Subsecuencia creciente más larga VS subsecuencia decreciente más larga

Son equivalentes: si hay un algoritmo que resuelve uno (A) de ellos, se obtiene un algoritmo para resolver el otro (B) con un costo adicional de O(n).

Se traduce una instancia de B a una instancia de A. Luego usa el algoritmo para resolver la instancia de A. Finalmente obtiene la solución de A a solución de B

Sea $S=\{A_1, A_2,\ldots, A_n\}$ la secuencia del problema B y $C=\max S$ entonces $S'=\{C-A_1, C- A_2,\ldots, C-A_n\}$ es una secuencia traducida para el problema A.

Cualquier subsecuencia $\{A_{i1}, A_{i2},\ldots, A_{ik}\}$ de S es creciente (decreciente) sii $\{C-A_{i1}, C-A_{i2},\ldots, C-A_{ik}\}$ es subsecuencia decreciente (creciente) en S'.

Subsecuencia creciente más larga

Determinar la subsecuencia creciente más larga de una sucesión de números.

Ejemplo:

S = { 9,5,2,8,7,3,1,6,4)

Las subsecuencias más largas son {2,3,4} o {2,3,6 }

Para construir un algoritmo de programación dinámica definimos:

l_i = *longitud de la secuencia creciente más larga que termina con el número* s_i

p_i = *predecesor de* s_i *en la secuencia creciente más larga que termina con el número* s_i

Vale el principio de optimalidad en este caso?. Cómo se expresaría?.

Relación de recurrencia:

$$l_0 = 0$$

$$l_i = \max_{j<i} l_j + 1, \qquad \text{con } s_j < s_i$$

La solución al problema la da

$$\max_{1 \le i \le n} l_i$$

sucesión	9	5	2	8	7	3	1	6	4
longitud	1	1	1	2	2	2	1	3	3
predecesor	-	-	-	2	2	2	-	3	3

Complejidad?

$O(n^2)$ (se podría implementar en $O(n \log n)$)

Cuál es la complejidad espacial?.

Cómo hacemos para tener también la sucesión y no sólo la longitud?.

En la implementación original, utilizamos tres arreglos:
S: es el arreglo de input, representa la secuencia
L: donde L[i] almacena la subsecuencia creciente más larga que termina con S[i]
P: donde P[i] es el predecesor de S[i] en la subsecuencia creciente más larga que termina con S[i].

```
Comenzar
  MAXLi:=0;
  Para i desde 1 hasta n hacer
    L[i]:=1;
    P[i]:=nulo;
    para j desde 1 hasta i-1 hacer
      Si S[j]<S[i] y L[i]<=L[j] entonces
        L[i]:=L[j]+1;
        P[i]:=j;
      Fin Si
    Fin Para
    si MAXLi<L[i] entonces
      MAXLi:=L[i];
  Fin Para
  retornar MAXLi
Fin
```

Ahora en la nueva implementación, vamos a utilizar dos arreglos auxiliares A y B que van a ir aumentando su tamaño y actualizando para informar en todo momento en A[i] el menor S[j] recorrido hasta el momento tal que P[j]=i, en B[i] almacena el índice j. El arreglo A está siempre ordenado de menor a mayor. Vean el ejemplo de la clase:

sucesión	9	5	2	8	7	3	1	6	4
longitud	1	1	1	2	2	2	1	3	3
predecesor	-	-	-	2	2	2	-	3	3

Inicialmente

A

B

Después de recorrer el primer elemento

A 9

B 1

Después de recorrer los primeros 2 elementos

A 5

B 2

Después de recorrer los primeros 3 elementos

A 2

B 3

Después de recorrer los primeros 4 elementos

A 2 8

B 3 4

Después de recorrer los primeros 5 elementos

A 2 7

B 3 5

Después de recorrer los primeros 6 elementos

A 2 3

B 3 6

Después de recorrer los primeros 7 elementos

A 1 3

B 7 6

Después de recorrer los primeros 8 elementos

A 1 3 6

B 7 6 8

Después de recorrer todos los elementos

A 1 3 4

B 7 6 9

Entonces con estos dos arreglos, podemos en cada paso tomar un nuevo elemento S[i], hacer una búsqueda binaria (O(log n)) en A para ver en qué posición debería estar S[i], llamamos la posición resultante j, entonces L[i]:=j y P[i]:=B[i-1] si i>1 sino P[i]:=nulo. A y B se pueden actualizar de la siguiente manera: si A tiene tamaño menor que j o S[i]<A[j], A[j]:=S[i] y B[j]:=i.

Claramente esta implementación tiene complejidad O(n log n). El arreglo A en realidad no hace falta porque A[i]=S[B[i]](con B alcanza).