# Cammini minimi in un DAG

Affrontiamo ora un tipo di problema che illustra una versione un po' più complicata della programmazione dinamica.

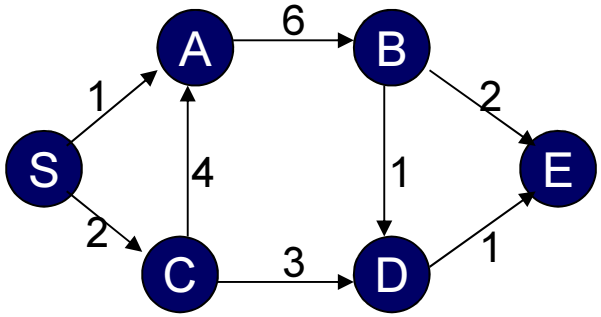
Nell'esempio precedente abbiamo definito una ricorrenza basata su una scelta binaria: l'intervallo n apparteneva o non apparteneva a una soluzione ottima. Nel problema che consideriamo ora, la ricorrenza coinvolge quelle che possiamo chiamare "scelte multiple": ad ogni passo, per la struttura di una soluzione ottima, si devono considerare un numero polinomiale di possibilità.

La programmazione dinamica si adatta in modo molto naturale a questa situazione più generale.

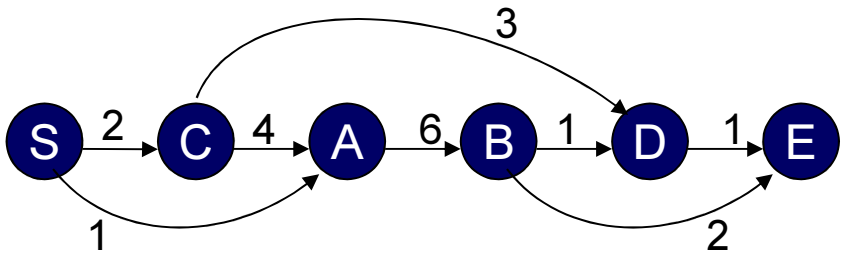
Consideriamo il problema di trovare i ***cammini di lunghezza minima*** in un grafo orientato aciclico (DAG) e pesato.

# Cammini minimi in un DAG

Dato il seguente DAG, supponiamo di voler trovare le distanze di tutti i nodi dal nodo S.




Ordinamento topologico




Ad esempio focalizziamoci sul nodo D. L'unico modo per raggiungerlo dal nodo S è attraverso uno dei suoi predecessori, B o C; il cammino più breve sarà ottenuto dal confronto delle due strade:

$$dist(D) = \min \{dist(B) + 1, dist(C) + 3\}$$

# Cammini minimi in un DAG

Possiamo scrivere la stessa relazione per ogni nodo v:

$$dist(v) = \min \{dist(u) + w(u,v) \mid (u,v) \in E\}$$

dove w è la funzione peso.

Se consideriamo i nodi in ordine da sinistra a destra, quando si considera un nodo v, si dispone sempre dell'informazione necessaria a calcolare *dist*(v), supposto che sia definita la distanza *dist*(S), che ovviamente assumeremo uguale a 0.

La complessità nel caso peggiore di un algoritmo basato su questa definizione sarà data dalla somma della complessità dell'ordinamento topologico ($O(V + E)$) e del numero di confronti per calcolare le distanze ($O(V^2)$), cioè $O(V^2)$

# Cammini minimi in un DAG

La tecnica usata è molto generale: in ogni nodo calcoliamo una funzione dei valori dei nodi predecessori (nel nostro caso un minimo, ma potrebbe essere un massimo, una somma, …)

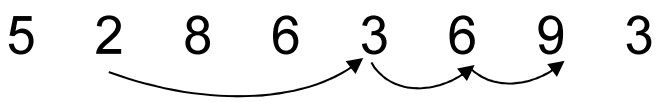
Nella programmazione dinamica non si ha un DAG:
il DAG è implicito.
I suoi nodi sono i sottoproblemi e gli archi
esprimono le dipendenze tra i sottoproblemi.

# Lunghezza massima di una sottosequenza crescente

Data una sequenza di numeri $a_1, a_2, \ldots, a_n$, definiamo sua sottosequenza un sottinsieme di numeri presi nell'ordine, della forma: $a_{i1}, a_{i2}, \ldots, a_{ik}$, con $1 <= i_1 < i_2 < \ldots < i_k <= n$

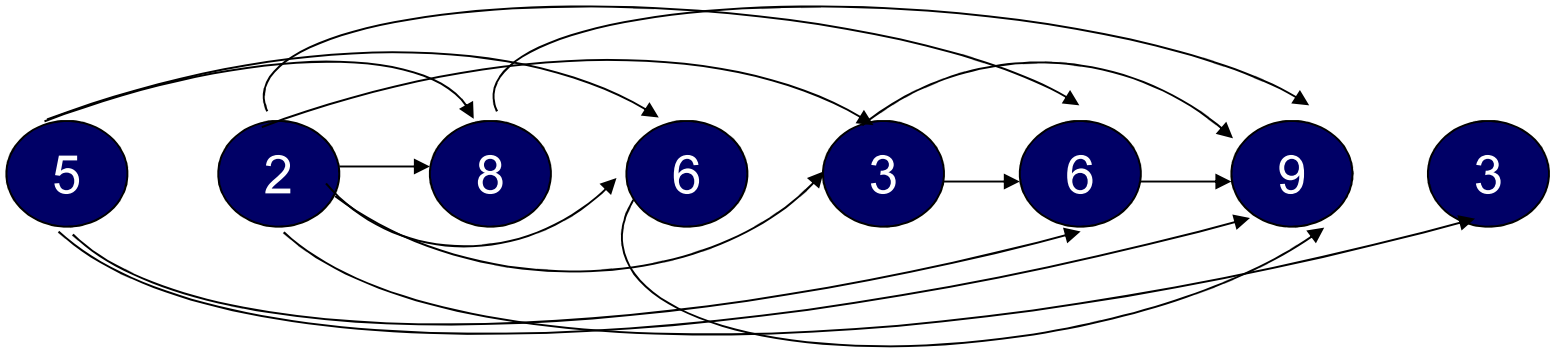
Il problema che vogliamo risolvere è quello di determinare una sottosequenza crescente più lunga in una sequenza data.

Ad esempio una delle sottosequenze di lunghezza massima nella sequenza 5 2 8 6 3 6 9 3 è 2 3 6 9:



# Lunghezza massima di una sottosequenza crescente

Il grafo che rappresenta questo problema può essere ottenuto associando un vertice i ad ogni elemento $a_i$ della sequenza e introducendo un arco orientato (i,j) se $a_i$ precede $a_j$ nella sequenza e $a_i < a_j$.




Il grafo così ottenuto è un DAG, sul quale il problema diventa quello di trovare il cammino di lunghezza massima.

# Lunghezza massima di una sottosequenza crescente

Si ottiene pertanto la seguente definizione della lunghezza massima di una sottosequenza crescente fino all'elemento j-esimo:

$$L(j) = \max \{L(i) \mid i < j \ \& \ (i,j) \in E\} + 1$$

Definiamo ora il problema indipendentemente dal grafo costruito. Se X[1..n] e` il vettore che memorizza i numeri della sequenza, la condizione $(i,j) \in E$ viene sostituita da X[i] < X[j].

$$L(1) = 1$$
$$L(j) = \max \{L(i) \mid i < j \ \& \ X[i] < X[j]\} + 1$$

Convenzione: il massimo su un insieme vuoto è 0.

L(1) è il problema più piccolo a cui sappiamo dare subito risposta. La sottosequenza di lunghezza massima della sequenza di lunghezza 1 è ovviamente formata da un elemento.

# Lunghezza massima di una sottosequenza crescente

Esempio: per la sequenza 5 2 8 6 3 6 9 3 i valori di L(j) sono i seguenti:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| X | 5 | 2 | 8 | 6 | 3 | 6 | 9 | 3 |
| L | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 2 |

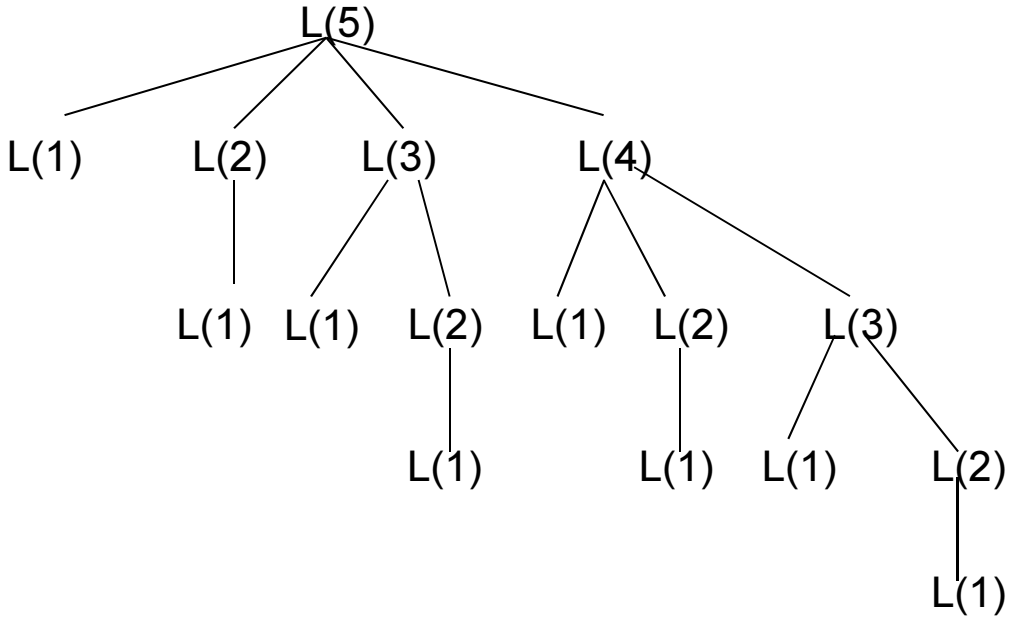
Ricorsione? No grazie

# Lunghezza massima di una sottosequenza crescente

*Supponiamo, nel caso peggiore, che il grafo abbia un arco da ogni vertice a tutti quelli che lo seguono nell'ordine topologico.* Avremo allora:

$$L(j) = \max\{L(1), L(2), \ldots L(j-1)\} + 1$$

Ad esempio per j = 5 si avrebbe il seguente albero delle chiamate:

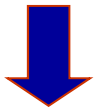



Con un numero di nodi esponenziale in n !

# Lunghezza massima di una sottosequenza crescente

I sottoproblemi sono di poco più piccoli del problema da risolvere. Per calcolare L(n) serve il valore L(n-1)!

L'albero di ricorsione ha profondità polinomiale nei dati in input e un numero di nodi esponenziale, ma i nodi sono ripetuti e i sottoproblemi distinti sono pochi.



Conviene l'algoritmo iterativo

L'algoritmo ricorsivo ha una complessità polinomiale quando i sottoproblemi hanno dimensione sostanzialmente inferiore a quella del problema.

# Lunghezza massima di una sottosequenza crescente

```
lung-ss (X[], n)
    var: MX, L
    L[1] ←1
    for i ← 2 to n
            L[i] ← 1
            for j ← 1 to i – 1
                if (X[j] < X[i] and L[j] + 1 > L[i])
                    L[i] ← L[j] + 1
    MX ← L[1]
    for i ← 2 to n
            if (L[i] > MX)   MX ← L[i]
    return MX
```

# Lunghezza massima di una sottosequenza crescente

Se vogliamo conoscere anche la sottosequenza?

Un modo è quello di ricordare per ogni elemento il suo predecessore (o la posizione del predecessore) nella sequenza più lunga a lui associata.

Riprendiamo l'esempio sulla sequenza 5 2 8 6 3 6 9 7 e indichiamo con 0 la non esistenza di un predecessore.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| X | 5 | 2 | 8 | 6 | 3 | 6 | 9 | 3 |
| L | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 2 |
| PR | 0 | 0 | 1 | 1 | 2 | 5 | 6 | 2 |

# Lunghezza massima di una sottosequenza crescente

L'algoritmo precedente può essere facilmente modificato:

```
lung-ss (X[])
    var: MX, indice, L[], PR[], SMAX[]    \\per SMAX vedi dopo
    L[1] ←1;  PR[1] ← 0
    for i ← 2 to n
            L[i] ← 1; PR[i] ← 0
            for j ← 1 to i – 1
                if (X[j] < X[i] and L[j] + 1 > L[i])
                    L[i] ← L[j] + 1
                    PR[i] ← j
    MX ← L[1] ;   indice ← 1
    for i ← 2 to n
            if (L[i] > MX)   MX ← L[i] ; indice ← i
   . . . . . .
    return MX
```

# Lunghezza massima di una sottosequenza crescente

Inseriamo inoltre prima del return la preparazione dell'array che contiene la soluzione.

```
SMAX [MX] ← X[indice]
i ← MX
while (PR[indice] > 0)
        i ← i - 1
        SMAX[i] ← X[PR[indice]]
        indice ← PR[indice]
return (MX, SMAX)
```