Corso di laurea in Informatica
Progettazione d'algoritmi

Tecnica Programmazione Dinamica 3b

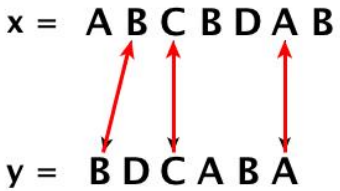
Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

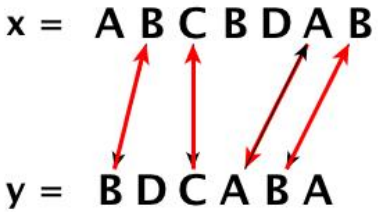
Il problema della massima sottosequenza comune. Date due sequenze di simboli X e Y, vogliamo trovare una sottosequenza comune X e Y che abbia lunghezza massima.

Ad esempio, per X = ABCBDAB e Y = BDCABA

- una delle possibili sottosequenze comuni di lunghezza 3 è BCA:



- una sottosequenza di lunghezza massima è BCAB



- altre sottosequenze di lunghezza massima sono BCBA e BDAB

Progettate un algoritmo che risolva il problema in tempo $O(n \cdot m)$ dove n sono i simboli in X ed m quelli in Y.

Concentriamoci preliminarmente sul calcolare il valore della soluzione ottima:

Sia $X = x_1, x_2, \ldots x_n$ e $Y = y_1, y_2, \ldots y_m$.

Definiamo una tabella bidimensionale T di dimensioni $(n+1) \times (m+1)$ dove:

- $T[i][j]$= massima lunghezza possibile per sottosequenze comuni al prefisso di X di lunghezza i ed al prefisso di Y di lunghezza j.

$T[i][j]$ **con** $i, j \geq 1$ conterrà dunque la lunghezza massima per sottosequenze comuni alle due stringhe

$X' = x_1 x_2 \ldots x_i$

$Y' = y_1 y_2 \ldots y_j$

La soluzione al nostro problema sarà nella cella $T[n][m]$

resta ora da definire la formula ricorsiva che permette di calcolare la cella $T[i,j]$ a partire dalle celle precedenti già calcolate:

$$T[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ T[i-1][j-1] + 1 & \text{se } X[i-1] = Y[j-1] \\ \max\{T[i-1][j],\ \ T[i][j-1]\} & \text{altrimenti} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- se non possiamo usare i simboli di una delle due sequenze è ovvio che la sottosequenza comune avrà lunghezza 0
- se le due sequenze terminano con lo stesso simbolo allora la sottosequenza comune più lunga terminerà con quel simbolo e per il resto sarà la sequenza più lunga tra $x_1, x_2 \ldots x_{i-1}$ e $y_1, y_2 \ldots y_{i-1}$, vale a dire $T[i,j] = T[i-1,j-1] + 1$.
- se le due sequenze terminano con simboli diversi, diciamo a per X e b per Y, allora deve verificarsi uno dei seguenti tre casi:
  - la sottosequenza più lunga termina con a. In questo caso l'ultimo simbolo di Y non influisce e si ha: $T[i][j] = T[i][j-1]$
  - la sottosequenza più lunga termina con b. In questo caso l'ultimo simbolo di X non influisce e si ha: $T[i][j] = T[i-1][j]$
  - la sottosequenza più lunga termina con un simbolo c diverso da a e b. In questo caso l'ultimo simbolo di X e l'ultimo simbolo di Y non influiscono e si ha: $T[i][j] = T[i-1][j-1]$ e dovendo prendere il caso che produce la sottosequenza più lunga deve aversi: $T[i][j] = \max(\ T[i][j-1], T[i-1][j], T[i-1][j-1]\ ) = \max(\ T[i][j-1], T[i-1][j\ )$ dove l'ultimo passaggio segue perché $T[i-1][j-1] \leq T[i][j-1]$ e $T[i-1][j-1] \leq T[i-1][j]$.

$$T[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ T[i-1][j-1] + 1 & \text{se } X[i-1] = Y[j-1] \\ \max\{T[i-1][j],\ \ T[i][j-1]\} & \text{altrimenti} \end{cases}$$

Implementazione:

Complessità $\Theta(n \cdot m)$

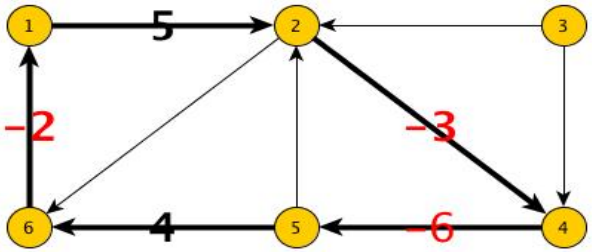
```
def es(X, Y):
    n, m = len(X), len(Y)
    T = [ [0]*(m+1) for _ in range(n+1) ]
    for i in range(n+1):
        for j in range(m+1):
            if i == 0 or j == 0:
                T[i][j] = 0
            elif X[i-1] == Y[j-1]:
                T[i][j] = T[i-1][j-1] + 1
            else:
                T[i][j] = max( T[i-1][j], T[i][j-1] )
    return T[n][m]


>>> X = 'ABCBDABO'
>>> Y = 'BDCABAO'
>>> es(X,Y)
5 # grazie alla sottosequenza 'BCBAO'
```

Algoritmi per la ricerca di cammini su grafi con pesi anche negativi

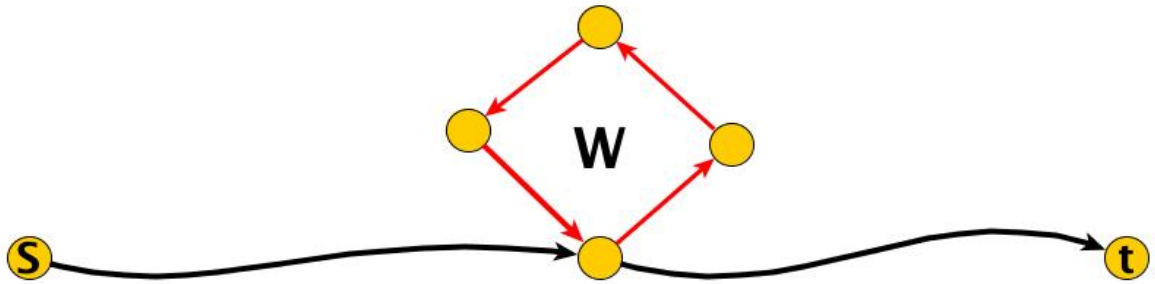
Problema: dato un grafo diretto e pesato G con archi i cui pesi possono essere anche negativi e fissato un suo nodo s, vogliamo calcolare il costo minimo dei cammini che portano da s agli altri nodi del grafo (il costo è $+\infty$ se non c'è cammino)

Definizione: Un ciclo negativo è un ciclo diretto formato da archi con pesi la cui somma è negativa.



Il ciclo evidenziato in figura è negativo di costo $5-3-6+4-2=$ - 2

Se in un cammino tra i nodi s e t è presente un nodo che appartiene ad un ciclo negativo allora tra s e t non esiste il cammino di costo minimo.



- se per il ciclo W si ha $costo(W) < 0$, ripassando più volte attraverso il ciclo W possiamo abbassare arbitrariamente il costo del cammino da s a t.

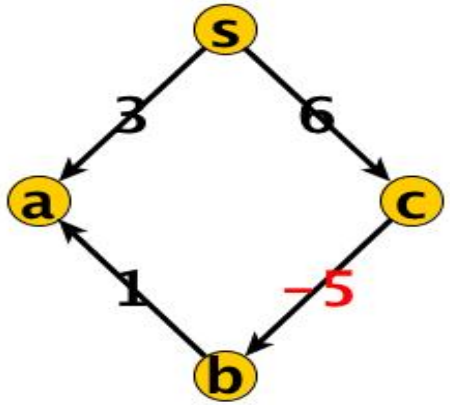
Alla luce di quanto appena visto il problema non era ben formulato. Ecco di seguito la giusta formulazione:

Problema: dato un grafo diretto e pesato G con archi i cui pesi possono essere anche negativi **ma che non contiene cicli negativi** e fissato un suo nodo s, vogliamo calcolare il costo minimo dei cammini che portano da s agli altri nodi del grafo (il costo è $+\infty$ se non c'è cammino).

E' un problema che abbiamo già affrontato nel caso in cui i pesi del grafo G sono tutti non negativi (algoritmo di Dijkstra).

Con la presenza di archi di peso negativo l'algoritmo di Dijkstra può produrre errori.

Considera ad esempio il seguente grafo:



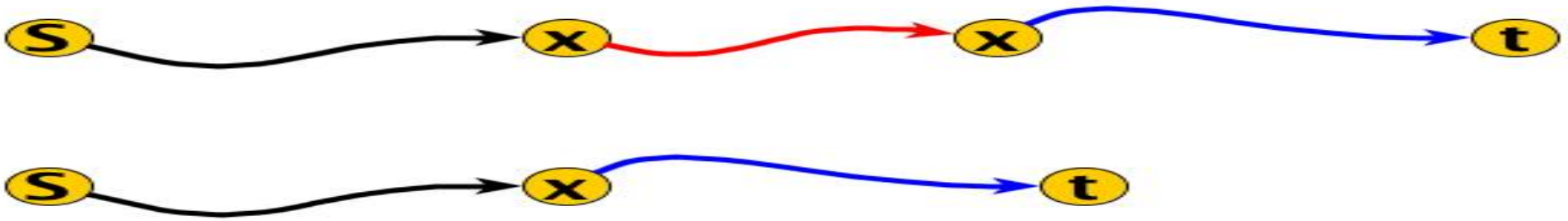
L'algoritmo di Dijkstra sceglie come prima cosa il cammino composto dal solo arco **(s,a)** di costo **3**, ma il cammino da **s** ad **a** che passa per **c** e **b** costa 6 - 5 +1= **2**.

Vedremo ora un algoritmo basato sulla tecnica della programmazione dinamica che risolve il problema in tempo $O(n^2 + m \cdot n)$.
L'algoritmo è noto in letteratura come **algoritmo di Bellman-Ford.**

Proprietà: se G non ha cicli negativi allora per ogni nodo t raggiungibile dalla sorgente s esiste un cammino di costo minimo che ha lunghezza al più $n-1$

Prova:

- consideriamo il più corto cammino minimo e assumiamo per assurdo che abbia lunghezza superiore ad $n-1$.
- c'è allora un nodo x toccato più volte dal cammino. Di conseguenza il cammino contiene un ciclo.
- possiamo eliminare dal cammino il ciclo ed ottenere un cammino di lunghezza inferiore che è ancora minimo (ricorda che il ciclo eliminato non può avere costo negativo).



Per quanto appena visto: nella ricerca dei cammini minimi possiamo restringerci a cammini di lunghezza al più $n-1$. Tutto questo suggerisce di considerare i sottoproblemi che si ottengono limitando la lunghezza dei cammini.

Definiamo così la seguente tabella di dimensione $n \times n$:

$$T[k][j] = \begin{cases} \text{costo di un cammino minimo da } s \text{ a } j \text{ di lunghezza al più } k & \text{se esiste} \\ +\infty & \text{altrimenti} \end{cases}$$

La soluzione al nostro problema sarà data dagli n valori che troviamo nell'ultima riga della tabella:

$$T[n-1][0], \quad T[n-1][1], \quad T[n-1][2], \ldots\ldots \quad T[n-1][n-1]$$

Infatti il costo minimo per andare da s al generico nodo t sarà $T[n-1][t]$.

Resta da definire la formula ricorsiva che permette di calcolare i valori delle celle $T[k][j]$ in funzione di celle già calcolate:

$$T[k][j] = \begin{cases} 0 & \text{se } j = s \\ +\infty & \text{se } k = 0 \\ \min_{<x,\ j>\in E} \Big(T[k-1][x] + costo(x,j) \Big) & \text{altrimenti} \end{cases}$$

Per giustificare il caso $k > 0$ e $v \neq s$ si può ragionare in questo modo:

- se al contrario il cammino ha costo al più k al nodo $j \neq s$ devo arrivare provenendo da uno dei suoi vicini x attraversando l'arco $< x,\ v >$ e a x devo essere arrivato con un cammino minimo che ha attraversato al più $k-1$ archi per cui: $T[k][v] = T[k-1][x] + costo(x,v)$. Tra tutti i nodi x per cui questo è possibile devo prendere quello di costo minimo, e allora $T[k][v] = \min_{(x,\ j)\in E}(T[k-1][x] + costo(x,j))$

$$T[k][j] = \begin{cases} 0 & \text{se } j = s \\ +\infty & \text{se } k = 0 \\ \min_{<x,\ j>\in E} \Big(T[k-1][x] + costo(x,j) \Big) & \text{altrimenti} \end{cases}$$

Implementazione:

```
def CostoCammini(G, s):
    from math import inf
    n = len(G)
    T =[ [inf]*n for _ in range(n) ]
    T[0][s] = 0
    GT = Trasposto(G)
    for k in range(1,n):
        for j in range(n):
            if i == s:
                T[k][j]=0
            else:
                for x,costo in GT[j]:
                    T[k][j] = min( T[k][j], T[k-1][x] + costo )
    return T[n-1]
```

```
def Trasposto(G):
    n=len(G)
    GT = [ [] for _ in G ]
    for i in range(n):
        for j,costo in G[i]:
            GT[j].append( (i,costo) )
    return GT
```

```
G=[
   [(1,3),(3,6)],
   [],
   [(1,1)],
   [(2,-5)],
   [(0,-2)]
  ]

>>> costo_cammini(G,4)
[-2, 0, -1, 4, 0]
>>> costo_cammini(G,0)
[0, 2, 1, 6, inf]
```

- l'inizializzazione della tabella T richiede tempo $\Theta(n^2)$
- la costruzione del grafo trasposto GT richiede tempo $O(n+m)$. Nota che grazie a questo pre-processing avremo un accesso efficiente ai nodi che hanno un arco diretto in j.
- per i tre for annidati (for k in $range(1,n)$ ... for j in $range(n)$ ... for $< x,\ costo >$ in $GT[j]$) è ovvio il limite superiore $O(n^3)$. Tuttavia un'analisi più attenta permette di dare un limite più stretto:
  - I due *for* più interni (for j in $range(n)$ ... for $< x,\ costo >$ in $GT[j]$) hanno costo totale $\Theta(m)$. Sostanzialmente il tempo richiesto è quello di scorrere tutte le liste di adiacenza del grafo GT che hanno lunghezza totale m.

Dal costo ai cammini:

Per ritrovare anche i cammini, oltre che il loro costo, con la tabella T bisogna calcolare anche l'albero P dei cammini minimi. questo si può fare facilmente mantenendo per ogni nodo j il suo predecessore cioè il nodo u che precede j nel cammino. Il valore di $P[j]$ andrà aggiornato ogni volta che il valore di $T[k][j]$ cambia (ovvero diminuisce) in quanto abbiamo trovato un cammino migliore.

```
def CostoCammini1(G, s):
    from math import inf
    n = len(G)
    T =[ [inf]*n for _ in range(n) ]
    P = [-1]* n
    GT = Trasposto(G)
    P[s]=s
    T[0][s] = 0
    for k in range(1,n):
        for j in range(n):
            if j == s :
                T[k][j] = 0
            else:
                for x, costo in GT[j]:
                    if T[k-1][x] + costo <T[k][j]:
                        T[k][j] =  T[k-1][x] + costo
                        P[i] = x
    return T[n-1], P
```

Con questa implementazione al termine dell'algoritmo

- $T[n-1][j] \neq +\infty$ indica che j è raggiungibile a partire da s, in questo caso $P[j]$ conterrà il nodo che precede j nel cammino minimo da s a j
- $T[n-1][j] = +\infty$ indica che j non è raggiungibile a partire da s, in questo caso $P[j]$ conterrà il valore -1.

Ottimizzazioni:

- il contenuto di una cella della riga k dipende dal contenuto delle celle alla riga $k-1$, quindi:
  - Se la riga k della tabella T è identica alla riga $k-1$ anche le righe seguenti non varieranno, tanto vale allora terminare l'algoritmo senza aver calcolato le righe restanti della tabella. Questo accorgimento non modifica la complessità asintotica dell'algoritmo ma in pratica può contare molto.
  - Non serve memorizzare l'intera tabella T bastano le ultime due righe. Perciò l'algoritmo può essere facilmente modificato in modo da utilizzare memoria $O(n)$ anziché $O(n^2)$

L'algoritmo appena descritto è conosciuto con il nome di Algoritmo di Bellman-Ford.

A priori non sappiamo se il grafo su cui applichiamo l'algoritmo ha cicli negativi (e quindi se i cammini restituiti sono effettivamente i cammini minimi o semplicemente i cammini minimi con lunghezza al più $n-1$).

Una piccola modifica alla versione dell'algoritmo di Bellman-Ford appena descritto permette di scoprire se il grafo contiene cicli negativi raggiungibili da s o meno:

- calcola una riga in più della tabella (vale a dire la riga n) con il costo dei cammini minimi di lunghezza al più n.
- Le righe n ed $n-1$ della tabella risultano identiche se e solo se nel grafo non ci sono cicli negativi raggiungibili da s.

Implementare questo test ovviamente non cambia l'asintotica dell'algoritmo.

ESERCIZIO.

Si ha una sequenza di n carte, ciascuna carta ha come valore un numero intero. Due giocatori a turno prendono una delle due carte agli estremi della sequenza. Al termine del gioco il punteggio di ciascun giocatore è dato dalla somma dei valori delle carte da lui prese. Lo scopo del gioco è ottenere il punteggio superiore a quello dell'avversario.

Data la sequenza dei valori delle n carte tramite la lista A (dove $A[i]$ è il valore dell'i-esima carta della sequenza), progettare un algoritmo che in tempo $O(n^2)$ determini qual'è la cifra massima che posso sempre vincere indipendentemente dalle mosse del mio avversario.

Ad esempio per $A = [10, 100, 2, 1]$ la risposta è 101 (basta prendere ala prima mossa la carta 1 e poi al secondo turno la carta 100.

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

Uso una tabella bidimensionale $n \times n$ dove:
$T[i][j]$ = il valore massimo che posso guadagnare (indipendentemente dalle mosse del mio avversario) se si gioca sul sottovettore che inizia in A[i] e termina in A[j], $i \leq j$.
La soluzione al problema sarà il valore $T[0][n-1]$.

resta ora da definire la formula ricorsiva che permette di calcolare la cella $T[i,j]$ a partire dalle celle precedenti già calcolate:

$$T[i][j] = \begin{cases} A[i][j] & \text{se } i = j \\ \max\Big(A[i] + (S[i+1][j] - T[i+1][j]),\quad A[j] + (S[i][j-1] - T[i][j-1])\Big) & \text{altrimenti} \end{cases}$$

Dove con $S[i]][j]$ denoto la somma degli elementi della sequenza che vanno da $A[i]$ ad $A[j]$.

Nota che $S[i][j] - T[i][j]$ rappresenta quanto si può al più guadagnare giocando sul sottovettore che inizia in $A[i]$ e termina in $A[j]$ quando tocca all'avversario muovere per primo.

La ricorrenza viene fuori dal seguente ragionamento:

- Se i=j il gioco si svolge su una sequenza di un solo elemento e dovendo essere io a muovere per primo si ha T[i][j]=A[i]

- Se $i < j$ posso scegliere una delle due carte agli estremi:
  - Se scelgo A[i] seguirà una partita sul vettore che va da $A[i+1]$ a $A[j]$ con il mio avversario che muove per primo. Guadagnerò quindi di certo A[i]+ (S[i+1][j]-T[i+1][j]) .
  - Se scelgo A[j] seguirà una partita sul vettore che va da $A[i]$ a $A[j-1]$ con il mio avversario che muove per primo. Guadagnerò quindi di certo A[j]+ (S[i][j-1]—T[i][j-1]).

  Posso dunque guadagnare $\max\Big(A[i] + (S[i+1][j] - T[i+1][j]),\quad A[j] + (S[i][j-1] - T[i][j-1])\Big)$

Nota che la tabella è definita solo per $i \leq j$ eper poter calcolare $T[i][j]$ devo disporre dei valori $T[i+1][j]$ e $T[i][j-1]$ devo quindi riempire la tabella per diagonali a partire dalla diagonale principale.

Implementazione

Complessità $\Theta(n^2)$

```python
def CalcolaS(A):
    n = len(A)
    S = [[0]*n for _ in range(n)]
    for i in range(n):
        for j in range(i,n):
            if i == j:
                S[i][j] = A[i]
            else:
                S[i][j] = S[i][j-1] + A[j]
    return S
```

```python
def es(A):
    n=len(A)
    T=[[0]*n for _ in range(n)]
    S= CalcolaS(A)
    for i in range(n):
        T[i][i]= A[i]
    for l in range(1,n):
        for i in range(n-l):
            T[i][i+l]=max(A[i]+(S[i+1][i+l]-T[i+1][i+l]),A[i+l] + (S[i][i+l-1]-T[i][i+l-1]))
    return T[0][n-1], T

>>> A = [1, 2, 3]
>>> es(A)
4 #per la tabella si ha: T= [[1, 2, 4], [0, 2, 3], [0, 0, 3]]

>>> B = [10, 100, 2, 1]
101 # per la tabella si ha: T= [10, 100, 12, 101], [0, 100, 100, 101], [0, 0, 2, 2], [0, 0, 0, 1]]
```

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO.

Ho una matrice $3 \times n$ e voglio sapere quanti modi ho di riempirla coi tre simboli a, b, e c in modo che nella matrice non compaiano mai due simboli uguali adiacenti sulla stessa riga o sulla stessa colonna.

Ad esempio:

- per n=1 la risposta è 12 (se uso tutti e 3 i simboli ho 6 possibili modi di riempire la matrice (*abc*, *acb*, *bac*, *bca*, *cab*, *cba*) se uso solo due simboli ho altri 6 modi di colorare la matrice (*aba*, *aca*, *bab*, *bcb*, *cac*, *cbc*), con un solo simbolo non è possibile colorare senza adiacenti uguali

Progettare un algoritmo che dato n restituisce il numero di modi che ho di riempire la matrice risolve il problema in tempo $O(n)$

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

```
per n =  1 la risposta è  12
per n =  2 la risposta è  54
per n =  3 la risposta è  246
per n =  4 la risposta è  1122
per n =  5 la risposta è  5118
per n =  6 la risposta è  23346
per n =  7 la risposta è  106494
per n =  8 la risposta è  485778
```