) y que retorne las estaturas en el rango $[i, j]$ (no importa si las retorna desordenadas).

P.ej., si *datos* = [1.80, 1.65, 1.79, 1.56, 1.57, 1.70, 1.76, 1.66, 1.86, 1.92], $i = 3$, $j = 7$, entonces el output del algoritmo debe ser 1.65, 1.79, 1.70, 1.76, 1.66.

Respuesta:

Si utilizamos QuickSelect para encontrar el elemento en la posición i del arreglo vamos a tener todas las alturas mayores a la derecha y las menores a la izquierda. Dado esto podemos usar QuickSelect en el sub-arreglo datos[i+1:] y encontrar la posición j. Esto a su vez hace que los elementos a su izquierda sean menores a datos[j]. Por lo tanto, los elementos entre las posiciones i y j son los elementos buscados **[4ptos]**. La complejidad esperada del algoritmo es O(N) ya que QuickSelect toma tiempo esperado O(N) y estamos usando ese algoritmo 2 veces **[2ptos]**.