Programación dinámica

Nicolás Roulet

Ejercicio

Bondiola

El Magnate de la Bondiola piensa reventar la Costanera instalando puestos en la franja que está al sur de Ciudad Universitaria. Tiene una lista de n posibles lugares donde ubicar sus puestos. Para cada posible lugar conoce la distancia d_i en metros desde la puerta del Pabellón 1 de Ciudad Universitaria, y una estimación $b_i > 0$ del beneficio que le daría un puesto instalado ahí ($i = 1, 2, ...n$). Esta información está ordenada de forma creciente por distancia. El empresario tiene dinero suficiente para instalar los n puestos.

Sin embargo, sabe que si dos puestos están muy cerca, el beneficio de cada uno se reduce, de modo que no quiere que ningún par de puestos instalados quede a menos de m metros de distancia uno de otro. Esto significa que si dos puestos $i_1 < i_2$ son instalados, entonces debe cumplirse que $d_{i_2} - d_{i_1} \geq m$.
Diseñar un algoritmo que determine el mayor beneficio estimado que se puede obtener con la instalación de los puestos. El algoritmo debe tener complejidad temporal $\mathcal{O}(n^2)$ y estar basado en programación dinámica. Mostrar que el algoritmo propuesto es correcto y determinar su complejidad (temporal y espacial).

El problema

Resumiendo

Tenemos dos listas de números, una de distancias y otra de beneficios. Queremos elegir una subsecuencia de ambas tal que maximice el beneficio y no tenga dos elementos a distancia menor que m.

Tenemos dos listas de números, una de distancias y otra de beneficios. Queremos elegir una subsecuencia de ambas tal que maximice el beneficio y no tenga dos elementos a distancia menor que m.

Entrada: ¿Qué nos dan?

d	$[1, 3, 6, 10, 12]$
b	$[3, 4, 1, 2, 5]$
m	3

El problema

Resumiendo

Tenemos dos listas de números, una de distancias y otra de beneficios. Queremos elegir una subsecuencia de ambas tal que maximice el beneficio y no tenga dos elementos a distancia menor que m.

Entrada: ¿Qué nos dan?

d	$[1, 3, 6, 10, 12]$
b	$[3, 4, 1, 2, 5]$
m	3

Salida: ¿Qué nos piden?

10 (resultado de elegir los puestos a distancias 3, 6 y 12).

Los de distancia 1 y 3 no pueden ir ambos, lo mismo para 10 y 12.

Ideas

Como lo encaramos?
Acá dice "Programación Dinámica"...

$$\downarrow$$

Ideas

Vayamos resolviendo subproblemas. Si ya resolvimos el beneficio máximo para los puestos con índices menores a i, cómo calculamos el i-ésimo?

Ideas

Vayamos resolviendo subproblemas. Si ya resolvimos el beneficio máximo para los puestos con índices menores a i, cómo calculamos el i-ésimo?

Hay dos opciones: o utilizamos el i-ésimo lugar para poner un puesto, o no lo usamos.

- El benefico máximo que podemos conseguir usando el i-ésimo puesto es b_i más el beneficio máximo de usar los puestos a distancia menor que $d_i - m$.
- Si no usamos el i-ésimo, entonces es el beneficio máximo hasta $i - 1$

Tomamos el máximo entre los dos.
Si sólo tenemos un puesto, el beneficio máximo se obtiene usándolo.

La función

Nuestra función está definida en el rango $1..n$. $f(i)$ indica el beneficio máximo que se puede obtener considerendo los primeros i lugares posibles para poner puestos.

La función

Nuestra función está definida en el rango $1..n$. $f(i)$ indica el beneficio máximo que se puede obtener considerendo los primeros i lugares posibles para poner puestos.

$$f(1) = b[1]$$

Sea $j = max\{1 <= j <= i | d_i - d_j >= m\}$

$$f(i) = \begin{cases} max(f(i-1), b[i] + f(j)) & \text{si } \exists j \\ max(f(i-1), b[i]) & \text{si no} \end{cases} \qquad \forall 1 < i \leq n$$

Primer solución

```
int maximoBeneficio(d, b, m)
    n = |d|
    res = vector(n)
    res[0] = b[0]
    para i = 1..n-1:
        j = i
        mientras j >= 0 y d[i] - d[j] < m:
            j--
        si j >= 0:
            res[i] = max(res[i-1], b[i] + res[j])
        si no:
            res[i] = max(res[i-1], b[i])
    return res[n-1]
```

Primer solución

```
int maximoBeneficio(d, b, m)
    n = |d|
    res = vector(n)
    res[0] = b[0]
    para i = 1..n-1:
        j = i
        mientras j >= 0 y d[i] - d[j] < m:
            j--
        si j >= 0:
            res[i] = max(res[i-1], b[i] + res[j])
        si no:
            res[i] = max(res[i-1], b[i])
    return res[n-1]
```

Complejidad: $\mathcal{O}(n^2)$

¿Como mejorarlo?

Hay un costo $\mathcal{O}(n)$ inevitable producto de recorrer los arreglos. Podemos intentar mejorar el ciclo interno.
Lo que hace ese ciclo es buscar el máximo índice menor que i que esté a distancia mayor o igual a m de i. Como ese máximo es siempre creciente en cada ciclo, podemos buscarlo a partir del máximo de la iteración anterior. Entonces, en total sólo recorreríamos una vez el arreglo. Veamos cómo implementar eso.

Solución

```
int maximoBeneficio(d, b, m)
    n = |d|
    res = vector(n)
    res[0] = b[0]
    distm = -1
    para i = 1..n-1:
        mientras d[i] - d[distm + 1] >= m:
            distm++
        si distm >= 0:
            res[i] = max(res[i-1], b[i] + res[distm])
        si no:
            res[i] = max(res[i-1], b[i])
    return res[n-1]
```

Solución

```
int maximoBeneficio(d, b, m)
    n = |d|
    res = vector(n)
    res[0] = b[0]
    distm = −1
    para i = 1..n−1:
        mientras d[i] − d[distm + 1] >= m:
            distm++
        si distm >= 0:
            res[i] = max(res[i−1], b[i] + res[distm])
        si no:
            res[i] = max(res[i−1], b[i])
    return res[n−1]
```

Complejidad: $\mathcal{O}(n)$

Ejemplo

d:	1	3	6	10	12
b:	3	4	1	2	5
res:					

distm

Ejemplo

d:	1	3	6	10	12
b:	3	4	1	2	5
res:	3				

distm

Ejemplo

d:	1	3	6	10	12
b:	3	4	1	2	5
res:	3	4			

distm

Ejemplo

d:	1	3	6	10	12
b:	3	4	1	2	5
res:	3	4	5		

distm

Ejemplo

d:	1	3	6	10	12
b:	3	4	1	2	5
res:	3	4	5	7	

distm

Ejemplo

d:	1	3	6	10	12
b:	3	4	1	2	5
res:	3	4	5	7	10

distm

¿Preguntas?