31/05/21

Programmazione dinamica (IV parte)

Progettazione di Algoritmi a.a. 2020-21

Matricole congrue a 1

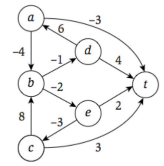
Docente: Annalisa De Bonis

Algoritmo di Bellman-Ford per i cammini minimi

Osservazione

- L'algoritmo di Bellman-Ford di fatto calcola le lunghezze dei cammini minimi da v a t per ogni v (risolve Single Destination Shortest Paths)
- Queste lunghezze sono contenute nella riga n-1
- L'algoritmo puo` essere scritto in modo che prenda in input un vertice sorgente s ed un vertice destinazione t ma il contenuto della tabella M dipende solo da t.
- In altri termini, una volta costruita la tabella per un certo t, possiamo ottenere la lunghezza del percorso piu` corto da un qualsiasi nodo v al nodo t andando a leggere l'entrata M[n-1,v]

Bellman-Ford: esempio



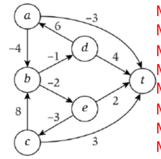
M

	t	a	b	c	d	e
0	0	∞	∞	∞	∞	∞
1	0	-3	∞	3	4	2
2	0	-3	0	3	3	0
3	0	-4	-2	3	3	0
4	0	-6	-2	3	2	0
5	0	-6	-2	3	0	0

S

	t	a	b	c	d	e
0	t	Ø	Ø	Ø	Ø	Ø
1	t	t	Ø	t	t	t
2	t	t	e	t	a	c
3	t	b	e	t	a	c
4	t	b	e	t	a	c
5	t	b	e	t	a	c

Bellman-Ford: esempio



$M[5,a]=\min\{c_{ab}+M[4,b] , c_{at}+M[4,t]\}=\min\{-4-2, -3+0\}=-6$

$M[4,b]=\min\{c_{bd}+M[3,d],c_{be}+M[3,e]\}=\min\{-1+4,-2+0\}=-2$

$M[4,t]=0$

$M[3,d]=\min\{c_{dt}+M[2,t],c_{da}+M[2,a]\}=\{4+0, 6+3\}=4$

$M[3,e]=\min\{c_{ec}+ M[2,c], c_{et}+ M[2,t]\}=\min\{-3+3, 2+0\}= 0$

$M[2,t]=0$

$M[2,c]=\{c_{cb}+ M[1,b], c_{ct}+ M[1,t]\}=\min \{8+\infty, 3+0\}=3$

$M[2,a]=\{c_{ab}+ M[1,b], c_{at}+ M[1,a]\}=\min \{8+\infty, 3+0\}=3$

$M[1,t]=0$

$M[1,b]=\min\{c_{bd}+ M[0,d], c_{be}+ M[0,e]\}=\{-1+\infty ,-2+\infty\}=\infty$

$M[1,a]=\min\{c_{ab}+ M[0,b], c_{at}+ M[0,t]\}=\{-4+\infty,-3+0\}=-3$

$M[0,t]=0$

$M[0,d]=M[0,b]=M[0,e]=\infty$

computati dal basso verso l'alto

	t	a	b	c	d	e
0	0	∞	∞	∞	∞	∞
1	0	-3	∞	3	4	2
2	0	-3	0	3	3	0
3	0	-4	-2	3	3	0
4	0	-6	-2	3	2	0
5	0	-6	-2	3	0	0

Algoritmo che produce il cammino minimo

```
FindPath(i,v):
 if S[i,v]= Ø
    output "No path"
     return
 if v= t
    output t
    return
 output v
 FindPath(i-1,S[i,v])
```

prima volta invocato con i=n-1 e v uguale al nodo per il quale vogliamo computare il cammino minimo fino a t

tempo O(n) perche'
- se ignoriamo il tempo per la chiamata ricorsiva al suo interno, il tempo di ciascuna chiamata e` O(1)
- vengono effettuate al piu` n-1 chiamate

63

Bellman-Ford: esempio

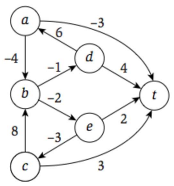
S

	t	a	b	c	d	e
0	t	Ø	Ø	Ø	Ø	Ø
1	t	t	Ø	t	t	t
2	t	t	e	t	a	c
3	t	b	e	t	a	c
4	t	b	e	t	a	c
5	t	b	e	t	a	c

Supponiamo di voler conoscere il percorso minimo tra a e t
Invoco FindPath(5,a)

output a e effettua ricorsione con i=4 e v=S[5,a]=b
output b e effettua ricorsione con i=3 e v= S[4,b]=e
output e e effettua ricorsione con i=2 e v= S[2,e]=c
output c e effettua ricorsione con i=1 e v= S[1,c]=t
output t ed esci

Il percorso minimo da a verso t e` a,b,e,c,t




```
FindPath(i,v):
 if S[i,v]= Ø
    output "No path"
     return
 if v= t
    output t
    return
 output v
 FindPath(i-1,S[i,v])
```


Miglioramento dell'algoritmo

- Usiamo un array unidimensionale M:
  - M[v] = percorso da v a t piu` corto che abbiamo trovato fino a questo momento
  - Per ogni i=1,...,n-1 computiamo cosi` M[v] per ogni v:

$$M[v] = \min(M[v], \min_{(v,w)\in E}(c_{vw} + M[w]))$$

1. computiamo per ogni arco (v,w) uscente da v la distanza $c_{v,w}$ +M[w] che rappresenta la lunghezza del percorso piu` corto computato fino a quel momento per andare da v a t passando per l'arco (v,w)
2. tra tutte le lunghezze computate in 1, prendiamo quella piu` piccola e se questa e` minore di M[v] aggiorniamo M[v]

Miglioramento dell'algoritmo

1. computiamo per ogni arco (v,w) uscente da v la distanza $c_{v,w}$ +M[w].
2. tra tutte le lunghezze computate in 1. prendiamo quella piu` piccola e se questa e` minore di M[v] aggiorniamo M[v]

- l'algoritmo che vedremo calcola le distanze al punto 1 considerando solo quegli archi (v,w) per cui si ha che M[w] ha cambiato valore all'iterazione precedente
- di fatto l'algoritmo scandisce tutti i nodi w del grafo e ogni volta che ne incontra uno il cui valore M[w] e` cambiato all'iterazione precedente va ad esaminare tutti gli archi (v,w) entranti in w. Per ciascuno di questi archi calcola la distanza $c_{v,w}$ +M[w] e se questa e` minore di M[v], pone M[v]= $c_{v,w}$ +M[w]
  - si noti che alla fine l'algoritmo avra` esaminato per ogni nodo v tutti gli archi (v,w) per cui si ha che M[w] ha cambiato valore all'iterazione precedente

31/05/21

Computazione del cammino minimo

- Per ogni vertice v memorizziamo in S[v] il successore di v, cioe` il primo nodo che segue v lungo il percorso da v a t di costo M[v].
- S[v] viene aggiornato ogni volta che M[v] viene aggiornato. Se M[v] viene posto uguale a c_{vw}+M[w] allora si pone S[v]=w.

67

Implementazione efficiente di Bellman-Ford

```
Push-Based-Shortest-Path(G, s, t) {
   foreach node v ∈ V {
      M[v] ← ∞
      S[v] ← ϕ   //nel libro si chiama first[v]
   }

   M[t] = 0 , S[t]=t
   for i = 1 to n-1 {
      foreach node w ∈ V {
      if (M[w] has been updated in previous iteration) {
         foreach node v such that (v, w) ∈ E {
            if (M[v] > M[w] + c_vw) {
               M[v] ← M[w] + c_vw
               S[v] ← w
            }
         }
      }
      if no M[v] value changed in this iteration i
        return M[s]
   }
   return M[s]
}
```

NB: in una certa iterazione del for esterno quando si calcola una distanza M[w]+c_{vw} potrebbe accadere che M[w] sia stata aggiornata gia` in quella stessa iterazione.

68

Implementazione efficiente di Bellman-Ford
Alcune osservazioni sull'algoritmo

- Il risparmio in termini di spazio si basa sul fatto che non e` necessario portarsi dietro tutta la matrice M perche' nell'algoritmo di fatto ogni volta che si riempe una nuova riga di M si fa uso solo dei valori della riga precedente
  - per riempire la riga i si usano solo i valori presenti della riga i-1
  - quindi perche' portarsi dietro anche le altre righe?

•Un primo immediato miglioramento lo si ottiene andando a modificare la prima versione dell'algoritmo in modo che
1. usi un array unidimensionale M
2. ad ogni iterazione del for piu` esterno vada ad aggiornare ciascun valore M[v] allo stesso modo in cui prima computava i valori M[i,v].
   - Per far questo invece di utilizzare i valori M[i-1,v] utilizzera` i valori M[v] computati all'iterazione precedente che saranno stati salvati in un array di appoggio

Con questa modifica usiamo 2 array unidimensionali per computare le lunghezze dei percorsi e un array S per tenere traccia dei successori-> spazio O(n)

69

Algoritmo di Bellman-Ford : I miglioramento

MA: array di appoggio

```
Improved-Shortest-Path_1(G, t) {
   foreach node v ∈ V
      M[v] ← ∞
      MA[v] ← ∞
      S[v] ← Ø // Ø  indica che non ci sono percorsi
                //da v a t di al piu` 0 archi
   M[t] ← 0
   MA[t]← 0
   S[t] ← t  //t indica che non ci sono successori
                 //lungo il percorso ottimo da t a t
   for i = 1 to n-1
      foreach node v ∈ V
         foreach edge (v, w) ∈ E
            if MA[w] + c_vw < M[v]
               M[v] ← MA[w] + c_vw
               S[v]← w  //serve per ricostruire i
                        //percorsi minimi verso t

         MA[v]=M[v] //salvo M[v] nell'array di appoggio
}
```


70

Implementazione efficiente di Bellman-Ford
Alcune osservazioni sull'algoritmo

- L'algoritmo Push-Based-Shortest-Path si basa oltre che sull'osservazione fatta nella slide precedente anche sulla seguente osservazione:
- Se in una certa iterazione i del for esterno il valore di MA[w] è lo stesso dell'iterazione precedente (M[w] non è stato aggiornato nel corso dell'iterazione i-1) allora i valori **MA[w] + c_{vw}** computati nell'iterazione i sono esattamente gli stessi computati nell'iterazione i-1.
- Questa osservazione dà l'idea per un secondo miglioramento dell'algoritmo: quando in una certa iterazione i del for esterno, l'algoritmo calcola M[v] va a considerare solo quei nodi w per cui esiste l'arco (v,w) e tali che M[w] e` stato modificato durante l'iterazione i-1.
- L'algoritmo nella slide successiva realizza questa idea in questo modo: scandisce ciascun nodo w del grafo e controlla se il valore di M[w] è cambiato nell'iterazione precedente e solo in questo caso esamina gli archi (v,w) entranti in v e per ciascuno di questi archi computa **MA[w] + c_{vw}**
  - Cio` equivale a scandire tutti i nodi v e a controllare per ogni arco (v,w) uscente da v se M[w] e` cambiato nell'iterazione precedente prima di calcolare **MA[w] + c_{vw}**

Algoritmo di Bellman-Ford : II miglioramento

MA: array di appoggio

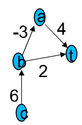
```
Improved-Shortest-Path_2(G, t) {
   foreach node v ∈ V
      M[v] ← ∞
      MA[v] ← ∞
      S[v] ← Ø // Ø  indica che non ci sono percorsi
                //da v a t di al piu` 0 archi
   M[t] ← 0
   MA[t] ← 0
   S[t] ← t  //t indica che non ci sono successori
               //lungo il percorso ottimo da t a t
   for i = 1 to n-1
     foreach node w ∈ V
        if M[w] has been updated in iteration i-1
         foreach edge (v, w) ∈ E
             if MA[w] + cvw < M[v]
                M[v] ← MA[w] + cvw
                S[v] ← w  //serve per ricostruire i
                           //percorsi minimi verso t
    foreach node v ∈ V
      MA[v]=M[v]//salvo M[v]nell'array di appoggio
}
```


Implementazione efficiente di Bellman-Ford
Alcune osservazioni sull'algoritmo

- Torniamo per un momento al fatto che un miglioramento dell'algoritmo consiste nell'usare un array unidimensionale M.
- Abbiamo detto che per far ciò l'algoritmo può usare un array di appoggio che memorizza i valori di M computati dall'iterazione precedente del for esterno.
- Domanda: cosa accade se non utilizziamo un array di appoggio?
- Consideriamo l'iterazione i del for esterno.
- Se non utilizziamo un array di appoggio, quando calcoliamo **M[w] + c_{vw}** , siamo costretti ad usare i valori M[w] presenti in M.
  - Quando calcoliamo **M[w] + c_{vw}**, il valore **M[w]** potrebbe essere uguale al valore computato nell'iterazione i-1 o potrebbe gia` essere stato aggiornato nell'iterazione i (anche piu` di una volta) .
  - Nel caso M[w] sia stato già modificato nell'iterazione i allora M[w] conterrà la lunghezza di un percorso piu` corto rispetto al valore di M[w] computato nell'iterazione precedente.
    - Di conseguenza M[v] potrebbe essere aggiornato con un valore piu` piccolo di quello che si sarebbe ottenuto utilizzando il valore di M[w] computato nell'iterazione precedente.

Implementazione efficiente di Bellman-Ford
Alcune osservazioni sull'algoritmo

- Conseguenze dell'osservazione nella slide precedente:
  - Dopo ogni iterazione i, M[v] potrebbe contenere la lunghezza di un percorso per andare da v a t formato da **piu` di i** archi.
  - La lunghezza di M[v] e` sicuramente non piu` grande della lunghezza del percorso piu` corto per andare da v a t formato da **al massimo** i archi.



  - Esempio, Consideriamo il grafo qui di fianco.
  - Iterazione i=1: supponiamo di esaminare i nodi w in questo ordine t,a,b,c. Quando esaminiamo w=t, poniamo M[a]=4 e M[b]=2. Quando si esamina w=a si ha M[a]=4 e di conseguenza M[b] da 2 che ora diventa 1 (lunghezza del percorso b,a,t). Quando poi esaminiamo b, M[c] da ∞ che era diventa 7 (lunghezza di c,b,a,t).
  - Nell'implementazione con array di appoggio, alla fine della prima iterazione avremmo avuto M[b]=2 e M[c] = ∞.

- Il terzo e ultimo miglioramento consiste nel modificare `Improved-Shortest-Path_2` in modo che non usi l'array di appoggio. In questo modo si ottiene l'algoritmo `Push-Based-Shortest-Path`.

Miglioramento dell'algoritmo

Teorema. Durante l'algoritmo **Push-Based-Shortest-Path**, M[v] e` la lunghezza di un certo percorso da v a t, e dopo i round di aggiornamenti (dopo i iterazioni del for esterno) il valore di M[v] **non è più grande della lunghezza del percorso minimo da v a t che usa al piu` i archi**

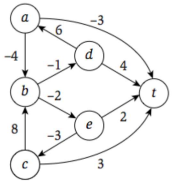
- Non usare un array di appoggio in pratica accelera i tempi per ottenere i percorsi piu` corti fino a t formati da al piu` n-1 archi (che sono quelli che ci interessa ottenere).
- **Nulla cambia per quanto riguarda l'analisi asintotica dell'algoritmo**

- Conseguenze sullo spazio usato da **Push-Based-Shortest-Path**
- Memoria: O(n).

- Tempo:
- il tempo e` sempre O(nm) nel caso pessimo pero` in pratica l'algoritmo si comporta meglio.
  - Possiamo interrompere le iterazioni non appena accade che durante una certa iterazione i nessun valore M[v] cambia

75

Miglioramento dell'algoritmo: un esempio



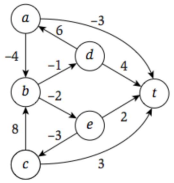
		t	a	b	c	d	e
inizializzazione	M	0	∞	∞	∞	∞	∞
	S	t	ϕ	ϕ	ϕ	ϕ	ϕ
i=1 w=t	M	0	-3	∞	3	4	2
	S	t	t	ϕ	t	t	t
i=1 w=a	M	0	-3	∞	3	3	2
	S	t	t	ϕ	t	a	t
i=1 w=b	M	0	-3	∞	3	3	2
	S	t	t	ϕ	t	a	t
i=1 w=c	M	0	-3	∞	3	3	0
	S	t	t	ϕ	t	a	c
i=1 w=d	M	0	-3	2	3	3	0
	S	t	t	d	t	a	c
i=1 w=e	M	0	-3	-2	3	3	0
	S	t	t	e	t	a	c

le celle arancioni sono quelle il cui valore non e` cambiato nell'i-esima iterazione

76

31/05/21

Miglioramento dell'algoritmo: un esempio

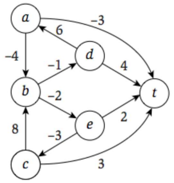


		t	a	b	c	d	e
fine iteraz. i =1	M	0	-3	-2	3	3	0
	S	t	t	e	t	a	c
i=2 w=a	M	0	-3	-2	3	3	0
	S	t	t	e	t	a	c
i=2 w=b	M	0	-6	-2	3	3	0
	S	t	b	e	t	a	c
i=2 w=c	M	0	-6	-2	3	3	0
	S	t	b	e	t	a	c
i=2 w=d	M	0	-6	-2	3	3	0
	S	t	b	e	t	a	c
i=2 w=e	M	0	-6	-2	3	3	0
	S	t	b	e	t	a	c

le celle arancioni sono quelle il cui valore non e` cambiato nell'i-esima iterazione

77

Miglioramento dell'algoritmo: un esempio

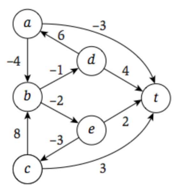


		t	a	b	c	d	e
fine iteraz. i =2	M	0	-6	-2	3	3	0
	S	t	b	e	t	a	c
i=3 w=a	M	0	-6	-2	3	0	0
	S	t	b	e	t	a	c

le celle arancioni sono quelle il cui valore non e` cambiato nell'i-esima iterazione

78

Miglioramento dell'algoritmo: un esempio



	t	a	b	c	d	e	
M	0	-6	-2	3	0	0	fine iteraz. i=3
S	t	b	e	t	a	c	
M	0	-6	-2	3	0	0	i=4 w=d
S	t	b	e	t	a	c	

le celle arancioni sono quelle il cui valore non e` cambiato nell'i-esima iterazione

Minimum Coin Change Problem

- Dato un insieme infinito di monete con valori $v_1<v_2<v_3<\ldots<v_n$ e una somma di denaro V, fornire una formula per calcolare il minimo numero di monete richieste per cambiare la somma di denaro V. Assumiamo $v_1=1$ in modo che il problema ammetta sempre una soluzione.

- Ad esempio: Banconota di 6 euro

Valori monete: 1,2,4

- Possiamo cambiare la banconota in 5 modi:
- {1,1,1,1,1,1}, {1,1,1,1,2}, {1,1,2,2},{1,1,4}, {2,4}
- La soluzione che include meno monete e` quindi {2,4}

Minimum Coin Change Problem

- Greedy non sempre funziona
- Strategia Greedy: esamina i valori delle monete in ordine decrescente e per ciascun valore esamninato utilizza quante piu` monete di quel valore
- Sistema di monete canonico: sistema per il quale la strategia greedy fornisce la soluzione ottima
- Esempio di sistema canonico: sistema USA include monete con questi valori 1, 5, 10, 25 cent.
  - Voglio cambiare 8 cent. La strategia greedy produce la soluzione {5,1,1,1} che è la soluzione ottima.
- Esempio di sistema non canonico: 1, 4, 5 cent.
- Voglio cambiare 8 cent. La strategia greedy produce la soluzione {5,1,1,1} mentre la soluzione ottima e` {4,4}

Minimum Coin Change Problem

OPT(i,v)= minimo numero di monete per cambiare una banconota di valore v quando abbiamo a disposizione monete di valore $v_1,\ldots,v_i$

- Se $v_i \le v$, bisogna considerare sia il caso in cui la soluzione include monete di valore v_i sia il caso in cui non le contiene:
  - Numero monete nella soluzione ottima tra quelle che contengono una moneta di valore v_i e` = 1+ numero monete nella soluzione ottima per l'importo $v-v_i$ quando si possono utilizzare monete di valore $v_1,\ldots,v_i$
  - Numero monete nella soluzione ottima tra quelle che non contengono una moneta di valore v_i e` = numero monete nella soluzione ottima per l'importo v quando si possono utilizzare monete di valore $v_1,\ldots,v_{i-1}$ ($v_1=1$)

  → $OPT(i,v)= \min\{OPT(i,v-v_i)+1, OPT(i-1,v)\}$
- Se $v_i > v$, l'unico caso possibile e` quello in cui la soluzione non include monete di valore v_i → $OPT(i,v) = OPT(i-1,v)$
- Se v=0 allora OPT(i,v)=0 per ogni i
- Se i=1 allora OPT(i,v)=v per ogni v

Minimum Coin Change problem

```
MinCoinChange(n, v1,…,vn,V)   //v1<…<vn
For i = 1 to n
  M[i, 0] ← 0 //importo da cambiare = 0 → soluzione contiene 0 monete
For v = 1 to V
  M[1, v] ← v //si possono usare solo monete da un euro
For i = 1 to n
   For v= 1 to V
    if v < vi
     then M[i, v] ← M[i-1,v]
    else
     M[i, v] ← min{1+M[i,v-vi],M[i-1,v]}
```


83

Minimum Coin Change problem

Esercizio: scrivere l'algoritmo che costruisce la soluzione ottima del minimum change coin problem.

84

31/05/21

Coin change problem

- Dato un insieme infinito di monete C di n diversi valori $v_1<v_2<v_3<…<v_n$ ed una certa somma di denaro di valore V, fornire una strategia per trovare in quanti modi possiamo usare le monete in C per cambiare V.
- Ad esempio: Banconota di 6 euro

Valori monete: 1,2,4

- Possiamo cambiare la banconota in 5 modi:
- {1,1,1,1,1,1} {1,1,1,1,2}, {1,1,2,2},{1,1,4}, {2,2,2}, {2,4}

85

Coin change problem

N(i,v)=numero di modi in cui possiamo cambiare v con monete di valore $v_1,…,v_i$

- Se $v_i \le v$ allora la soluzione puo` includere o meno una moneta di valore v_i
  - Dobbiamo sommare il numero di soluzioni che includono monete di valore v_i al numero di soluzioni che non includono monete di valore v_i
  - $N(i,v)= N(i,v-v_i)+ N(i-1, v)$
- Se il valore v_i e` maggiore dell'importo da coprire allora
  - Le soluzioni possibili sono solo quelle che non includono monete di valore v_i
  - $N(i,v)=N(i-1, v)$

86

Sottosequenza comune piu` lunga

Def. Dati una sequenza di caratteri $x=x_1,x_2,\ldots,x_m$ ed un insieme di indici $\{k_1,k_2,\ldots,k_t\}$ tali che $1 \le k_1<k_2<\ldots<k_t \le m$, la sequenza formata dai caratteri di x in posizione $k_1,k_2,\ldots,k_t$ viene detta sottosequenza di x.
N.B. I caratteri della sottosequenza non devono essere necessariamente consecutivi in x.

Problema: Date due sequenze $x=x_1,x_2,\ldots,x_m$ e $y=y_1,\ldots,y_n$, vogliamo trovare la sottosequenza piu` lunga comune ad entrambe le sequenze.

Esempio: x=BACBDAB e y=BDCABA,
BCAB e` una sottosequenza comune a x e y di lunghezza massima.
I caratteri della sequenza BCAB appaiono nelle posizioni 1, 3, 6, 7 in x e nelle posizioni 1, 3, 4, 5 in y.
BDAB e` un'altra una sottosequenza comune a x e y di lunghezza massima.
anche BABA e` un'altra sottosequenza comune a x e y di lunghezza massima.

Sottosequenza comune piu` lunga

Approccio brute force: Per ogni sottosequenza di x controlla se la sottosequenza compare in y.
Ci sono 2^m sottosequenze di x per cui l'algoritmo sarebbe esponenziale.

Perche ci sono 2^m sottosequenze di $x=x_1,x_2,\ldots,x_m$?
Risposta: ogni sottosequenza corrisponde ad una sequenza di m bit dove il k-esimo bit e` 1 se x_k fa parte della sottosequenza e 0 se x_k non fa parte della sottosequenza.

Sottosequenza comune piu` lunga

Input: $x=x_1,x_2,\ldots,x_m$ e $y=y_1,\ldots,y_n$

Sia OPT(i,j) la lunghezza della sottosequenza più lunga comune a $x_1,\ldots,x_i$ e $y_1,\ldots,y_j$.

Per calcolare OPT(i,j) consideriamo i 3 seguenti casi:

- Se $x_i = y_j$ allora la sottosequenza comune piu` lunga termina con $x_i = y_j$
  - In questo caso la soluzione ottima e` formata dalla sottosequenza piu` lunga comune a $x_1,\ldots,x_{i-1}$ e $y_1,\ldots,y_{j-1}$ seguita dal carattere $x_i = y_j$
- Se $x_i \ne y_j$ e la sottosequenza comune piu` lunga termina con un simbolo diverso da $x_i$ allora la soluzione ottima e` data dalla soluzione ottima per $x_1,\ldots,x_{i-1}$ e $y_1,\ldots,y_j$
- Se $x_i \ne y_j$ e la sottosequenza comune piu` lunga termina con un simbolo diverso da $y_j$ allora la soluzione ottima e` data dalla soluzione ottima per $x_1,\ldots,x_i$ e $y_1,\ldots,y_{j-1}$

Sottosequenza comune piu` lunga

Se i=0 o j=0 allora banalmente la sottosequenza comune piu` lunga ha lunghezza 0 perche' almeno una delle due sequenze e` vuota.

$$OPT(i,j) = \begin{cases} OPT(i,j)=0 & \text{se } i=0 \text{ o } j=0 \\ OPT(i-1,j-1)+1 & \text{se } i>0,\ j>0 \text{ e } x_i = y_j \\ \max\{OPT(i-1,j),OPT(i,j-1)\} & \text{altrimenti} \end{cases}$$

Sottosequenza comune piu` lunga: algoritmo

```
ComputaLunghezzaLCS(X,Y)
m← lunghezza di X
n ← lunghezza di Y
For i=1 to m
     M[i,0]← 0
For j=0 to n
     M[0,j]← 0
For i=1 to m
     For j=1 to n
          If x_i=y_j
               Then M[i,j]← 1+M[i-1,j-1]
                    b[i,j]="↖"
          Else if  M[i-1,j]≥M[i,j-1]
               Then M[i,j]← M[i-1,j]
                    b[i,j]="↑"
          Else  M[i,j]← M[i,j-1]
                    b[i,j]="← "
```

L'algoritmo oltre a computare i valori M[i,j]=OPT(i,j), memorizza nelle entrate b[i,j] della matrice b delle frecce in modo che successivamente la sottosequenza comune piu` lunga possa essere ricostruita agevolmente (si veda algoritmo nella slide successiva).

Sottosequenza comune piu` lunga: algoritmo

x=BACBDAB e y=BDCABA

		∅	B	D	C	A	B	A
		0	**1**	**2**	**3**	**4**	**5**	**6**
∅	**0**	0	0	0	0	0	0	0
B	**1**	0	1	1	1	1	1	1
A	2	0	1	1	1	2	2	2
C	3	0	1	1	2	2	2	2
B	4	0	1	1	2	2	3	3
D	5	0	1	2	2	2	3	3
A	6	0	1	2	2	3	3	4
B	7	0	1	2	2	3	4	4

Seguendo le frecce viene stampata BABA

31/05/21

L'algoritmo che stampa la sottosequenza comune piu` lunga

```
Stampa-LCS(b,X,i,j)
If i=0 or j=0
    Then return
If b[i,j]="↖"
  Then Stampa-LCS(b,X,i-1,j-1)
          print(x_i )
Else if b[i,j]="↑"
  Then Stampa-LCS(b,X,i-1,j)
Else Stampa-LCS(b,X,i,j-1)
```