Práctica 1 de Algorítmica (Programación dinámica)

La subsecuencia creciente más larga

Curso 2017-2018

El problema de la subsecuencia creciente más larga

Descripción del problema

Dada una secuencia de N números enteros:

$x_1, x_2, \ldots, x_N$

nos piden encontrar una subsecuencia:

$x_{i_1}, x_{i_2}, \ldots x_{i_k}$ (donde $1 \leq i_1 < i_2 < \cdots < i_k \leq N$)

que sea estrictamente creciente, es decir, $x_{i_1} < x_{i_2} < \cdots < x_{i_k}$.

Además, nos piden encontrar alguna de las subsecuencias crecientes de mayor longitud (decimos *alguna de* porque no tiene por qué ser única).

Por ejemplo, para la secuencia $10, 3, 5, 12, 7, 20, 18$ una posible subsecuencia sería $10, 3, 20$ pero ésta no es creciente. Un ejemplo de secuencia creciente es $10, 12, 20$ pero ésta tampoco es la de mayor longitud puesto que es de longitud 3 y existen otras de mayor longitud: $3, 5, 12, 20$ es de mayor longitud (porque no hay otras de longitud mayor a 4) pero no es la única ya que $3, 5, 12, 18$ también es creciente y de la misma longitud.

El problema de la subsecuencia creciente más larga

Este problema se puede resolver por programación dinámica. Aunque esta forma de resolver el problema no es asintóticamente óptima (tiene un coste $O(N^2)$ y se conocen soluciones con coste $O(N \log N)$) resulta bastante sencillo de resolver y es un buen ejemplo para practicar con la técnica de programación dinámica.

Como es habitual en la resolución de problemas mediante la técnica de programación dinámica, vamos a preocuparnos inicialmente del problema de encontrar la **longitud** de la subsecuencia creciente más larga y no de qué subsecuencia concreta da lugar a esa longitud máxima.

Solamente cuando hayamos resuelto el problema de manera iterativa es cuando plantearemos recuperar la subsecuencia mediante la técnica conocida generalmente como *recuperar el camino*.

El problema de la subsecuencia creciente más larga

Ejercicio 1 (a realizar en papel)

Volviendo a la secuencia del ejemplo: $10, 3, 5, 12, 7, 20, 18$

Nuestro objetivo es calcular, para cada elemento de la secuencia, cuál es la *longitud* de una subsecuencia estrictamente creciente que termina en dicho elemento.

Para ello debes plantear una ecuación recursiva donde:

- El primer elemento solamente puede formar parte de una secuencia de longitud 1.
- En general, una subsecuencia creciente que termina en x_N tiene como penúltimo elemento un elemento x_i con $i < N$ tal que $x_i < X_N$.

Ejercicio 2

Realiza la transformación de recursiva a iterativa implementado un algoritmo iterativo que calcule, para cada elemento de la secuencia, la longitud de la mayor subsecuencia creciente que termina en cada posición.

Por ejemplo, para la siguiente entrada:

```
ejercicio2([210, 816, 357, 107, 889, 635, 733, 930, 842, 542])
```

El ejercicio 2 debería calcular el siguiente vector:

```
>>> [1, 2, 2, 1, 3, 3, 4, 5, 5, 3]
```

Ejercicio 3

La subsecuencia creciente de mayor longitud no tiene que terminar necesariamente en el último elemento del vector. Una vez resuelto los ejercicios 1 y 2, es obvio que la mayor longitud se puede obtener calculando el máximo del vector generado en el ejercicio anterior.

```
>>> max([1, 2, 2, 1, 3, 3, 4, 5, 5, 3])
5
>>> ejercicio3([210, 816, 357, 107, 889, 635, 733, 930, 842, 542])
5
```

Ejercicio 4

Una vez resuelto el problema de obtener la longitud de la subsecuencia creciente más larga, vamos a recuperar una subsecuencia de dicha longitud. Tienes 2 opciones:

- Utilizar punteros hacia atrás o *backpointers*: consiste en crear un vector paralelo al vector solución donde nos apuntamos, para cada elemento de la secuencia, cuál sería su predecesor en la subsecuencia creciente más larga. Una vez terminado el proceso de programación dinámica, basta con recuperar la secuencia desde su último elemento hacia atrás empezando por la posición donde la longitud en el vector solución es máxima.

- Calcular el camino sin punteros hacia atrás: Se trata de realizar el mismo recorrido hacia atrás sin utilizar un vector auxiliar. En ese caso, hay que recalcular, solamente para los elementos de la subsecuencia buscada, su predecesor.

En cualquiera de los dos casos, el resultado de la función sería similar a este ejemplo:

```
>>> ejercicio4([210, 816, 357, 107, 889, 635, 733, 930, 842, 542])
[210, 357, 635, 733, 842]
```

Una pequeña reflexión (ejercicio opcional)

El problema de encontrar la subsecuencia creciente de mayor longitud está directamente relacionado con otros problemas bien conocidos en programación dinámica:

- La subsecuencia común más larga (Longest Common Subsequence o LCS).
- El camino más largo en un grafo acíclico.

Intenta razonar cómo se puede expresar el problema que nos ocupa en términos de alguno de estos dos problemas.

Mejorando el algoritmo (ejercicio muy opcional)

El algoritmo que se os ha propuesto tiene coste cuadrático, con lo que resulta mucho peor que otra solución conocida con coste $O(N \log N)$).

El coste dominante está en la parte de buscar la mayor longitud entre todos los elementos anteriores en el vector que tengan un valor inferior. Se propone utilizar un vector auxiliar `menores` que almacene, para cada posible longitud de la subsecuencia creciente, el **menor** valor asociado a una secuencia de dicha longitud. Ejemplo:

Utilizando la secuencia anterior $10, 3, 5, 12, 7, 20, 18$, asignamos el valor 10 a la secuencia de longitud 1 (`menores = [10]`) y procesamos el resto de elementos:

- Cuando visitamos 3, al ser menor que 10 no podemos construir una secuencia creciente de longitud 2 pero actualizamos el menor valor asociado a una secuencia de longitud 1, que pasa a ser 3 (`menores = [3]`).
- Cuando visitamos 5, lo comparamos directamente con 3 mirando únicamente el vector `menores`. Como 5 es mayor, sabemos que la longitud asociada será 2 y actualizamos el vector `menores` que pasará a ser `[3,5]`.
- Al visitar `12` buscamos el primer valor en `menores`, desde el final, que sea menor que `12`, y éste es `5`, con lo que podemos asociar a `12` la longitud 3 y añadimos 12 a `menores`.

El problema de la subsecuencia creciente más larga

- Al procesar `7` buscamos de nuevo en `menores=[3,5,12]` desde el final, el primer valor inferior es `5`, con lo que actualizamos `menores` a `[3,5,7]` y asociamos a `7` la longitud 3.
- Y así sucesivamente...

Con esto el bucle interno del algoritmo ha de recorrer el vector `menores` y no el vector original, y además basta con parar al llegar a un valor inferior al que estamos visitando.

A poco que pienses verás que el vector `menores` permanece siempre ordenado (es estrictamente creciente). De hecho, la solución óptima se basa en utilizar la búsqueda dicotómica en este vector.