Esercizio 2 (11 punti) Una *longest common substring* di due stringhe X e Y è una sottostringa di X e di Y di lunghezza massima. Si vuole progettare un algoritmo efficiente per calcolare la lunghezza di una longest common substring. Per semplicità si assuma che entrambe le stringhe di input abbiano stessa lunghezza n.

(a) Qual è la complessità dell'algoritmo esaustivo che analizza tutte le possibili sottostringhe comuni?

(b) Assumendo di conoscere un algoritmo che determina se una stringa di m caratteri è sottostringa di un'altra stringa di n caratteri in tempo $O(m+n)$, come si può modificare l'algoritmo del punto precedente per renderlo più efficiente?

(c) Progettare un algoritmo di programmazione dinamica più efficiente di quello del punto precedente. Sono richiesti relazione di ricorrenza sulle lunghezze (senza dimostrazione) e algoritmo bottom-up. (Suggerimento: considerare la lunghezza della longest common substring dei prefissi $X_i = \langle x_1, \ldots, x_i \rangle$ e $Y_j = \langle y_1, \ldots, y_j \rangle$ *che termina con* x_i *e* y_j, rispettivamente.)

LCS = sottostringa di lunghezza più grande comune

X è superstringa

Y è sottostringa

Entrambe hanno stessa lunghezza

$$X = \text{"}sabbatico\text{"}$$

$$Y = \text{"}abba\text{"}$$

$Y \subseteq X$ è sottostringa

(a)

$O(n^2)$ dato che entrambe hanno la stessa lunghezza e, alla peggio, devo guardare sia X che Y

(b)

M+n alla fine è un "pattern matching"

- Usiamo la programmazione dinamica per fare un unico ciclo su $for\ i = 1\ to\ n$ con uno stesso indice e verifichiamo se $a[i] \leq a[i+1]$ (è sottostringa/sottoinsieme dell'array) e itero aggiungendo la sottostringa alla soluzione ottima ($max += lcs$)

(c)

Esattamente come negli esercizi di prog. dinamica, vogliamo rendere efficiente l'operazione comune; significa quindi che uso sempre l'algoritmo base di LCS (prog. dinamica):

```
LCS(X, Y)
 1  m = X.[length]
 2  n = Y.length
 3  for i = 0 to m
 4       L[i, 0] = 0
 5  for j = 0 to n
 6       L[0, j] = 0
 7  for i = 1 to m
 8       for j = 1 to n
 9            if x_i = y_j
10                 L[i, j] = L[i − 1, j − 1] + 1
11                 B[i, j] = '↖'
12            else if L[i − 1, j] ≥ L[i, j − 1]
13                 L[i, j] = L[i − 1, j]
14                 B[i, j] = '↑'
15            else
16                 L[i, j] = L[i, j − 1]
17                 B[i, j] = '←'
18  return (L[m, n], B)
```

Qui scorriamo tutte le stringhe; creeremo quindi un algoritmo che salva già il minimo e scorre le stringhe sapendo che Y è contenuta in X e, avendola già salvata, ripete solo la stessa operazione:

```
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1] + 1
            else:
                L[i][j] = max(L[i-1][j], L[i][j-1]) // quindi è m*n
    return L[m][n]
```

Spiegazione:

1. Creiamo una matrice L di dimensioni (m+1) x (n+1), dove m e n sono le lunghezze di X e Y.
2. L[i][j] rappresenta la lunghezza della LCS considerando i primi i caratteri di X e j caratteri di Y.
3. Riempiamo la matrice usando la relazione di ricorrenza mostrata nel commento.
4. Il valore in L[m][n] sarà la lunghezza della LCS.

Questo algoritmo ha una complessità temporale di O(mn) e spaziale di O(mn), che è significativamente più efficiente dell'approccio esaustivo.

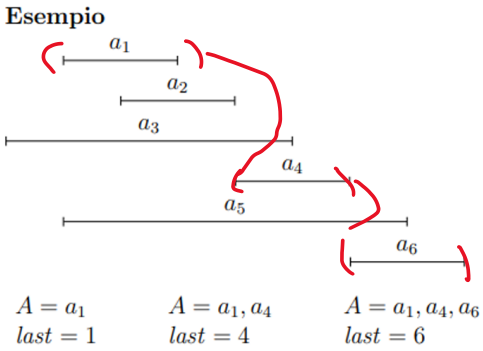
Per un approccio bottom-up, potremmo iniziare considerando sottostringhe di lunghezza 1, poi 2, e così via, memorizzando e riutilizzando i risultati intermedi per calcolare quelli successivi.

Struttura logica greedy

- Array di lunghezza n
- If condizione dell'algoritmo (salvi la cosa che ti serve)

$$if(min > \cdots)\ \min =$$

```
GREEDY-SEL(S, f)
1  n = S.length
2  A = {a_1}
3  last = 1 // indice dell'ultima attività selezionata
4  for m = 2 to n
5      if s_m ≥ f_last
6          A = A ∪ {a_m}
7          last = m
8  return A
```



- Di solito, nei greedy salvo almeno un elemento per fare "pivot"
  - Quindi vuol dire che mi salvo la scelta buona partendo da una certa
- Quindi itero
- La scelta greedy è normalmente una condizione
  - Nella condizione, salvo la scelta ottima

- Nel greedy (a livello teorico) si parla di “proprietà di sottostruttura ottima”
- Vuol dire → la proprietà vale perché lo dimostro io e ho fatto la scelta giusta

Dimostrazione di algoritmi greedy → cut and paste

L’algoritmo dice:

- Scorri tutte le attività
- Controlla se la prima attività che inizia dopo la scelta ottima (S_m) è considerabile come attività ottima (vuol dire – “riesco ad attaccargliela”)
  - Allora salvala come indice
  - E riparti da lì col greedy