Segmento più lungo di spessore al più s (Esercizio 6.18)

Il Problema e l'algoritmo ignorante

Lo spessore di un segmento di vettore numerico $v[inf, sup)$ di v, è definito come **spess**$(v[inf, sup)) = \max_{i,j \in [inf, sup)} v[i] - v[j]$. Dare un algoritmo di complessità $\mathcal{O}(n \log n)$ che dato un vettore v di lunghezza n e un valore s, calcola il segmento più lungo di spessore minore o uguale ad s.

SUGG.: Utilizzare un'opportuna struttura dati per calcolare in modo efficiente **spess**$(v[i, j))$ conoscendo **spess**$(v[i-1, j))$ oppure **spess**$(v[i, j-1))$ oppure dare una soluzione Divide et Impera.

Al solito c'è una soluzione **esaustiva** o **ignorante** semplice: si analizzano tutti gli intervalli $[i, j)$ con $i < j$: sono $\sim n^2/2$ e il calcolo dello spessore è (in media) $\mathcal{O}(n)$, ottenendo una complessità $\mathcal{O}(n^3)$.

```
def maxSegSpess(v, s): #REQ: s>=0
    n, L = range(v), 1
    for i in range(n):
        for j in range(i+1, n):
            if max(v[i:j]) - min(v[i:j]) < s
                    and j - i > L:
                L = j - i
    return L
```

Piccole migliorìe

L'algoritmo esaustivo può essere migliorato semplicemente osservando che **max**($v[i, j+1]$) = max(**max**($v[i, j]$), $v[j]$). Similmente per il **min** e quindi per lo spessore, e quindi non serve ricalcolare daccapo i minimi e massimi per ogni intervallo.

Questo porta immediatamente a un algoritmo $\theta(n^2)$.

```
def maxSegSpess(v, s): #REQ: s>=0
    n, L = range(v), 1
    for i in range(n):
        maxs, mins = v[i], v[i]
        for j in range(i+2, n):
            maxs, mins = max(maxs, v[j-1]), min(mins, v[j-1])
            if maxs - mins < s and j - i > L:
                L = j - i
    return L
```

Sfruttando la monotonìa…

Come nel segmento di somma s (caso positivo), abbiamo che lo spessore è monotono rispetto all'inclusione di segmenti, cioè se $[inf, sup) \subseteq [inf', sup')$ allora $\mathbf{spess}(v[inf, sup)) \leq \mathbf{spess}(v[inf', sup'))$.

Quindi, se analizzo i segmenti del tipo $[i, j]$, quando $\mathbf{spess}(v[i, j)) > s$ posso escludere tutti i segmenti del tipo $[i, j']$ con $j' > j$.

Se $\mathbf{spess}(v[i, j)) < s$, posso anche evitare di riesaminare tutti i segmenti del tipo $[i', j]$ con $i' > i$, in quanto più corti.

Di conseguenza, se $\mathbf{spess}(v[i, j)) < s$, vado a vedere $\mathbf{spess}(v[i, j+1))$, viceversa se $\mathbf{spess}(v[i, j)) \geq s$, ricomincio a vedere $\mathbf{spess}(v[i+1, j))$.

A differenza delle somme, non sembra però possibile calcolare $\mathbf{spess}(v[i+1, j))$ e $\mathbf{spess}(v[i, j+1))$ a partire da $\mathbf{spess}(v[i, j))$ in $\Theta(1)$.

Algoritmo

Ecco il codice, semplice ed efficace.

Per la **terminazione**, osservate che la funzione $2n - i - j$ è positiva, perché $i \leq j < n$ e a ogni iterazione, o incremento i o incremento j e quindi decresce.

$i \leq j$ perché se $i = j$, allora lo spessore è 0, ma allora al prossimo giro, sicuramente si incrementa j (assumendo $s > 0$). Ma questo ci dà anche un upper bound al numero di iterazioni, che è $2n$.

```
def maxSegSpess(v, s): #REQ: s>=0
    n, maxL, i, j = len(v), 0, 0, 0
    while j < n:
        if spess(v, i, j) < s
            if j - i > maxL:
                maxL = j - i
            j = j + 1
        else:
            i = i + 1
    return maxL
```

Complessità e miglioramenti

Calcolando **`spess(v, i, j)`** con la funzione sotto il calcolo dello spessore è $\mathcal{O}(sup - inf)$ che a sua volta è $\mathcal{O}(n)$ in media. Ciò porta a un algoritmo di complessità $\mathcal{O}(n^2)$.

Osservare che comunque questo algoritmo ha **casi ottimi molto migliori** del precedente!

Ma è possibile calcolare massimi e minimi in tempo $\mathcal{O}(\log n)$?

In generale no, ma organizzando gli elementi del segmento $v[i, j]$ in uno **heap** oppure in un **ABR bilanciato** sì.

Se avessimo gli elementi del segmento $v[i, j]$ memorizzati in un red-black tree: max e min si calcolano in $\Theta(\log (j - i)) = \mathcal{O}(\log n)$.

Inoltre l'operazione $j = j + 1$ causerà l'inserimento dell'elemento $v[j]$ (costo $\Theta(\log (j - i)) = \mathcal{O}(\log n)$), mentre l'operazione $i = i + 1$ causerà la rimozione $v[i]$ (costo $\Theta(\log (j - i)) = \mathcal{O}(\log n)$).

```
def spessore(v, inf, sup):
    return max(v, inf, sup) - min(v, inf, sup)
```

Algoritmo con red-black tree

La funzione è del tutto analoga, a parte la gestione dell'albero R.

Essendo R un red-black tree, è bilanciato e le operazioni di calcolo del massimo, minimo, inserzione e rimozione si fanno in $\Theta(\log(j - i)) = \mathcal{O}(\log n)$.

Ognuna di queste operazioni si ripete al più $\mathcal{O}(n)$ volte, per cui l'algoritmo ha una complessità $\boldsymbol{\mathcal{O}(n \log n)}$.

```
def maxSegSpess(v, s): #REQ: s>=0
    n, maxL, i, j = len(v), 0, 0, 0
    R = newRedBlack()
    while j < n:
        if spessABR(R) < s
            if j - i > maxL:
                maxL = j - i
            j = j + 1
            insert(R, v[j])
        else:
            remove(R, v[i])
            i = i + 1
    return maxL
```

```
def spessABR(R):
# PREC: R è ABR bilanciato
    return maxABR(R) - minABR(R)
```

Soluzioni con Heap

Pur ottenendo essenzialmente le stesse complessità, è intuitivo che uno **heap** permetta di implementare una **coda con priorità** in modo **più efficiente** (sia in tempo che in spazio) **di un red-black tree**.

Dovendo calcolare massimi e minimi, dovremo avere **due heap**, un **min-heap** e un **max-heap**.

Il problema è che nell'algoritmo precedente, dobbiamo **estrarre elementi che non sono il massimo** (o il minimo), operazioni che non sappiamo fare in modo efficiente.

D'altro canto, se $v[i]$ non è il minimo [massimo] rimuoverlo non modifica lo spessore: posso lasciarlo nell'heap.

Possiamo quindi memorizzare sull'heap un record con le informazioni valore ($v[i]$) e indice (i): in rimozione rimuoverò tutti i minimi [massimi] che hanno un indice $i' \leq i$ dell'elemento $v[i]$ che sto rimuovendo (nessuno se $v[i]$ non è il minimo [massimo]).

Algoritmo con Heap

Ecco il codice. L'unica cosa interessante è la funzione ***removeMin*** [***removeMax***] che rimuove ricorsivamente tutti i minimi successivi di indice minore di *i*: il numero di estrazioni di minimo [massimo] sarà in ogni caso limitato dalla lunghezza *n* del vettore.

```
def maxSegSpess(v, s): #REQ: s>=0
    n, maxL, i, j = len(v), 0, 0, 0
    Hmin, Hmax = newMinHeap(), newMaxHeap()
    while j < n:
        if spessHeap(R) < s
            if j - i > maxL:
                maxL = j - i
            j = j + 1
            insert(Hmin, v[j], j)
            insert(Hmax, v[j], j)
        else:
            removeMin(Hmin, v[i], i)
            removeMax(Hmax, v[i], i)
            i = i + 1
    return maxL
```

```
def removeMin(H, v, i):
# PREC: H è un minHeap
    val, ind = min(H)
    if ind <= i:
        estraiMin(H)
        removeMin(H, v, i)
```

Algoritmo Divide et Impera

L'algoritmo divide et impera è concettualmente semplice: si divide il segmento in esame in 2 parti (meglio se uguali), $v[inf, m)$ e $v[m,sup]$. Ricorsivamente posso assumere di ottenere i segmenti di lunghezza massima di spessore al più s inclusi…

ma c'è una **terza possibilità**, e cioè che il **segmento ottimo sia a cavallo** dell'indice m. Calcolo questa terza possibilità e ritorno il maggiore dei 3.

Vediamo poi come calcolare il segmento a "cavallo" in tempo $\Theta(n)$, in modo di avere la relazione di ricorrenza $T(n) = 2\,T(n/2) + \Theta(n)$, che come dovrebbe essere noto risove a $\Theta(n \log n)$.

```
def maxSegSpessDeI(v, l, r, s):
    if l+1==r: return 1
    m = puntoMedio(l, r)
    Ll = maxSegSpessDeI(v, l, m, s)
    Lr = maxSegSpessDeI(v, m, r, s)
    Lc = maxSegCavallo(v, l, m, r, s)
    return max(Ll, Lr, Lc)
```

Segmento a cavallo: idea

A prima vista, questo problema non sembra più facile del problema generale, ma tutti i segmenti $[l, r)$ da considerare hanno $l<m$ e $r>m$, per cui possono essere visti come l'unione dei due intervalli non vuoti $[l, m) \cup [m, r)$.

Occorre osservare che $\mathbf{max}(v[l, r)) = \max(\mathbf{max}(v[l, m)), \mathbf{max}(v[m, r)))$. Similmente per il minimo e quindi per lo spessore.

Come ci può aiutare tutto ciò? Avendo in mente l'algoritmo che usa la monotonìa, ci **precalcoliamo tutti i massimi/minimi dei segmenti nella forma** $v[i, m)$ (si può fare in **tempo lineare** partendo da m-1 e andando verso sinistra) e anche **tutti i massimi/minimi dei segmenti nella forma** $v[m, i)$ (si può fare in **tempo lineare** partendo da m e andando verso destra).

A questo punto si cerca il segmento di spessore $< s$ di lunghezza massima, partendo dal segmento $v[l, m+1)$, "allargandosi" a destra se lo spessore è minore di s, e "stringendosi" a sinistra quando lo spessore è maggiore di s, fermandosi quando il limite sinistro raggiunge m, o quello destro raggiunge r.

Il tutto in tempo lineare in $r - l$.

Segmento a cavallo: codice

Ecco la funzione, che dovrebbe essere chiara per quanto detto nella slide precedente.

```
def maxSegCavallo(v, l, m, r, s):
# fase 1: precomputazione massimi/minimi
    maxi[m-1], mini[m-1] = v[m-1], v[m-1] # parte sinistra all'ind.
    for k in range (m-2, l-1, -1):
        maxi[k], mini[k] = max(v[k],maxi[k+1]), min(v[k],mini[k+1])
    maxi[m], mini[m] = v[m], v[m] # parte destra in avanti
    for k in range (m, r):
        maxi[k], mini[k] = max(v[k],maxi[k+1]), min(v[k],mini[k+1])
# fase 2: ricerca segmento più lungo
    L, i, j = 1, l, m
    while i < m and j < r:
        spess = max(maxi[i],maxi[j]) - min(mini[i],mini[j])
        if spess <= s and j - i + 1 > L: L = j - i + 1
        if spess <= s: j = j + 1 else: i = i + 1
    return L
```