RIPASSO

Fissata una funzione *g(n)* possiamo dire:

$O(g(n)) = \{ f(n) \mid \exists\, c \in \mathbb{R}, n_0 \in \mathbb{N} \quad : \forall\, n \geq n_0\ f(n) \leq c * g(n) \}$

Una funzione **f(n)** è detta *O grande di g(n)* se esiste una costante reale *c* tale che, per ogni punto successivo ad un certo punto n_0, **f(n)** assume valori minori o uguali ai valori assunti da *g(n) moltiplicata per c*.

$\Omega(g(n)) = \{ f(n) \mid \exists\, c \in \mathbb{R}, n_0 \in \mathbb{N} \quad : \forall\, n \geq n_0\ c * g(n) \leq f(n) \}$

Una funzione **f(n)** è detta *Omega grande di g(n)* se esiste una costante reale *c* tale che, per ogni punto successivo ad un certo punto n_0, **f(n)** assume valori maggiori o uguali ai valori assunti da *g(n) moltiplicata per c*.

E ancora
$\Theta(g(n)) = \{ f(n) \mid \exists\, c_1, c_2 \in \mathbb{R}, n_0 \in \mathbb{N} \quad : \quad \forall\, n \geq n_0\ c_1 * g(n) \leq f(n) \leq c_2 * g(n) \}$

Una funzione **f(n)** è detta *Theta di g(n)* se esistono due costanti reali c_1 e c_2 tali che, per ogni punto successivo ad un certo punto n_0, **f(n)** assume contemporaneamente

- valori maggiori o uguali ai valori assunti da *g(n) moltiplicata per* c_1
- valori minori o uguali ai valori assunti da *g(n) moltiplicata per* c_2

In parole povere, una funzione **f(n)** è detta *Theta di g(n)* se **f(n)** è contemporaneamente sia *Omega grande di g(n)* che *O grande di g(n)*.

PROGRAMMAZIONE DINAMICA

Abbiamo visto nel corso precedente come spesso cerchiamo la soluzione migliore di un algoritmo provando sia una soluzione iterativa che una ricorsiva, ma che facciamo quando nessuna di queste due da una soluzione accettabile?
L'idea su cui si basa la programmazione dinamica è evitare di calcolare più volte lo stesso risultato *(come accade a volte negli algoritmi ricorsivi, ad esempio con Fibonacci)* calcolandolo una sola volta per poi memorizzarlo per quando sarà necessario nuovamente.

WEIGHTED INTERVAL SCHEDULING

Consideriamo un insieme di n attività

{ 1, ..., n} $n \in \mathbb{N}$ $\forall$ attività i $\in$ { 1, ..., n } abbiamo $\begin{cases} s_i & \text{tempo inizio attività i} \\ f_i & \text{tempo fine attività i} \\ v_i & \text{valore/peso attività i} \end{cases}$

Consideriamo due attività *i* e *j* compatibili se queste non si sovrappongono, ovvero se **[s_i, f_i) $\cap$ [s_j, f_j) = $\emptyset$**

Definiamo le seguenti funzioni:

1. C : $\wp$({1, ..., n}) $\rightarrow$ { true, false }
   $\forall$ A $\subseteq$ {1, ..., n} C(A) = $\begin{cases} true & \text{se A contiene attività fra loro mutualmente compatibili} \\ false & \text{altrimenti} \end{cases}$
2. v : $\wp$({1, ..., n}) $\rightarrow \mathbb{R}$
   $\forall$ A $\subseteq$ {1, ..., n} v(A) = $\begin{cases} \sum_{i \in A} v_i & \text{se A} \neq \emptyset \\ 0 & \text{se A} = \emptyset \end{cases}$

Il problema WIS ci richiede di trovare, fra tutti i sottoinsiemi di attività mutualmente compatibili, quello con valore v massimo *(è dunque un problema di ottimizzazione)*.

ISTANZA
Un'istanza di tale problema è data dal numero di attività *n* e dalle triple associate ad ognuna delle *n* attività.

SOLUZIONE: S $\subseteq$ {1, ..., n} t.c. C(S) = true $\wedge$ v(S) = max{ v(A) } con A $\subseteq$ {1, ..., n}

Per risolvere un problema di programmazione dinamica dobbiamo innanzitutto individuare i sotto problemi più piccoli. Il generico sotto problema di WIS, identificato da *i : 0* $\leq i \leq$ *n* è nella forma:

Dato X_i = {1, ..., i} trovare $S_i \subseteq X_i$ t.c. C(S_i) = ture $\wedge$ v(S_i) = max{ v(A) } con A $\subseteq X_i$

Assumiamo ora di aver già risolto tutti i sotto problemi più piccoli di *i* e cerchiamo di capire come calcolare S_i e v(S_i).

Assumiamo anche che le attività siano ordinate per tempo di fine e definiamo
p(i) = max{ j | j < i $\wedge$ j è compatibile con i }

Esempio:
1 ——
2 ——————
3 ———
4 ————
5 ——————————
6 ———

v_i = 10, 2, 8, 1, 1, 3
p(i) = 0, 0, 1, 2, 1, 4

Per calcolare S_i avremo quindi due casi:

1. i $\in S_i$ allora $S_i = S_{p(i)} \cup$ { i }
   $v(S_i) = v(S_i) + v_i$
2. i $\notin S_i$ allora $S_i = S_{i-1}$
   $v(S_i) = v(S_{i-1})$

Possiamo ora definire le equazioni di ricorrenza.

CASO BASE

i = 0
$X_0 = \emptyset \implies S_0 = \emptyset$ e $v(S_0) = 0$

CASO PASSO

i > 0
$v(S_i) = \max\{ v(S_{p(i)}) ; v(S_{i-1}) \}$

$$S_i = \begin{cases} S_{p(i)} \cup \{i\} & \text{se } v(S_{p(i)}) + v_i \geq v(S_{i-1}) \\ S_{i-1} & \text{altrimenti} \end{cases}$$

Detto questo, è facile creare un algoritmo ricorsivo basato su queste equazioni, il problema di tale algoritmo però è che finirebbe per calcolare più volte lo stesso risultato portando il tempo ad essere esponenziale. La programmazione dinamica ci permette di memorizzare i risultati parziali dei sotto problemi più piccoli in modo da riutilizzarli senza doverli calcolarli nuovamente.

DEFINIZIONE VARIABILI

Definiamo quindi M[i] $\forall$ i t.c. $0 \leq i \leq n$ come la soluzione del sotto problema i-esimo, ovvero come il massimo valore che posso ottenere prendendo un insieme di attività mutualmente incompatibili considerando solo le prime *i* attività.

ALGORITMO

Scriviamo quindi l'algoritmo WIS definito come

```
WIS(){
   M[0] = 0
   for i = 0 to n
      M[i] = max(M[p[i]] + v[i], M[i-1])
   return M
}
```

Nell'esempio fatto prima l'algoritmo farebbe i seguenti passaggi
M[1] = max(M[0] + 10, M[0])
M[2] = max(M[0] + 2, M[1])
M[3] = max(M[1] + 8, M[2])
M[4] = max(M[2] + 1, M[3])
M[5] = max(M[1] + 1, M[4])
M[6] = max(M[4] + 3, M[5])

0	1	2	3	4	5	6
0	10	10	18	18	18	21

Possiamo poi stampare gli intervalli attraverso il seguente algoritmo

```
PrintW(i)
   if i != 0
      if v[i] + M[p[i]] > M[i - 1]:  print i  PrintW(p[i])
      else:    PrintW(i - 1)
```

LONGEST COMMON SUBSEQUENCE

Per alcuni esercizi è possibile trovare immediatamente una soluzione in programmazione dinamica, mentre per altri bisogna risolvere un problema associato che ci possa consentire di arrivare alla soluzione.

SEQUENZE

Definiamo **$X = \langle x_1, ..., x_n \rangle$** una sequenza.
Una sottosequenza **$Z = \langle z_1, ..., z_n \rangle$** con $k \leq n$ è una sequenza costituita dagli indici presi in ordine strettamente crescente dei caratteri di X.

Esempio:
$X = < A, C, D, B, A >$

$Z_0 = < A, B > = < 1, 4 >$ è una sottosequenza di X
$Z_1 = < A, B, B, A > = < 1, 4, 4, 5 >$ non è una sottosequenza di X, in quanto X non contiene due B
$Z_2 = < A, B, C, D > = < 1, 4, 2, 3 >$ non è una sottosequenza di X, in quanto gli indici non sono presi in ordine strettamente crescente.

Consideriamo ora due sequenze $X = \langle x_1, ..., x_n \rangle$ e $Y = \langle y_1, ..., y_m \rangle$;
diremo che Z è ***sottosequenza comune*** se è contemporaneamente sottosequenza di X e di Y.

Esempio:
$X = < A, C, D, B >$ $\quad$ $Y = < D, C, B, A >$

$Z_0 = < C, B >$ è sottosequenza comune a X e Y.

LCS

Il problema della LCS vuole trovare una sottosequenza comune a due sequenze date che sia la più lunga possibile *(dunque anche questo è un problema di ottimizzazione)*.

Proviamo a fare un ragionamento di tipo ricorsivo

Confrontiamo, nel caso generico, x_i con y_j, caratteri i-esimo e j-esimo delle sequenze X e Y

- Se $x_i = y_j$ $\quad$ $LCS(< x_1, ..., x_{i-1} >, \ < y_1, ..., y_{j-1} >) + 1$
  se i caratteri sono uguali è ovvio che mi conviene considerarli come facenti parte di Z
- Se $x_i \neq y_j$ $\quad$ $\max\{ LCS(< x_1, ..., x_{i-1} >, \ < y_1, ..., y_j >), LCS(< x_1, ..., x_i >, \ < y_1, ..., y_{j-1} >) \}$
  se sono diversi, cerco di capire quale dei due caratteri mi conviene scartare

Definiamo quindi il problema:
ISTANZA: $X = \langle x_1, ..., x_n \rangle$, $Y = \langle y_1, ..., y_m \rangle$

SOLUZIONE: La sottosequenza comune ad X e Y che sia la più lunga possibile.

SOTTOPROBLEMI

Cerchiamo di suddividere il problema in sottoproblemi di taglia minore, definendo quindi il generico sottoproblema (i, j) come tale: la lunghezza della più lunga sottosequenza comune considerando solo i primi *i* caratteri di X ed i primi *j* caratteri di Y.
Avremo quindi (m + 1)(n + 1) sottoproblemi.

DEFINIZIONE VARIABILI

Definiamo M[i][j] come la soluzione al sottoproblema di taglia (i, j), ovvero come la lunghezza massima della LCS che riesco ad ottenere considerando solo i primi *i* caretteri di X ed i primi *j* caratteri di Y.
Abbiamo detto di avere (n + 1)(m + 1) sottoproblemi, di conseguenza la dimensione della nostra matrice sarà M[n + 1][m + 1].

CASO BASE

Se i = 0 ∨ j = 0
$LCS(x_i, y_j) = 0$

Ho in totale (m + n + 1) casi base.

CASO PASSO

Se i > 0 ∧ j > 0

$$LCS(x_i, y_j) = \begin{cases} LCS(x_{i-1},\ y_{j-1}) + 1 & se\ x_i = y_j \\ max\{LCS(x_{i-1}, y_j),\ LCS(x_i, y_{j-1})\} & se\ x_i \neq y_j \end{cases}$$

SOLUZIONE: La soluzione *(parziale)* la troveremo nella cella M[n][m]
La soluzione è parziale nel caso ci interessi sapere anche quale sia la LCS oltre alla sua lunghezza, in tal caso possiamo o salvarci in un'altra matrice (n + 1)(m + 1) gli spostamenti fatti nella prima matrice (se il risultato (i,j) proviene dalla diagonale o da riga/colonna) per poi usarla per sapere quali caratteri abbiamo usato, o ripercorrere la matrice originale per trovare i simboli (operazione più costosa in termini computazionali).

ALGORITMO

```
LCS(X, Y) {
CASO BASE
   for i = 0 to n
      M[i][0] = 0
   for j = 0 to m
      M[0][j] = 0

CASO PASSO
   for i = 1 to n
      for j = 1 to m
         if x[i] == y[j]
            M[i][j] = M[i - 1][j - 1] + 1
         else
         M[i][j] = max( M[i][j - 1], M[i - 1][j] )

   return M[n][m]
}
```

TEMPO E SPAZIO

Il tempo di calcolo T(n, m) sarà
T(n, m) = (n + m) + (n * m) = Θ(n * m)
Se includessimo anche la risalita per trovare la sequenza il tempo asintomatico non cambierebbe *(... + (n + m))*.

Lo spazio S(n, m) sarà
S(n, m) = (n * m) = Θ(n * m)
Se includessimo anche la seconda matrice lo spazio asintotico non cambierebbe *(... + (n * m))*.
Notiamo però come l'algoritmo usi sempre solo due righe della colonna per eseguire i suoi calcoli, se ci interessasse solo la lunghezza della LCS potremmo ridurre lo spazio a solo due righe rendendolo lineare.

HEAVIEST COMMON SUBSEQUENCE

Date due sequenze X e Y vogliamo determinare la sottosequenza comune più pesante
(assumendo che ad ogni carattere venga associato un peso $w(x_i)$)

\begin{cases} w{*xi*} + *C*{i-1,\space j-1} & \text{se } x{*i*} = *y*{j}
\ max{*C{i, j-1}, \space C*{i-1, j}} & \text{se } x{*i*} \neq *y*{j} \end{cases}Definiamo quindi formalmente il problema:
ISTANZA: X = <x_1, ..., x_n> Y = <y_1, ..., y_m> e i relativi pesi associati ai caratteri

SOLUZIONE: La sottosequenza comune a X e Y a cui sia associato il massimo peso

SOTTOPROBLEMI

Identifichiamo il sottoproblema generico *(i, j)* che, date X e Y, ci fornisce il massimo peso ottenibile considerando solo i primi *i* caratteri di X ed i primi *j* caratteri di Y.
Avremo quindi in totale **(n + 1)(m + 1)** sottoprobelmi.

DEFINIZIONE VARIABILI

Definiamo $\mathbf{C_{i,\,j}}$ come la soluzione al sottoproblema di taglia *(i, j)*, ovvero come il massimo peso che riusciamo ad ottenere prendendo una sottosequenza di X e Y considerando però solo i primi *i* caratteri di X ed i primi *j* caratteri di Y.

Avendo (n + 1)(m + 1) sottoproblemi totali, C sarà una matrice di dimensione **(n + 1)(m + 1)**.

CASO BASE

Ovviamente quando *i* o *j* sono pari a zero, il risultato del sottoproblema è zero, dunque:
$C_{i,\,j} = 0$ per $i = 0 \vee j = 0$

Avremo quindi in totale (n + m + 1) casi base.

CASO PASSO

Supponiamo di voler dare la risposta al sottoproblema *(i, j)* ipotizzando di aver già risolto i sottoproblemi di taglia minore; se i caratteri x_i e y_j che sto analizzando sono uguali, allora sicuramente mi conviene aggiungere il loro peso a ciò che ottenuto precedentemente, mentre se sono diversi mi "porto avanti" il meglio che ho ottenuto fino ad allora.

$$C_{i,\,j} = \begin{cases} w(x_i) + C_{i-1,\,j-1} & \text{se } x_i = y_j \\ max\{C_{i,j-1},\ C_{i-1,j}\} & \text{se } x_i \neq y_j \end{cases}$$

SOLUZIONE: la soluzione la troveremo sempre in corrispondenza di $\mathbf{C_{n,\,m}}$.

ALGORITMO

```
HCS(X, Y){
CASO BASE
   for i = 0 to n
      C[i][0] = 0
   for j = 0 to m
      C[0][j] = 0

CASO PASSO
   for i = 1 to n
      for j = 1 to m
         if X[i] == Y[j]
            C[i][j] = C[i - 1][j - 1] + w(X[i])
         else
            C[i][j] = max(C[i - 1][j], C[i][j - 1])
   return C[n][m]
}
```

TEMPO E SPAZIO
Si può subito notare come l'algoritmo richieda **T(n) = Θ(n * m)** ed **S(n) = Θ(n * m)**, proprio come la LCS.
Difatti, l'algoritmo è lo stesso, solo che invece che incentrarsi sulla lunghezza della sottosequenza ottenuta, si concentra sul peso di questa.

LONGEST INCREASING SUBSEQUENCE

Abbiamo visto come identificare una sottosequenza comune a due sequenze che sia la più lunga possibile possa essere non immediato, ma concentriamoci ora su una sola sequenza: una sua sottosequenza è un qualuque suo carattere e la più lunga è, banalmente, la sequenza stessa.
Ma non appena introduciamo ulteriori vincoli il problema può diventare complesso anche considerando una sola sequenza.

Il problema LIS consiste nel trovare una sottosequenza della sequenza X che sia la più lunga sottosequenza possibile prendendo solo ***caratteri strettamente crescenti*** *(quindi ad esempio < A, C, B, D > non va bene perché C > B)*.
Il classico approccio è provare tutte le possibili combinazioni, ma questo richiederebbe un tempo troppo elevato.

Definiamo quindi il problema come segue:
ISTANZA: $X = \langle x_1, \ldots, x_n \rangle$

SOLUZIONE: La più lunga sottosequenza di X strettamente crescente.

SOTTOPROBLEMI

Cerchiamo quindi i sottoproblemi avendo in input una sequenza $\mathbf{X = \langle x_1, \ldots, x_2 \rangle}$;
Il sottoproblema di taglia *i* è: trovare la più lunga sottosequenza strettamente crescente che termini con l'*i*-esimo carattere di X.

È importante notare come, a differenza del problema LCS, nel LIS, per dar risposta ai sottoproblemi, dobbiamo considerare tutti i risultati già calcolati; difatti se aggiungessi dei caratteri ad X potrei migliorare una LIS che magari prima non era ottimale e farla diventare la più lunga.

Ragionando allora potremo segnarci, per ogni carattere, quale sia la più lunga sottosequenza strettamente crescente che riusciamo a trovare che finisca con quello stesso carattere.
Avremo quindi che per il primo carattere il massimo è 1, in quanto il sottoproblema di taglia 1 considera solo un carattere, mentre, ipotizzando di aver già risolto i sottoproblemi più piccoli, ci basta vedere quale sia la massima lunghezza di una LIS compatibile con il carattere x_i *(e quindi che sia minore di* x_j*)* e aggiungerci 1 *(relativo al carattere* x_j*)*.

DEFINIZIONE VARIABILI

Definiamo L[i] come la soluzione al sottoproblema di taglia *i*, ovvero come la lunghezza della più lunga sottosequenza strettamente crescente che termini con l'*i*-esimo carattere di X.
Attenzione: considero solo la sottosequenza più lunga che ***termini*** con il carattere *i*-esimo, perché devo controllare quale sia il miglior risultato che posso ottenere in ogni posizione, questo perché non so quali siano i prossimi caratteri e come possano influenzare il mio ottimo, potenzialmente cambiandolo totalmente *(mentre nella LCS i risultati successivi potevano solo incrementare l'ottimo già trovato)*.

CASO BASE

i = 1
L[1] = 1 *il massimo che posso fare con una sottosequenza di X che termini col suo primo carattere è certamente 1*
Ho dunque un solo caso base.

CASO PASSO

$1 < i \leq n$
$L[i] = 1 + \max\{ L[j] \mid 0 \leq j < i \mid x_j < x_i \}$

SOLUZIONE: $\max\{ L[i] \mid 0 \leq i \leq n \}$
Questa volta la soluzione non la troveremo sempre nella stessa posizione come nella LCS, ma ci sarà data dal massimo tra le soluzione ai sottoproblemi calcolati.

ALGORITMO

```
LIS(X) {
CASO BASE
      L[1] = 1

CASO PASSO
   for i = 2 to n
      max = 0
      for j = 0 to i
         if L[j] > max AND X[j] < X[i]
            max = L[j]

      L[i] = max + 1

   return max(L)
}
```

TEMPO E SPAZIO

Il tempo dell'algoritmo è facilmente calcolabile ed è pari a $\mathbf{T(n) = \Theta(n^2)}$.
Per evitare di scansionare nuovamente l'array per trovare il massimo totale, potremmo calcolarlo mentre costruiamo l'array controllando ad ogni fine ciclo interno se L[i] sia maggiore del massimo totale trovato fino ad allora, e nel caso aggiornarlo.

Lo spazio è dato dall'array L ed è pari a **S(n) = Θ(n)**.

Per risolvere il problema originale *(fornire la sottosequenza)*, potremmo ripercorrere l'array L all'indietro controllando, ogni volta che troviamo un valore minore di quello analizzato, se il carattere nella posizione di tale valore sia compatibile con quello nella posizione che stiamo analizzando.
Si potrebbe anche salvare in un altro array S di dimensione n quale carattere ha determinato il massimo nella posizione i, a seconda di cosa abbiamo bisogno di ottimizzare, spazio o tempo.

GROWING LCS

Abbiamo fino ad ora visto i problemi della LCS e della LIS e ci chiediamo ora se sia possibile stabilire, tra due sequenze $\boldsymbol{X = \langle x_1, \ldots, x_n \rangle}$ e $\boldsymbol{Y = \langle y_1, \ldots, y_n \rangle}$ una *più lunga* sottosequenza comune che sia anche *crescente*, combinando quindi i due problemi precedenti.

Cerchiamo ora di definire il valore di un generico sottoproblema ipotizzando di aver già risolto tutti i sottoproblemi di taglia minore, ovvero i problemi di taglia *(h, k)* con:
$0 \leq h < n \quad$ e $\quad 0 \leq k < m$
Consideriamo $\mathbf{Z_{h,\,k}}$ come la soluzione del problema *(h, k)*, se la conoscessi potrei decidere se aggiungere o meno il carattere, purtroppo però non so come sia fata $\mathbf{Z_{h,\,k}}$ perché mi manca dell'informazione.

PROBLEMA AUSILIARIO

Introduciamo quindi un problema ausiliario che ci permetta di ricostruire poi la soluzione del problema originale formulato come segue:
Date due sequenze X e Y, la soluzione è data dalla lunghezza della più lunga sottosequenza comune crescente che termini con $x_n = y_m$.

ISTANZA: $X = \langle x_1, \ldots, x_n \rangle \qquad Y = \langle y_1, \ldots, y_m \rangle$
SOLUZIONE: Lunghezza della più lunga sottosequenza comune e strettamente crescente di X e Y che ***termini con x_n e y_m se questi coincidono***

SOTTOPROBLEMI

Identifichiamo ora il sottoproblema generico del nostro problema ausiliario denotandolo con *(i, j)* definito come tale:
Date X e Y, determinare la lunghezza della più lunga sottosequenza crescente che termini coi caratteri $\mathbf{x_i}$ e $\mathbf{y_j}$ se questi coincidono.

Avremo quindi in totale **(n + 1)(m + 1)** sottoproblemi.

DEFINIZIONE VARIABILI

Definiamo quindi $\mathbf{C_{i,\,j}}$ come la soluzione al sottoproblema di taglia *(i, j)*, ovvero come la lunghezza della più lunga sottosequenza comune ad X e Y che termini con i caratteri x_i e y_j se questi coincidono.
Avendo (n + 1)(m + 1) sottoproblemi,
C sarà definita come una matrice di dimensione **(n + 1)(m + 1)**.

Ora, cerchiamo di calcolare $C_{i,j}$ con i, j > 0 assumendo di aver già risolto i sottoproblemi di taglia minore, ovvero quelli identificati da *(h, k)* con $0 \leq h < i$ e $0 \leq k < j$:

Avremo dunque che ogni qual volta le x_i e y_j considerati siano diverse, la risposta al sottoproblema è 0
(poiché non esiste una sottosequenza comune che termini sia con x_i che con y_j)

Mentre quando sono uguali, le aggiungo al meglio che sono riuscito ad ottenere i sottoproblemi precedenti.

CASO BASE

$\forall$ (i, j) con $x_i \neq y_j$ $C_{i,j} = 0$
In questo caso il numero di casi base dunque dipende dall'input.

CASO PASSO

$\forall$ (i, j) con $x_i = y_j$
$C_{i,j} = 1 + \max\{ C_{h,k} \mid 0 \leq h < i \text{ e } 0 \leq k < j \mid x_h < x_i \}$ *con max(∅) = 0*
(potrei aggiungere 'AND $y_k < y_j$' alla condizione, ma sarebbe ridondante in quanto $x_i = y_j$)

SOLUZIONE PROBLEMA ORIGINALE: $\max\{C_{i,j}\}$ con $1 \leq i \leq n$ e $1 \leq j \leq m$

ALGORITMO

```
C[N][M]{
   for i = 0 to n
      for j = 0 to m
         if X[i] != Y[j]
            C[i][j] = 0          CASO BASE
         else
            max = 0         CASO PASSO
            for h = 0 to i - 1
               for k = 0 to j -1
                  if X[h] < X[i] AND C[h][k] > max
                     max = C[h][k]

            C[i][j] = 1 + max
   return Max(C)
}
```

TEMPO E SPAZIO

Si può subito notare come l'algoritmo ci impieghi **$T(n, m) = \Theta(m^2 * n^2)$**
Mentre lo spazio è pari a **$S(n, m) = \Theta(m * n)$**

Potremmo ridurre un po' il tempo *(anche se quello asintotico non cambierebbe)* calcolando il massimo totale mentre costruiamo la matrice invece che ripercorerla alla fine per trovare il massimo.

VARIANTI

Vediamo ora alcune varianti della Growing LCS, ricordando che i problemi di questo tipo si riconducono ad una GLCS con un qualche vincolo aggiuntivo.
(Con tutto uguale si intende il discorso del problema ausiliario ed il caso base della GLCS)

CARATTERI UGUALI

ISTANZA

$X = \langle x_1, x_2 \dots, x_m \rangle$ m
$Y = \langle y_1, y_2 \dots, y_n \rangle$ n

SOLUZIONE: lunghezza di LCS tra X e Y senza 2 caratteri consecutivi uguali

TUTTO UGUALE MA:

CASO RICORSIVO (AUX)

$C_{i,j} = 1 + \max\{ C_{h,k} \mid 1 \leq h \leq i, \quad 1 \leq k \leq j \mid x_h \neq x_i \}$ Se $x_i = y_j$

PARI E DISPARI

ISTANZA

$X = \langle x_1, x_2 \dots, x_m \rangle$ m
$Y = \langle y_1, y_2 \dots, y_n \rangle$ n

SOLUZIONE: lunghezza di LCS tra X e Y che alterna pari e dispari

TUTTO UGUALE MA:

CASO RICORSIVO (AUX)

$C_{i,j} = 1 + \max\{ C_{h,k} \mid 1 \leq h \leq i, \quad 1 \leq k \leq j \mid x_h \,\%\, 2 \neq x_i \,\%\, 2 \}$ Se $x_i = y_j$

ALTERNA BLU E ROSSO

(Il Denny ci tiene a specificare che tifa Genova, non ci capisco un cazzo di calcio io però)

ISTANZA

X = <x_1, x_2 ... ,x_m> m

Y = <y_1, y_2 ... ,y_n> n

SOLUZIONE: lunghezza di LCS tra X e Y che alterna blu e rossi

TUTTO UGUALE MA:

CASO RICORSIVO (AUX)

$C_{i,j} = 1 + \max\{ C_{h,k} \mid 1 \leq h \leq i, \quad 1 \leq k \leq j \mid Col(x_h) \neq Col(x_i) \}$ Se $x_i = y_j$

("Per me esiste solo un pesto, ovvero il genovese, oggi lo mangio che l'ho fatto io" cit. Denny)
(Also, il riso java lo mangia quando ha il cagotto e odia il pesto dell'esselunga)

A CONSECUTIVE

ISTANZA

X = <x_1, x_2 ... ,x_m> m

Y = <y_1, y_2 ... ,y_n> n

Caratteri

SOLUZIONE: lunghezza di LCS tra X e Y nella quale non vi sono mai lettere 'A' consecutive

TUTTO UGUALE MA:

CASO RICORSIVO (AUX)

$$C_{i,j} = \begin{cases} 1 + max\{C_{h,k} \mid 1 \leq h \leq i,\ 1 \leq k \leq j \mid x_h \neq x_i\} & \text{Se } x_i = y_j \wedge x_i = \ 'A' \\ 1 + max\{C_{h,k} \mid 1 \leq h \leq i,\ 1 \leq k \leq j\} & \text{Se } x_i = y_j \wedge x_i \neq \ 'A' \end{cases}$$

KNAPSACK

Consideriamo un insieme *X* di *n* oggetti, ad ogni oggetto associamo un peso w_i ed un valore v_i. Abbiamo anche una certa capacità C > 0 dello zaino e vogliamo trovare un sottoinsieme di X che ci permetta di non sfondare lo zaino e massimizzare il valore degli oggetti presi.

Avremo quindi che v(A) = $\sum_{i \in A} v_i$ e che w(A) = $\sum_{i \in A} w_i$ con A generico sottoinsieme di X.

Definiamo quindi formalmente il problema:
ISTANZA: $n \in \mathbb{N}$ $\forall i \in \{1, ..., n\}$ v_i, w_i C

SOLUZIONE: $S \subseteq X$ t.c. $v(S) = \max_{A \subseteq X \,:\, w(A) \leq C} \{v(A)\}$ e $w(S) \leq C$

SOTTOPROBLEMI

Ipotizziamo che un sottoproblema di knapsack sia nella forma $X_i = \{1, ..., i\}$ t.c. $w(X_i) \leq C \land v(X_i)$ sia massimo.
Considerando l'oggetto *i*:
Se $w_i > C$ $i \notin S_i$ e quindi $S_i = S_{i-1}$

Se $w_i \leq C$, i potrebbe appartenere a S_i
- se non appartiene $S_i = S_{i-1}$
- se appartiene $S_i = \{i\} \cup S_j$

Il problema è che non so dire se S_j possa essere compatibile o meno con *i*, l'unica cosa che so di S_j è che è ammissibile e ottimo considerando solo i primi *j* oggetti.

Ci manca quindi dell'informazione, introduciamo quindi un sottoproblema ausiliario definito come tale:
Dato un insieme {1, ..., i} di oggetti, con $0 \leq i \leq n$ e una capacità c t.c. $0 \leq c \leq C$, trovare un sottoinsieme tale che il peso sia $\leq$ c ed il suo valore massimo.

Avremo cosi quindi **(n + 1)(C + 1)** sottoproblemi.

DEFINIZIONE VARIABILI

Definiamo quindi $M_{i,\,c}$ come la soluzione al sottoproblema di taglia *(i, c)*, ovvero come il massimo valore che riusciamo ad ottenere considerando i primi *i* oggetti tali che il loro peso sia $\leq$ c.

Avendo (n + 1)(C + 1) sottoproblemi, M sarà definita come una matrice di dimensione (n + 1)(C + 1).

CASO BASE

Sappiamo dare la risposta immediata a tutti quei sottoproblemi tali che
$i = 0 \lor c = 0$
$M_{i,\,c} = 0$
(se considero 0 elementi o se il peso che devo rispettare è 0, ovviamente non posso prendere nulla)

CASO PASSO

$i > 0 \land c > 0$

$$M_{i,\,c} = \begin{cases} M_{i-1,\,c} & \text{se } w_i > c \\ max\{M_{i-1,\,c-w_i} + v_i,\ M_{i-1,\,c}\} & \text{se } w_i \leq c \end{cases}$$

Considerando il caso passo ho due sottocasi: se il peso dell'oggetto *i* supera la capacità c, non potrò di certo aggiungerlo alla mia soluzione ottimale, mentre se è minore o uguale a c, potrei aggiungerlo o meno, a seconda di cosa mi convenga fare; controllo quindi se ottengo un valore più grande aggiungendo l'oggetto all'insieme {1, ..., i - 1} piuttosto che non aggiungendolo.

Se deduco che *i* non mi conviene aggiungerlo, mi basta considerare l'insieme {1, ..., i - 1} come soluzione al sottoproblema *(i - 1, c)*
Se invece mi conviene aggiungere *i* all'insieme, considero il sottoproblema minore di *(i, c)* il cui insieme risulta essere compatibile con l'oggetto *i* (ovvero tale che, insieme all'oggetto *i* non raggiunga una capacità > c).

SOLUZIONE: Otterrò la soluzione del problema andando a controllare M[n][C], ovvero il massimo valore che posso ottenere considerando tutti gli *n* oggetti che abbiano peso $\leq$ C.

ALGORITMO

```
KS(){
CASO BASE
   for i = 0 to n
      M[i][0] = 0
   for c = 1 to C
      M[0][c] = 0

CASO PASSO
   for i = 1 to n
      for c = 1 to C
         if w[i] > c
            M[i][c] = M[i - 1][c]
         else
            M[i][c] = max(M[i - 1][c - w[i<]] + v[i], M[i - 1][c])
}
```

PARTITION

Abbiamo n oggetti indivisibili di valore v_i da dividere fra due persone equamente.

ISTANZA: {1, ..., n} oggetti ognuno con un valore v_i

SOLUZIONE: true SSE posso dividere gli oggetti equamente fra due persone

SOTTOPROBLEMI

Ci accorgiamo subito che se la somma M dei valori degli oggetti è dispari, la risposta sarà subito *false*, mentre se n è pari dobbiamo calcolare un sottoinsieme avente somma M/2.

Definiamo allora un sottoproblema generico come segue:
Il sottoproblema di taglia *(i, j)* sarà true SSE esiste una somma tra i valori dei primi *i* oggetti uguale a *j*

Avremo quindi (n + 1)($\frac{M}{2}$ + 1) sottoproblemi.

DEFINIZIONE VARIABILI

Definiamo una variabile $M_{i, j}$ come la soluzione al sottoproblema di taglia *(i, j)*
$M_{i, j}$ vale true SSE $\exists$ una somma di valori scelti tra i pirmi *i* oggetti che abbia valore *j*.

Avendo (n + 1)($\frac{M}{2}$ + 1) sottoproblemi, M sarà una matrice di dimensione (n)($\frac{M}{2}$)

CASO BASE

I sottoproblemi per i quali sappiamo dare una risposta immediata sono del tipo:
$M_{0, j}$ = false $\quad \forall$ j *(se prendo 0 oggetti, non otterrò mai nessun valore > 0)*

$M_{i, j}$ = true se v_i = j *(mi basta prendere l'oggetto **i**-esimo)*

CASO PASSO

$M_{i, j} = M_{i - 1, j} \vee M_{i - 1, j - vi}$

Il sottoproblema *(i, j)* sarà true se era true anche con *i - 1* oggetti o se aggiungendo il valore dell'oggetto *i* a quanto calcolato fino ad ora ottengo somma *j* (per questo ho j **-** vi)

SOLUZIONE: True SSE M[n][M/2] = true