

39 + 8

a) Entrada: un arreglo $A[0..N]$ que cumpla con las condiciones:

- a. el elemento $A[0]$ es mayor o igual a $A[1]$
- b. el elemento $A[N-1]$ es menor o igual a $A[N]$
- c. $N \geq 0$
- d. El arreglo contiene números

Salida: un elemento $A[n]$ el cual cumple las condiciones:

- ~~5~~ a1. $A[n-1] \geq A[n]$ y $A[n] \leq A[n+1]$
- a2. $0 \leq n \leq N$
- a3. $A[n] \in A[0..N]$, es decir $A[n]$ es un elemento en $A[0..N]$ que se considera un mínimo local según la condición a1

Para este problema se pueden tener en cuenta unas instancias.

- $A = []$ y $N=0$ ← este arreglo no tiene mínimos locales
- $A = [2, 2]$ y $N=2$ ← posee mínimos locales, se cumple a, b, a1 ya que son iguales. $A[1]=2$ es una solución
- $A = [3, 1, 4]$ y $N=3$ ← tiene un mínimo local, $A[1]=1$ es una solución

Se puede asumir que solo puede tener al menos dos elementos en el arreglo y esos dos elementos son iguales para poder cumplir la propiedad a y b en la entrada.

Técnicamente el arreglo vacío no cumple las especificaciones, así que la respuesta es que no hay mínimos locales, y si el arreglo tiene un elemento, el mismo es un mínimo, pero no cumple con las especificaciones de la entrada.

- El problema no especifica que se deben encontrar TODOS los elementos, ya que si se hace, se debe recorrer todo el arreglo en $O(n)$, mientras que el problema busca 1 mínimo local en tiempo $O(\log n)$.

Punto 1

b. El algoritmo se asume que las entradas en si son validas, ~~pero~~ en la práctica se puede decir que hay un main() el cual toma los arreglos y verifica si es valido o no, por ende en el algoritmo mininolocal se asumen que las entradas son validas.

local Minimum (arreglo [0..N]): Código

```

1. n = len(arreglo)
2. mitad = ⌊n/2⌋
3. if (n == 3 and arreglo[0] >= arreglo[1] and arreglo[1] <= arreglo[2]):
    return arreglo[1]
4. if (mitad + 1 == n and arreglo[mitad - 1] >= arreglo[mitad]):
    return arreglo[mitad]
5. if (arreglo[mitad] < arreglo[mitad + 1]):
    return localMinimum(arreglo[0:mitad + 1])
6. else:
    return localMinimum(arreglo[mitad:n - 1])

```

no es necesario, pues ya pone de la especificación.

al copiando el arreglo. Toma más espacio del necesario.

Explicación del código

En las líneas 1 y 2 simplemente se busca obtener el valor de la mitad del arreglo para verificar luego arreglos mucho mas pequeños. La condición en la linea 3 me dicta que si el arreglo tiene 3 elementos los cuales son las condiciones de la entrada misma, si se cumple es porque el segundo elemento del arreglo es el minimo local. En la linea 4 buscamos la condición de cuando el arreglo tiene 2 elementos ya sea por particiones o naturalmente sea la entrada. Esto claro está para una entrada correcta que se verifica antes de enviar el arreglo. De 2 elementos diferentes. Esta linea funciona gracias a la linea 5. ya que el elemento mitad+1 sabemos que es mayor al elemento mitad. Esto cumple la propiedad del arreglo. Ahora solo resta encontrar que mitad-1 es mayor o igual que mitad, lo que nos da un minimo local.

Continuación punto 1.b

En caso de ser mitad mayor o igual a mitad +1, igual se cumple la propiedad del arreglo y la recursión encontrará entonces la solución ya que esa mitad pasa a ser el inicio de la partición y sabemos que por la derecha encuentra la solución, pues si llega a partir el arreglo y sabemos que el primer elemento es mayor a su elemento anterior, y sabemos que el primer elemento es mayor a la nueva mitad.

Punto 1.c

Con los copias del arreglo lo demostramos 8
Con los copias del arreglo lo demostramos 8

Como mencionamos arriba, tenemos entonces:

$1 < n \leq N$ ← donde marca la cantidad de elementos del arreglo y N el total de elementos inicialmente.

$A[0..N]$ ← donde el arreglo tiene elementos en la posición 0 hasta $N-1$

$0 < \text{mitad} < n \leq N$ ← mitad representa la mitad del arreglo.

$0 \leq x \leq n-1$ ← indica la posición del elemento en el arreglo.
 $\text{localminimun}(A[0..N], \text{Tamaño})$ ← opera y garantiza que hay un mínimo local. y tamaño es

Caso base

Si ($n == 2$) \wedge ($N == n$) ambos cumplen que $A[0] == A[1]$, no se hace nada.

Si ($n == 3$) \wedge $A[0] \geq A[1] \leq A[2] \wedge N == 3$, $A[1]$ es mínimo local.

Casos inductivos

Resulta que al enviar $\text{localminimun}(A[0..N], N)$, entonces se busca cuando sea mayor a 3. El algoritmo entonces busca dividir el arreglo $A[0..N]$ a la mitad y revisa si

$A[\text{mitad}] < A[\text{mitad}+1]$, se envía $\text{localminimun}(A[0..(\text{mitad}+1)], N)$ sabemos entonces que si sigue dividiendo encontramos que si en $A[0..(\text{mitad}+1)]$ $A[0]$ es el anterior a $A[\text{mitad}]$, eso convierte a $A[\text{mitad}]$ en $A[1]$ y al ser $A[\text{mitad}+1] > A[\text{mitad}]$ se cumple la condición de la entrada y $A[\text{mitad}+1] = A[2]$, caso base.

Si $A[\text{mitad}] < A[\text{mitad}+1]$ no se cumple, entonces se revisa

$\text{localminimun}(A[(\text{mitad}..N-1)], N)$, se tiene entonces que $A[\text{mitad}]$ es el $A[0]$ de la partición y $A[N]$ unto a $A[N-2]$ cumplen las condiciones de entrada, si sucede que el arreglo al dividirlo queda con $n=2$, si $A[\text{mitad}-1] \geq A[\text{mitad}]$, Es porque $A[\text{mitad}-2] > A[\text{mitad}-1]$ y/o $A[\text{mitad}+1] \geq A[\text{mitad}]$ lo que hace $A[\text{mitad}]$ sea un mínimo local.

Punto 2

11

a. se dice entonces que no se puede bailar $k+1$ durante $k + \text{wait}[k]$ y sabemos que K es la canción actual, entonces podemos asumir que $0 \leq \text{wait}[k]$ teniendo en cuenta el 0 como bailar o no la canción primera. (es decir si bailo la primera canción debo esperar $\text{Wait}[0]$).

Tambien podemos conocer el límite de k donde $0 \leq k \leq n$, donde k es el total de canciones y K la canción actual, entonces sabemos que $n-k$ es la cantidad de canciones que falta por "bailar".

Así entonces conocemos el límite de $\text{wait}[k]$, siendo $0 \leq \text{wait}[k] \leq n-k$ ya que no te pueden penalizar canciones que ya han pasado, y tampoco canciones que no estan en la "lista de competencia" (playlist).

Debemos encontrar el máximo puntaje dentro de 2 situaciones

Si se llega a bailar sumando a su espera, que puntaje se puede lograr o que puntaje se puede obtener no bailando, y verificar cual es mayor, ya que en teoría no se especifica que puntaje pueden dar ciertas canciones, lo que dejaría a un $\text{SCORE}[k] = 0$ algo posible, y $\text{SCORE}[k] + \text{wait}[n]$ no es una situación favorable, entonces debemos tener en cuenta, si bailo la canción k para así verificar cual es el mayor entre bailar esta k th canción + el máximo puntaje despues de esperar $\text{wait}[k]+1$, o no haber bailado la canción k th y bailar la siguiente. Es parecido a knapsack o la mochila.

Para que funcione entonces si se baila, $\text{wait}[k]$ debe ser la menor espera posible para acumular más puntos.

Entrada: $A[0..N]$ y $B[0..N]$ donde $0 \leq N$, es decir A y B tienen la misma cantidad de elementos.

- El arreglo A representa score, es decir, números enteros naturales $+0$
- El arreglo B representa wait, tiene números enteros naturales $+0$
- N es el número de canciones en el concurso.

Salida:

la máxima puntuación posible en $A[0..N]$ continuación en B[0..N]

Ejemplo

$A[3, 2, 1, 1, 3]$ $B[1, 3, 2, 2, 1]$

la máxima puntuación es 6 bailando la primera y última canción

Definición de función para $0 \leq k \leq N$ (Por memorización)

$\phi(k)$: "máxima puntuación obtenida que termina en $A[k]$ "
donde $k = N$

Funció objetivo: no mas, porque hay restricciones.

$\uparrow \sum_0^N A[k]$: $\phi(0)$ donde se entrega la máxima puntuación que termina en $A[0]$ donde $k = N$ puesto que $N - k$ marca el fin del concurso

Planteamiento recurrente. para $0 \leq k \leq N$

$$\phi(k) = \begin{cases} 0 & \text{para } k > N \\ \uparrow (A[k] + \phi(k+B[k]+1), \phi(k+1)) & \text{para } k \leq N \end{cases}$$

← mientras hayan canciones por bailar

bailo la canción k no bailo la canción k

4

hay que tener en cuenta que $\text{wait}[k]$ es la mejor situación es la mejor espera posible. Se ejecuta de atrás a adelante. Se retorna $\phi[0]$ con memorización en términos de la definición

The Big Dance Contest ($A[0..N-1], B[0..N-1]$):

$$\phi[N] = 0$$

for k desde $N-1$ ↓ la cantidad de canciones que faltan

$$B[k] = \uparrow (0; (B[k] \downarrow N-k))$$

$$\phi(k) = \uparrow (A[k] + \phi(k+B[k]+1), \phi(k+1))$$

return $[0]$

Parte de la justificación
porque para este caso es
mejor memorización
que tabulación

Tiempo $O(n)$ al recomer el
arraylo.

Punto 2 b. En este algoritmo es mejor utilizar memorización que tabulación ya que "la memoria" es un arreglo lineal donde se recorre solo una vez ya que los tamaños del arreglo e índices comparten el n , pero esto en tabulación es un arma de ~~doublefilo~~ doblefilo ya que se debería crear una tabla de tamaño $n \times n$, esto significa que se debería recorrer n filas y n columnas muchas veces, lo que resultaría en una solución de tiempo $O(n^2)$ ya que en recorrer las filas en tiempo $O(n)$ y las columnas, es tiempo $O(n)$ más entonces resumimos $O(n \times n) \Rightarrow O(n^2)$. Además, no es necesario resolver todos los subcasos o subproblemas, ya que no es necesario tener en cuenta cuantas posibles combinaciones tiene la primera canción, sino, específicamente, cuantas combinaciones tengo en específicas canciones. Tabulación contaría todos esos subproblemas, lo que devoraría más tiempo, ya que memorización también trabaja recursivamente mientras que tabulación es netamente iterativo.

*No se fuer
arguras*

Para tabulación se deberían hacer 2 ciclos que recorra $A[0..N]$ y otro ciclo que recorra $B[0..N]$ y vaya comparando el maximo en esas situaciones.

En mi enfoque se ahorra memoria y tiempo por recorrer el mismo arreglo y las mismas iteraciones de K , mientras que en el de tabulación compararía varias situaciones no necesarias en una tabla de $n \times n$.

Punto 3

11*3

a. En el primer día después de reboot se procesa s_1 hasta s_n . Si se hace un reboot, el día i del reboot se procesa 0 Teras. Si el ejercicio especifica que se empieza el día 1 después de haber sucedido un reboot, entonces es posible que no se tome x_1 .

Hay un "surplus" de datos donde $x_i > s_i$ para cada i . La solución óptima es hacer reboot cada dos días.

Si se tiene entonces que $x_i > s_i$, tomemos 5 como ejemplo. Para s_1 y $x_1 = 6$ tenemos entonces que $6 > 5$ y para que sea óptima hacer reboot cada dos días, la suma de $s_1 + s_2$ debe ser igual o mayor a x_1 , el problema es que a medida que s_i avance se vuelve cada vez más pequeño, por eso 6 es el mejor valor, ya que se vuelve cada vez más pequeño, por eso 6 es el mejor valor, ya que

$$5 + 1 = 6,$$

	1	2	3	4	5	6	...
x_i	6	6	6	6	6	6	
s_i	5	1	1	1	1		

Punto b.

Entrada: $A[0..n]$ y $B[0..n]$ donde $n \geq 0$

Salidad: máxima cantidad de terabytes procesados tomando

$\phi(i, j)$ $A[0..n]$ y el valor de reboot de $B[0..n]$ donde $0 \leq i \leq n$ y j son los días antes de haber hecho reboot, por ende, el reboot fue el día $i-j \leq n$.

$A[0..n]$ representa x_i

$B[0..n]$ representa s_i → el último día se debe utilizar ya que no hay ganancia al hacer reboot.

recurrencia

$$\phi(i, j) = \begin{cases} (A[n] + B[j]), & \text{si } n = i \\ \uparrow(\phi(i+1, 0)), & \text{si } i \leq n \text{ y } j \neq 0, \text{ el día después del primer reboot} \\ (A[i] + B[j]) + \phi(i+1, j+1), & \text{si } i > n \end{cases}$$

mis aplicaciones

objetivo

$$\phi(0, 0)$$

Porque $A[n] + B[j]$, porque según el problema los recursos máximos los puedes obtener luego de hacer reboot, esto significa que el último día no puede generar ningún beneficio el rebootear el sistema, por ende una solución óptima debe considerar hasta el día $n-1$, como último reboot para tomar así lo procesado en el día $A[n] + B[i]$

Si se hace un reboot entonces tenemos que "ignorar" ese día, por ende debemos tomar el $i+1$ y reiniciar j ya que el último día de reboot es el mismo.

Luego debemos tomar una cantidad hasta x_i como dice el problema, "you can only process up to x_i regardin of the speed", como estamos procesando Teras en $S[n]$ se busca entonces el mínimo entre ese día y la última vez que se hizo reboot, $\min(A[i], B[j]) + \text{distantes}$ se procesan el siguiente día $\phi(i+1, j+1)$.

$\phi(i, j) = ([\text{for } i \text{ in } n+1] [\text{for } j \text{ in } n])$

while ($i > 0$) \leftarrow se ejecuta n veces.

 while ($j < i$) \leftarrow como i va n veces, j se hace n veces.
 $\phi(i, j) = \max(\phi(i+1, 0), \min(A[i], B[j]) + \phi(i+1, j+1))$

X3
 $\begin{array}{c} j+1 \\ i+1 \end{array}$

return $\phi(0, 0)$

Como se recorre una matriz entonces la solución es
 $n \times n = n^2 \Rightarrow \underline{\underline{O(n^2)}}$