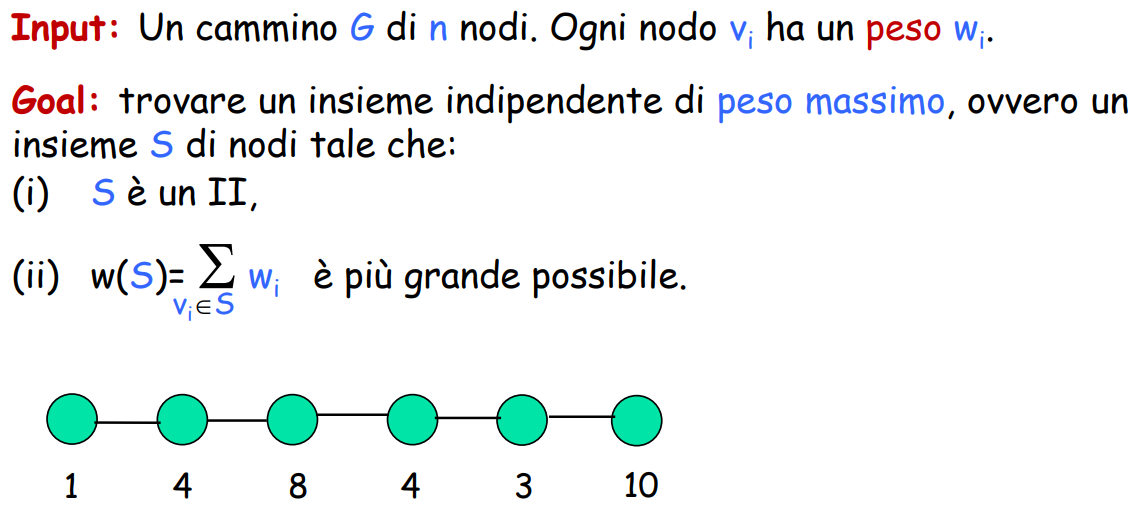
**Programmazione dinamica**

Un problema interessante: insieme indipendente di peso massimo (per un grafo a cammino).

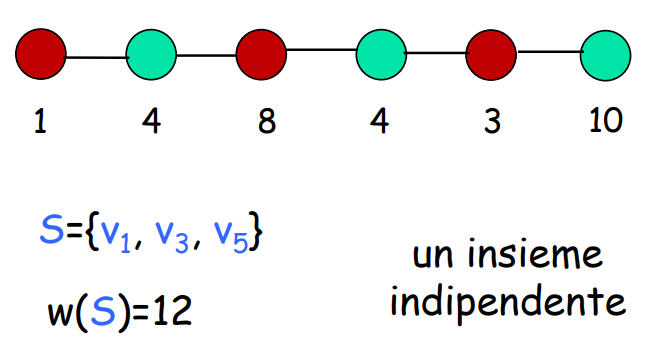
Principi generali della programmazione dinamica – sottoproblemi, relazioni fra sottoproblemi, tabelle.

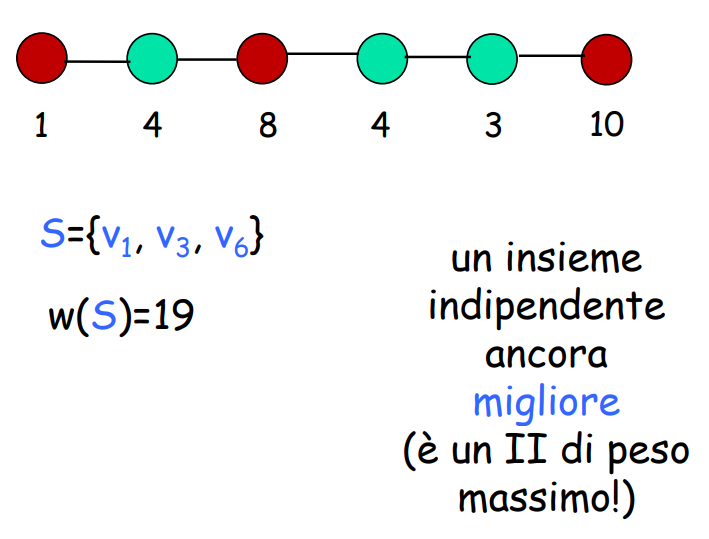
**Insieme Indipendente di peso massimo (su grafi a cammino).**

***Un insieme indipendente (II) di G è un sottoinsieme di nodi che non contiene due nodi adiacenti, ovvero per ogni coppia di nodi dell’insieme i due nodi non sono collegati da un arco.***

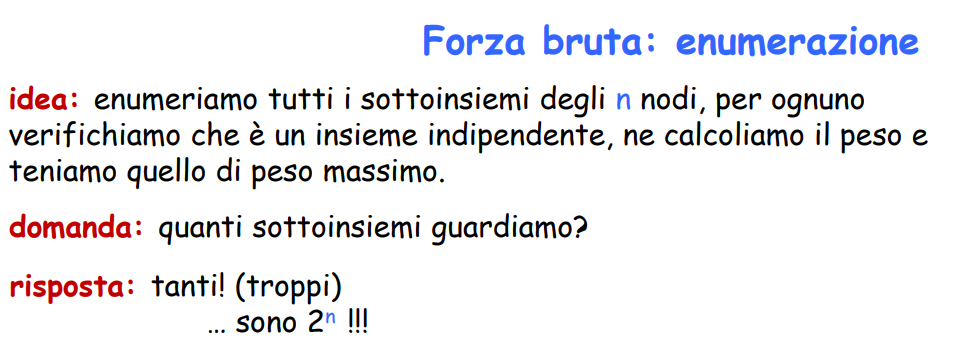


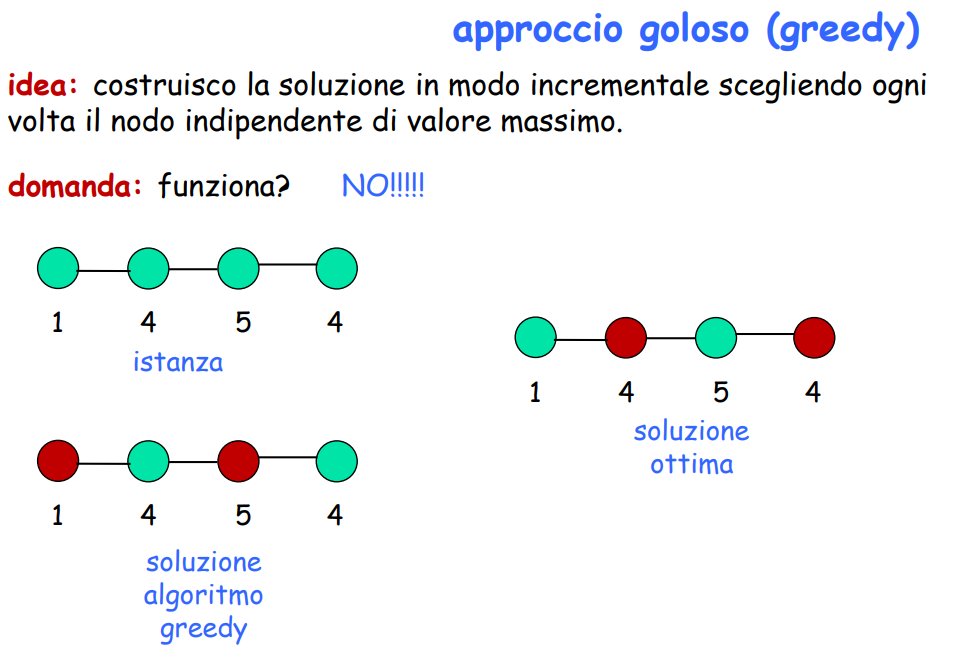
Esempi:

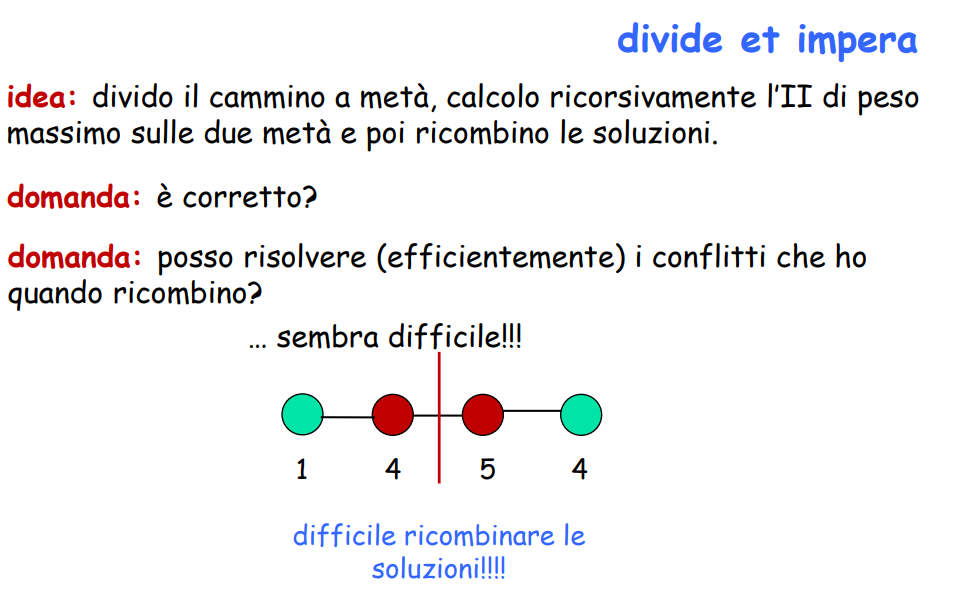
 



Per poter progettare l’algoritmo di risoluzione, ci si può approcciare in diversi modi:







## **L'approccio della programmazione dinamica:**

**Passaggio critico:**

* Ragionare sulla struttura e le proprietà della soluzione **ottima** del problema.

**Similarità con divide-et-impera:**

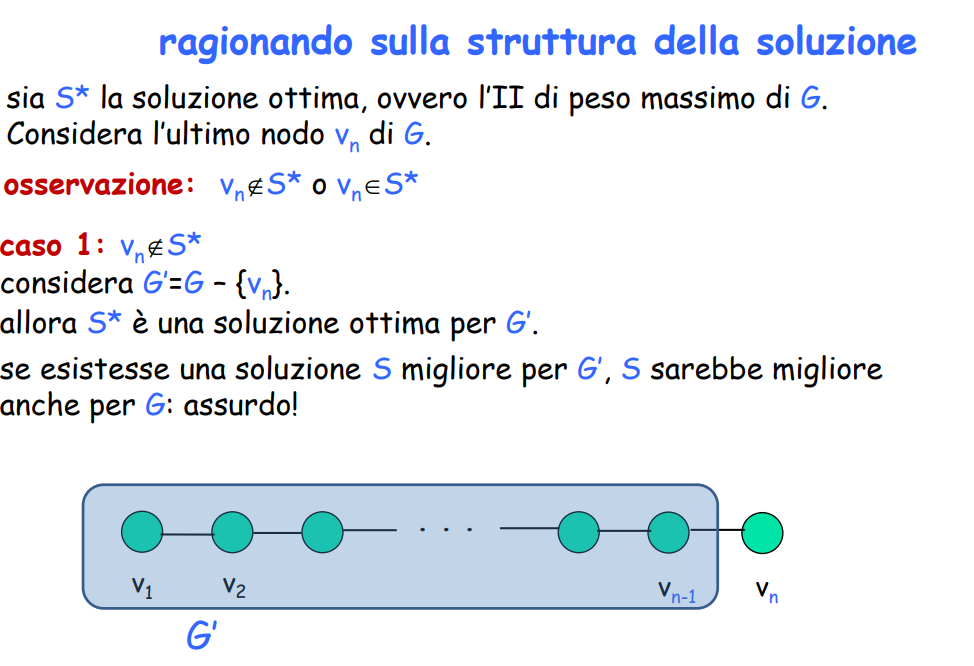
* Si ragiona in termini di soluzioni **ottime** di sottoproblemi più "piccoli", in modo simile a come si fa implicitamente nella tecnica del divide-et-impera.

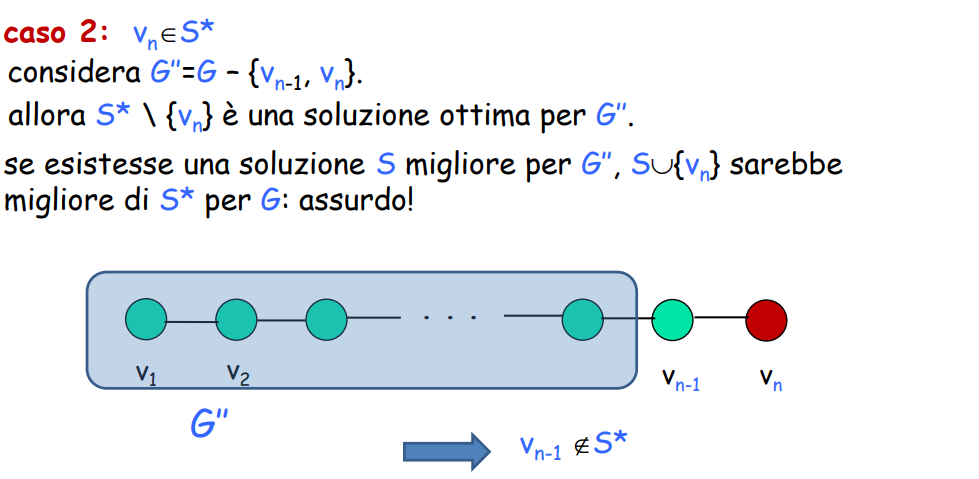
**Obiettivo:**

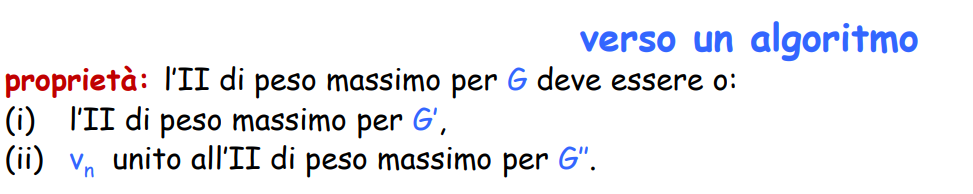
* Esprimere la soluzione del problema come una combinazione di soluzioni di (opportuni) sottoproblemi.

**Ricerca per forza bruta:**

* Se le combinazioni sono "poche", possiamo cercare la combinazione giusta per forza bruta.

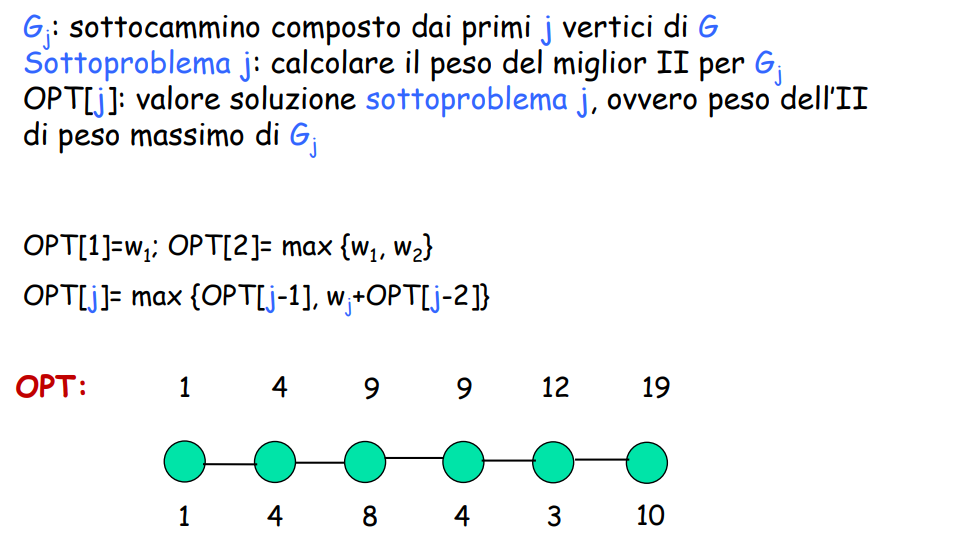


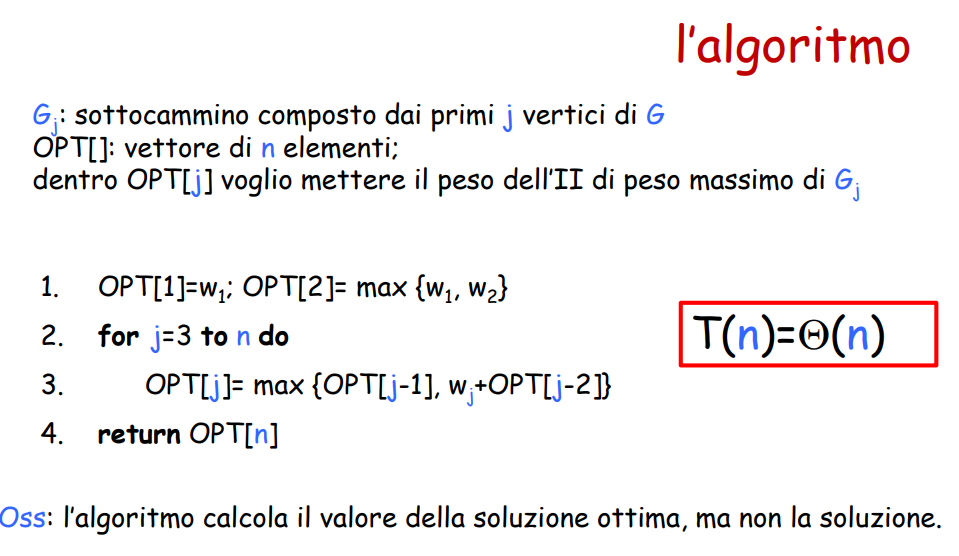


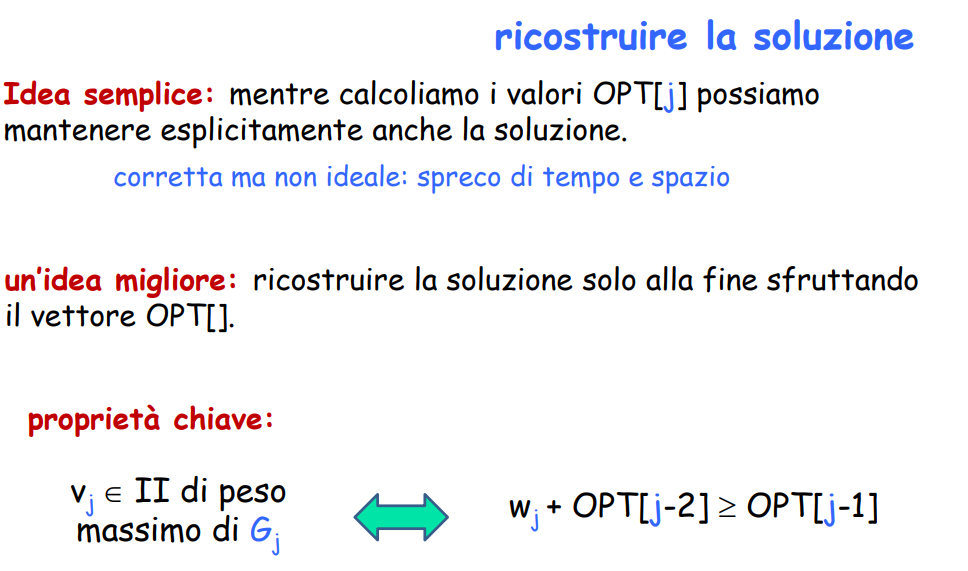


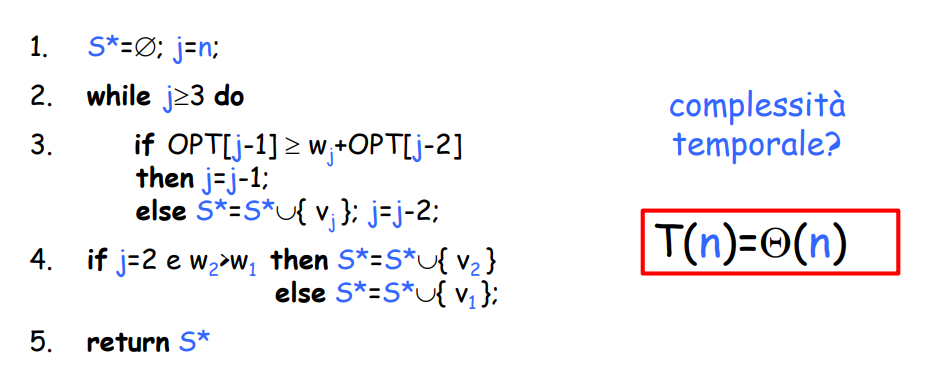
**N.B. Potremmo sviluppare la formula di Fibonacci2 ma il costo sarebbe esponenziale.**

**Procediamo iterativamente considerando prefissi di G dai più piccoli verso i più grandi:**

****

****

****

****

## 

## 

## **Programmazione Dinamica: principi generali**

**Identificare i sottoproblemi:**

1. **Individuare un numero ristretto di sottoproblemi che compongono il problema originale.**

**Ricorsione:**

1. **Descrivere la soluzione di un generico sottoproblema in funzione delle soluzioni di sottoproblemi "più piccoli".**

**Memorizzazione:**

1. **Memorizzare le soluzioni dei sottoproblemi in una tabella per evitare ricalcoli superflui.**

**Calcolo progressivo:**

1. **Avanzare opportunamente nella tabella, calcolando la soluzione del sottoproblema corrente in base alle soluzioni già memorizzate.**

**Proprietà desiderabili dei sottoproblemi:**

1. **Numero limitato: Per mantenere l'algoritmo efficiente, il numero di sottoproblemi dovrebbe essere contenuto.**
2. **Complessità ridotta: La risoluzione di tutti i sottoproblemi deve portare a un calcolo rapido della soluzione originale.**
3. **Struttura ricorsiva: Devono esistere sottoproblemi "piccoli" che si ricombinano per formare la soluzione completa.**
4. **Ordinamento ben definito: I sottoproblemi devono essere risolvibili in un ordine preciso per ottenere la soluzione corretta.**

**Ruolo chiave dei sottoproblemi:**

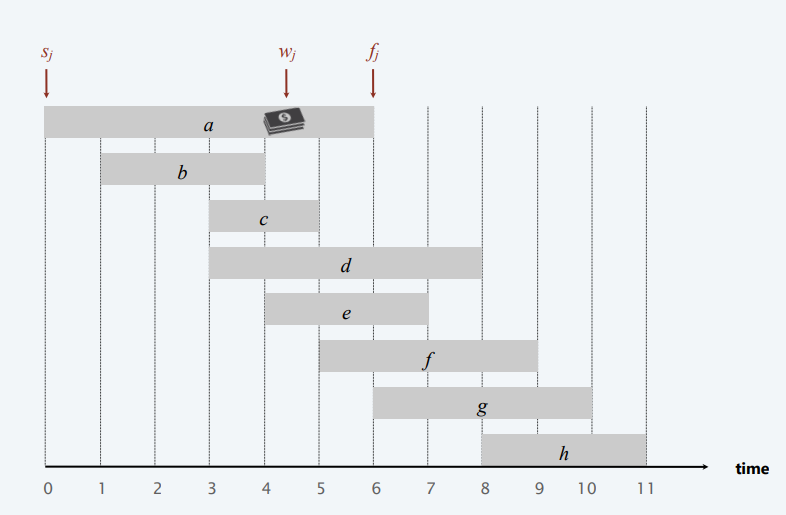
* **La definizione dei "giusti" sottoproblemi è fondamentale per il successo dell'algoritmo.**
* **La loro individuazione è un processo di scoperta che richiede intuito e analisi del problema.**
* **Solo dopo aver definito i sottoproblemi si può verificarne la correttezza e l'efficacia.**
* **La struttura della soluzione desiderata può fornire indicazioni utili per la loro identificazione e l'ordinamento.**

**Weighted interval scheduling (& memoization)**

**Il lavoro j inizia a sj, termina a fj e ha un peso wj > 0.**

**Due lavori sono compatibili se non si sovrappongono.**

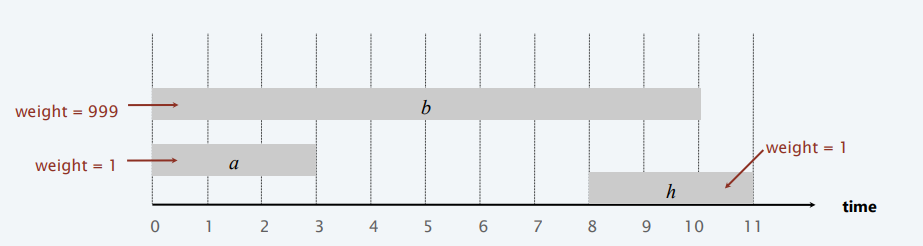
**Obiettivo: trovare un sottoinsieme a peso massimo di lavori reciprocamente compatibili.**

****

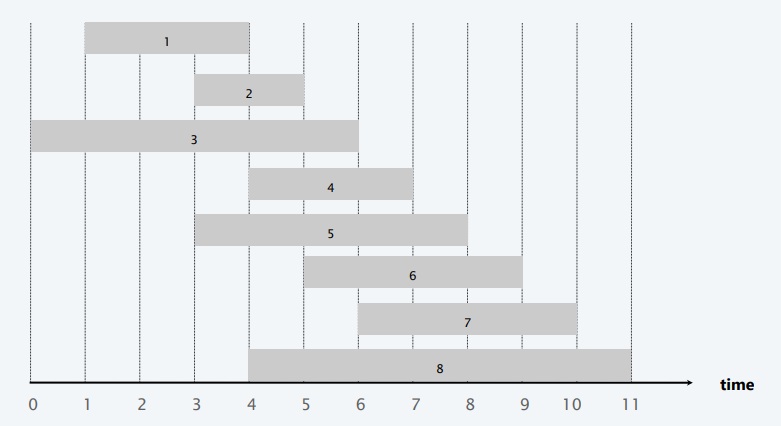
**Prendiamo in considerazione l’algoritmo greedy Earliest finish time first**

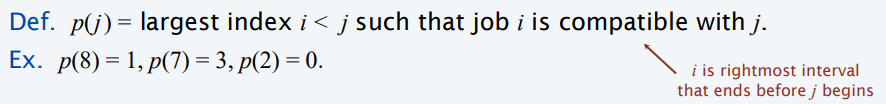
* **Considera i lavori in ordine crescente di tempo di completamento.**
* **Aggiungere un lavoro al sottoinsieme se è compatibile con i lavori scelti in precedenza.**
* **L'algoritmo Greedy è corretto se tutti i pesi sono 1.**

**Osservazione. L'algoritmo Greedy fallisce in modo spettacolare per la versione pesata.**

****

**Analisi:  
  
Convenzione. I lavori sono in ordine crescente di tempo di completamento: f1 ≤ f2 ≤ . . . ≤ fn.**

****

****

**P(j) è il più grande indice i relativo a job compatibili con j.**

**Algoritmo Programmazione Dinamica:**

**Definizione. OPT(j) = peso massimo di qualsiasi sottoinsieme di lavori reciprocamente compatibili per il sottoproblema costituito solo dai job 1, 2,.., j.**

**Goal. OPT(n) = peso massimo di qualsiasi sottoinsieme di lavori reciprocamente compatibili.**

**Caso 1. OPT(j) non contiene il job j.**

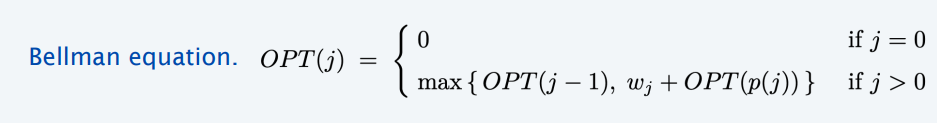
**・Deve essere una soluzione ottimale al problema costituito dai lavori rimanenti 1, 2, ..., j - 1.**

**Caso 2. OPT(j) seleziona il lavoro j.**

**・Raccoglie il profitto wj.**

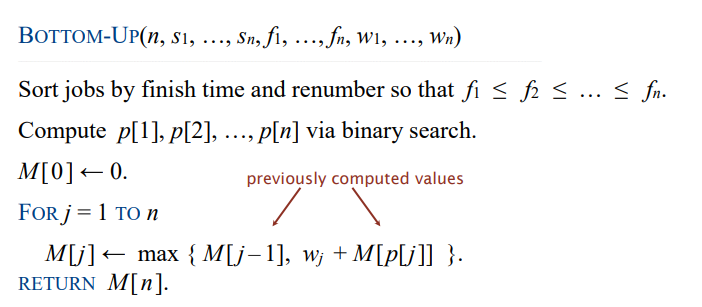
**・Non può utilizzare lavori incompatibili { p(j) + 1, p(j) + 2, ..., j - 1 }.**

**・Deve includere la soluzione ottimale al problema costituito dai restanti lavori compatibili 1, 2, ..., p(j).**

****

**Bellman = Equazione di ricorrenza che vi descrive il valore dell’ottimo. analizziamo il caso j > 0, opt j - 1 è la migliore soluzione per i primi j-1 job, viene interpretata come la migliore soluzione senza il job j. wj+optpj sarà la migliore soluzione per i primi j job sum wj.**

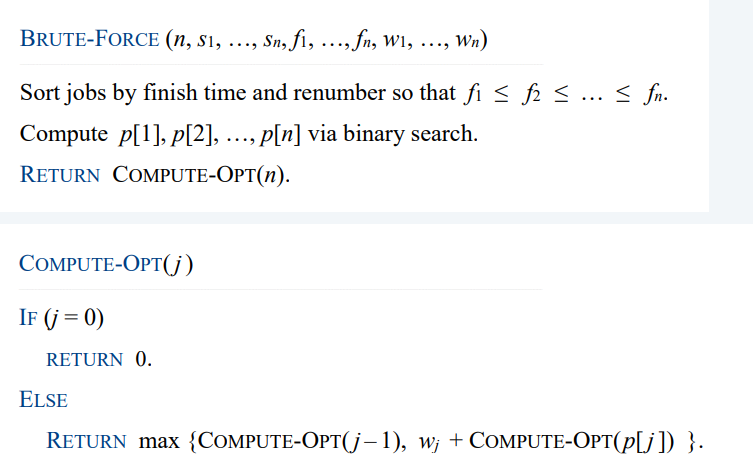
**Bottom-Up Dynamic Programming:**

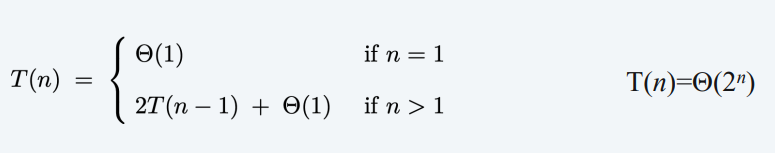
**Risoluzione in modo crescente.  
  
**

**Running Time:**

* **La versione bottom-up richiede un tempo O(n log n).**
* **Ordinamento per tempo di arrivo: O(n log n) tramite mergesort.**
* **Computare p[j] per ogni j: O(n log n) tramite ricerca binaria.**
* **Il ciclo FOR richiede un tempo O(n).**

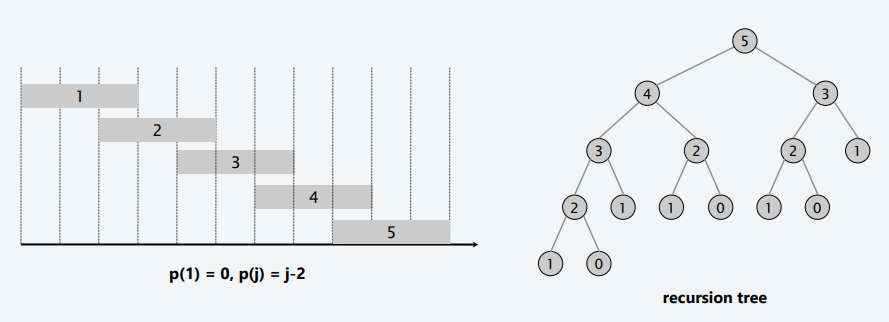
**Approccio Top-Down:  
turning back to recursion**

****

**Complessità Temporale:  
**

**Osservazione. L'algoritmo ricorsivo è spettacolarmente lento a causa della sovrapposizione dei sottoproblemi ⇒ algoritmo a tempo esponenziale.**

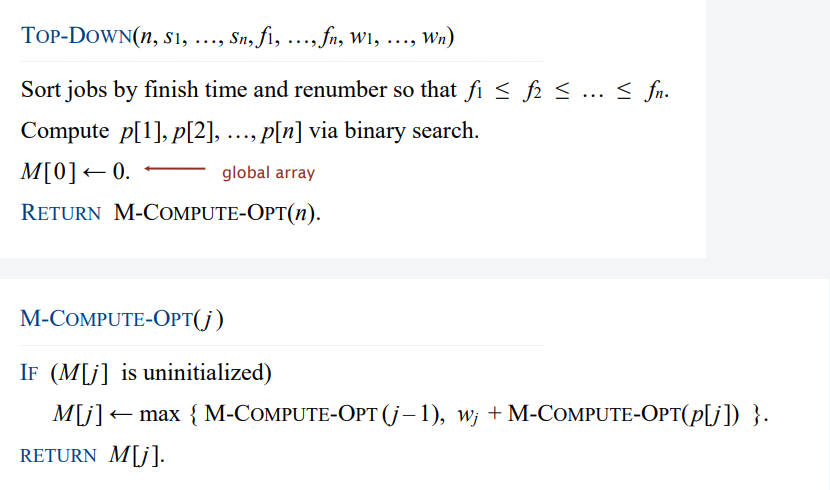
**Es. Il numero di chiamate ricorsive per una famiglia di istanze "a strati" cresce come la sequenza di Fibonacci.**

****

**memoization:**

**Top-down dynamic programming (memoization). Modalità Decrescente.**

**・Risultato Cache del sottoproblema j in M[ j ].**

**・Utilizzare M[j] per evitare di risolvere il sottoproblema j più di una volta.**

**La versione memoization richiede un tempo O(n log n).**

**Pf.  
Ordinamento per tempo di completamento: O(n log n) tramite mergesort.   
・Computare p[j] per ogni j: O(n log n) tramite ricerca binaria.**

**・M-COMPUTE-OPT(j): ogni invocazione richiede O(1) tempo e o :**

**- (1) restituisce un valore inizializzato M[j]**

**- (2) inizializza M[j] ed effettua due chiamate ricorsive**

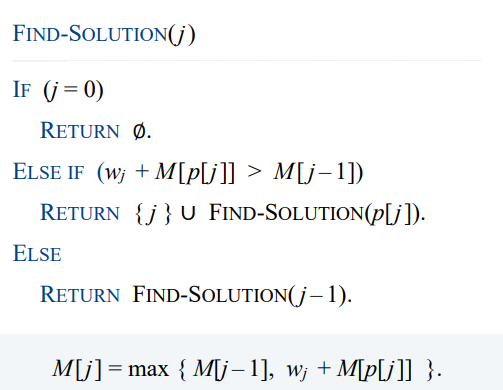
**・Misura di progresso Φ = # voci inizializzate tra M[1.. n].**

**- inizialmente Φ = 0; in tutto Φ ≤ n.**

**- (2) aumenta Φ di 1 ⇒ ≤ 2n chiamate ricorsive.**

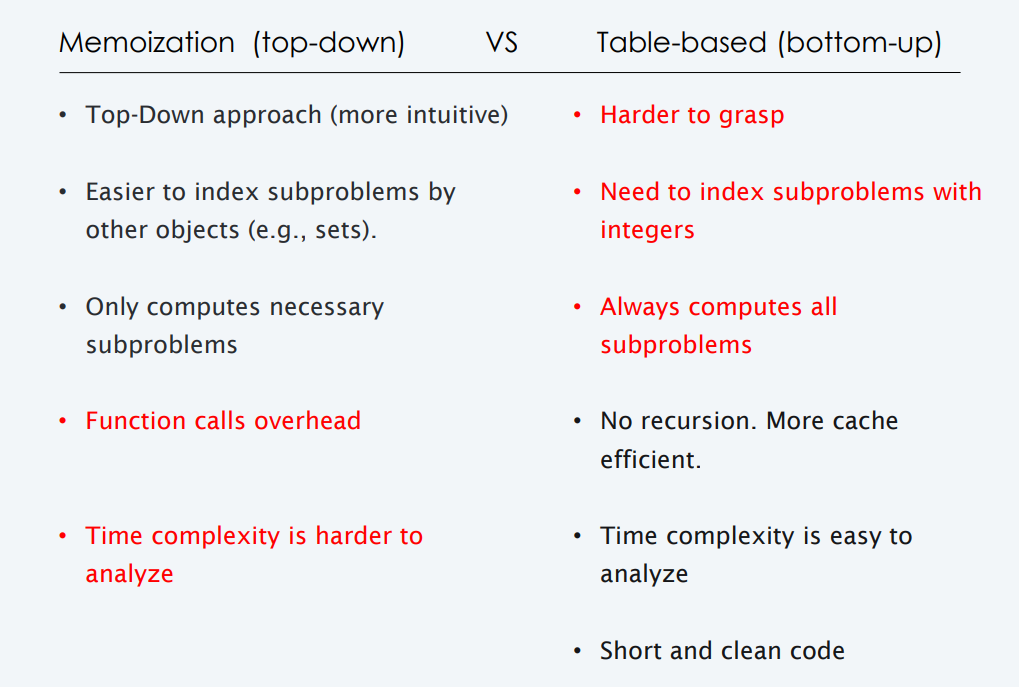
**Il tempo di esecuzione complessivo di M-COMPUTE-OPT(n) è O(n). ▪**

**Find-Solution:**

****

**Analisi. # Numero di chiamate ricorsive ≤ n ⇒ O(n).**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

****

**‣ Longest Increasing Subsequence**

**Problema: Bere il più possibile**

**Robert vuole bere il più possibile.**

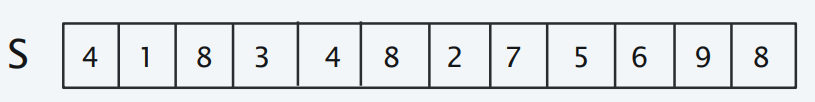
**- Robert cammina per le strade di Approdo del Re e incontra n taverne t1, t2, ..., tn, nell'ordine**

**- Quando Robert incontra una taverna ti, può fermarsi a bere o continuare a camminare.**

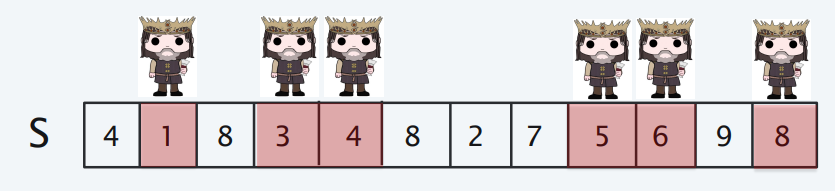
**- Il vino servito nella taverna ti ha una gradazione si (più alta, più forte).**

**- La forza delle bevande di Robert deve aumentare nel tempo**

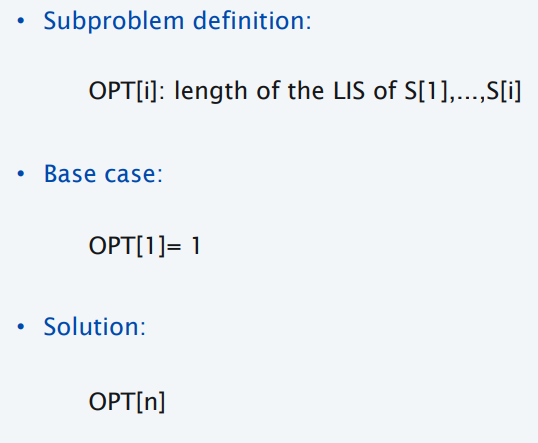
**- Obiettivo: calcolare il numero massimo di soste per bere di Robert**

****

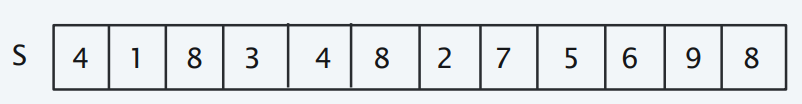
**=**

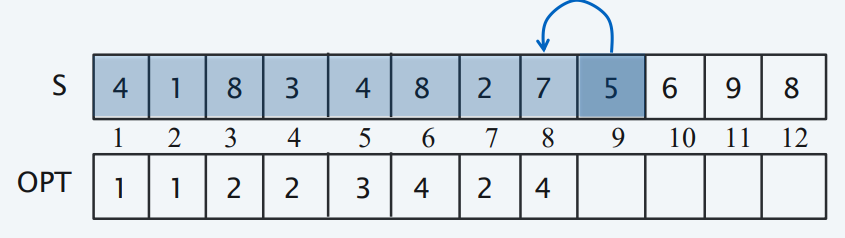
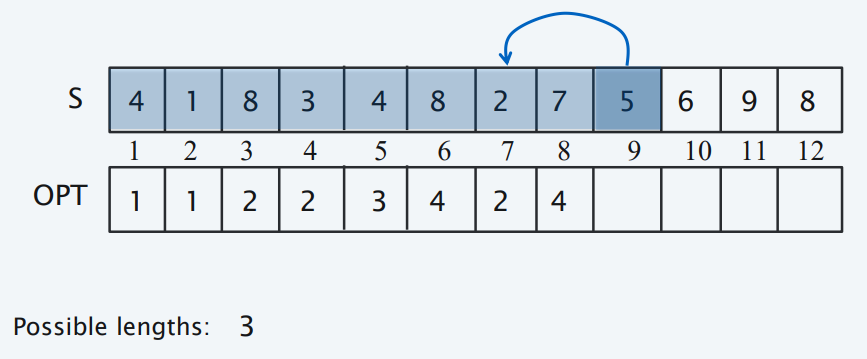
****

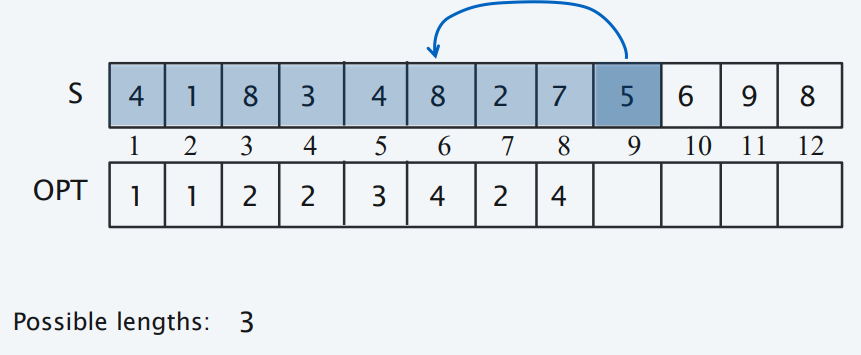
**optimal solution: 6   
This is a problem known as Longest Increasing Subsequence.**

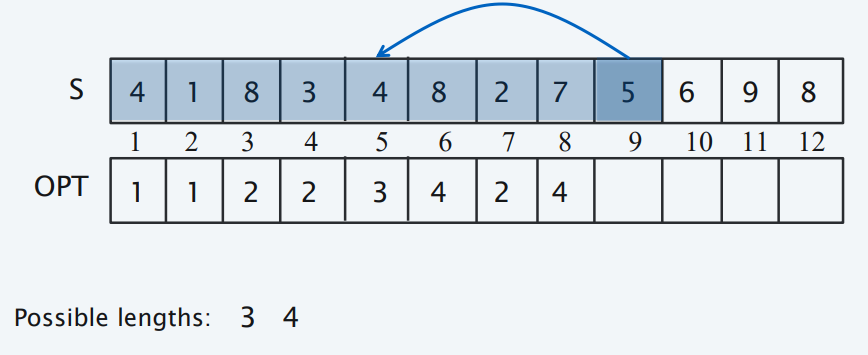
**Considerazioni:  
  
  
Suggerimento: a volte aggiungere vincoli ai sottoproblemi può essere utile!**

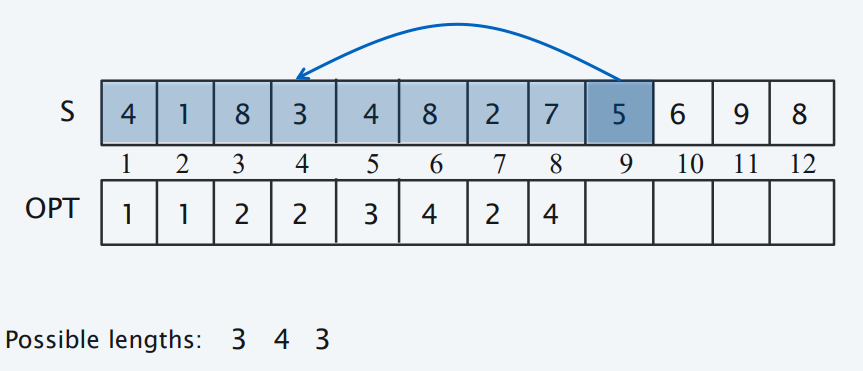
**OPT[i]: lunghezza del LIS di S[1],...,S[i] che termina con S[i]**

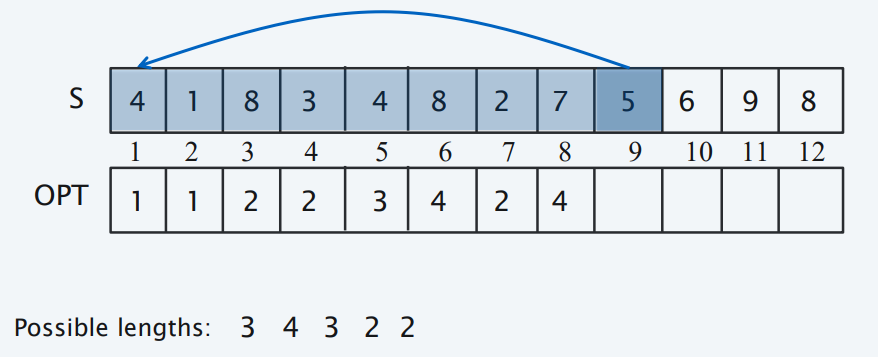
****

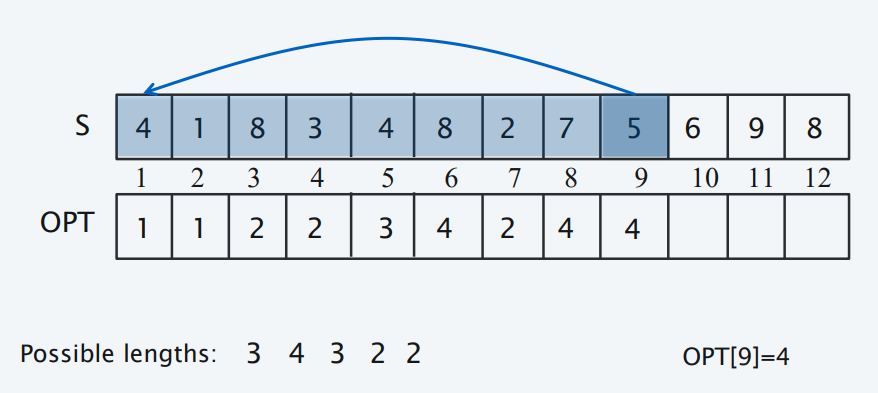
**Non posso combinare il 5 con il 7(8).  
  
**

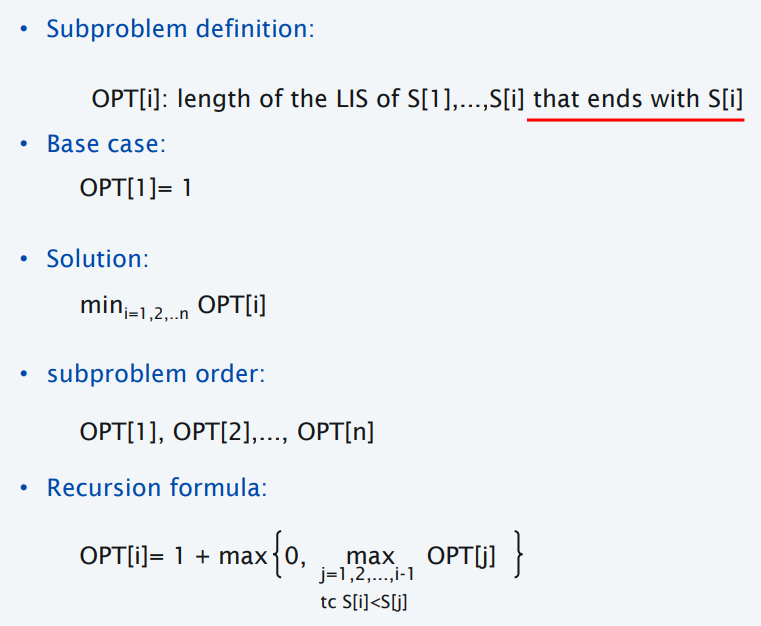
****

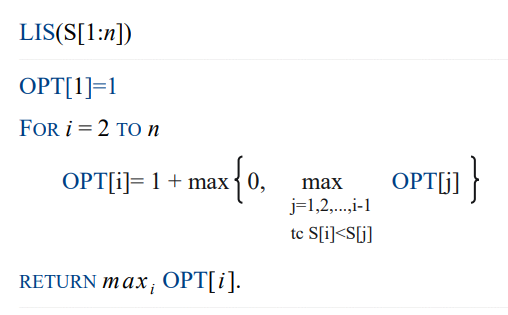
****

****

****

****

**Osservazioni post analisi:  
  
  
Correzione → tc S[j] < S[i].**

**Algoritmo Longest Increasing Subsequence:  
**

**Running time.**

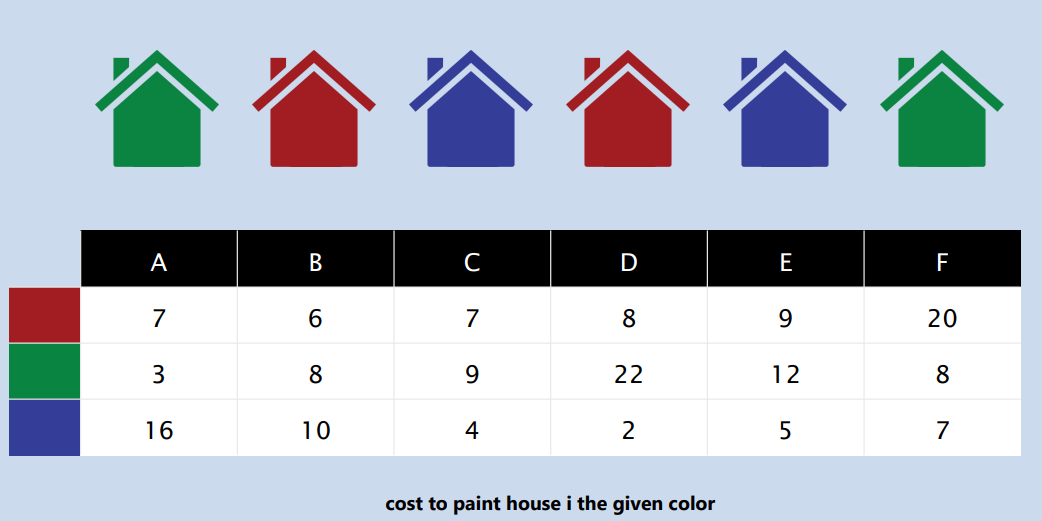
* **Ogni OPT[i] viene calcolato in tempo O(i)=O(n).**
* **Tempo O()**

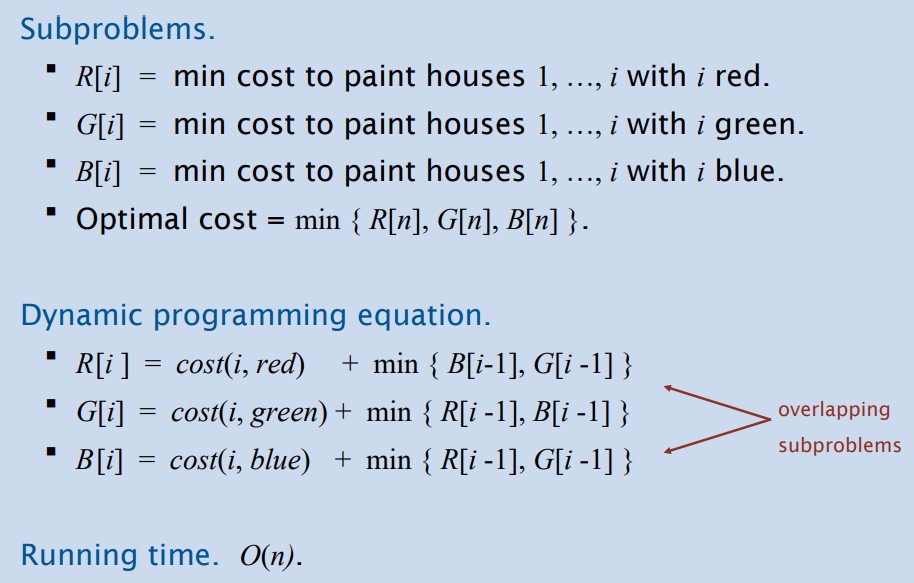
**HOUSE COLORING PROBLEM**

**Goal. Dipingere una fila di n case di colore rosso, verde o blu in modo che**

**・nessuna casa adiacente abbia lo stesso colore.**

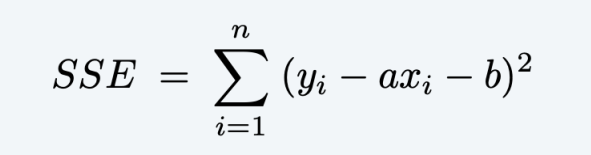
**・Minimizzare il costo totale, dove costo(i, colore) è il costo per dipingere i dato colore.**

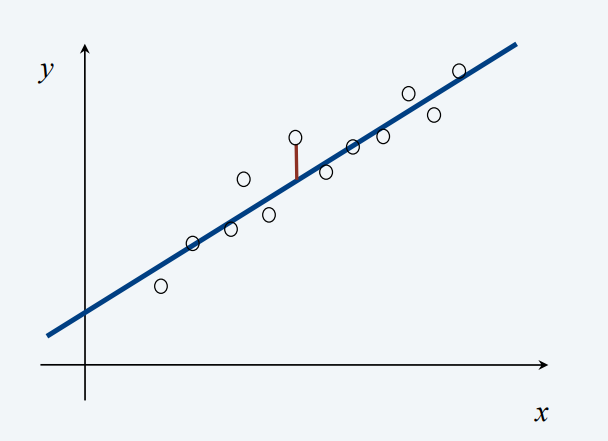
****

****

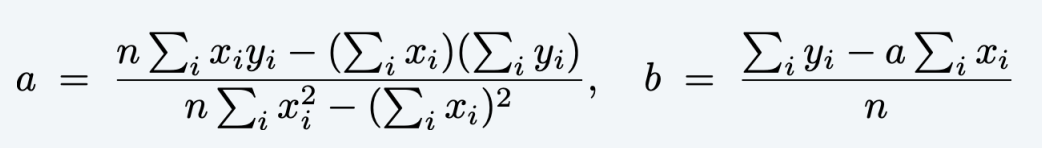
**‣ segmented least squares**

**I minimi quadrati. Problema fondamentale della statistica.**

* **Dati n punti nel piano: (x1, y1), (x2, y2), ..., (xn, yn).**
* **Trovare una retta y = ax + b che minimizzi la somma dell'errore quadratico: **

****

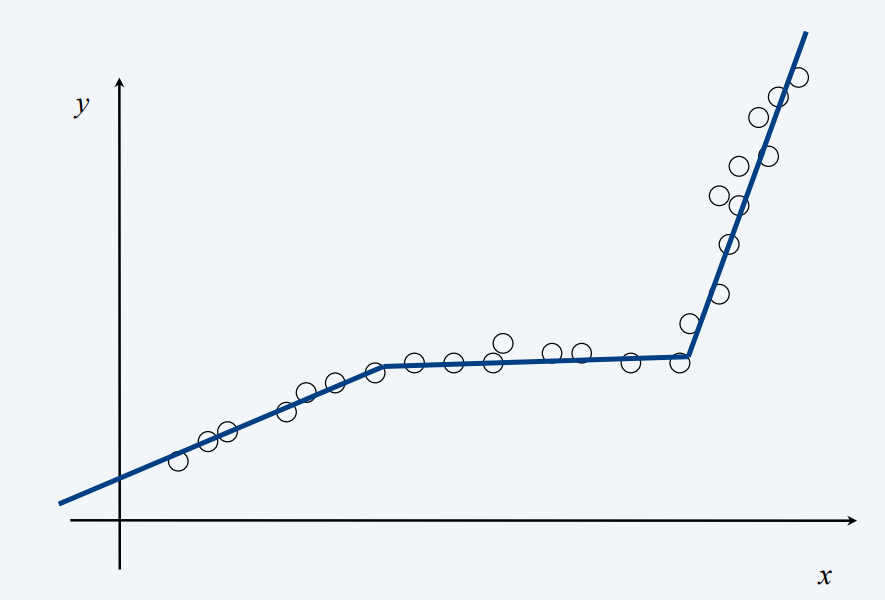
**Soluzione. Il calcolo ⇒ L'errore minimo si ottiene quando**

****

**Minimi quadrati segmentati.**

* **I punti giacciono approssimativamente su una sequenza di diversi segmenti di linea.**
* **Dati n punti nel piano: (x1, y1), (x2, y2) , ..., (xn, yn) con x1 < x2 < ... < xn, trovare una sequenza di rette che minimizzi f(x).**

**Qual è una scelta ragionevole per f(x) per bilanciare accuratezza e parsimonia?**

****

**Obiettivo. Minimizzare f (x) = E + c L per qualche costante c > 0, dove**

**・E = somma delle somme degli errori al quadrato in ogni segmento.**

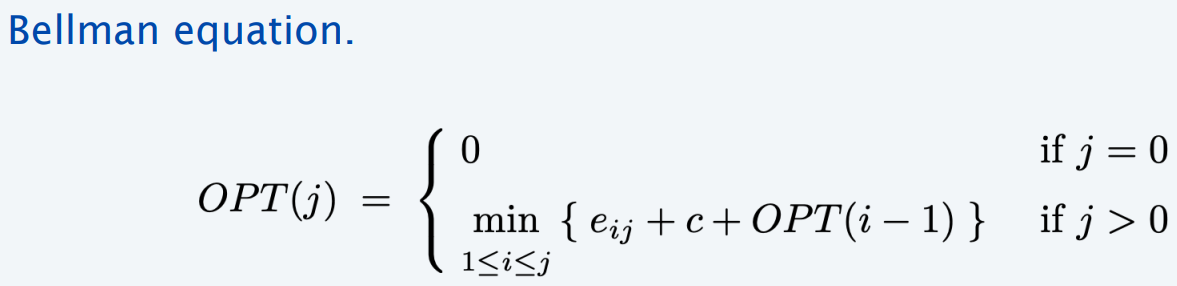
**・L = numero di linee.**

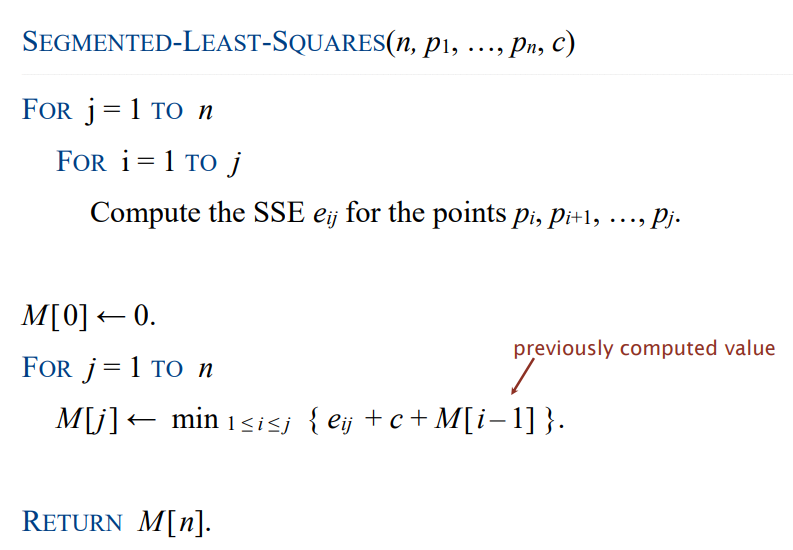
**Dynamic programming: multiway choice**

**Notazione.**

**・OPT(j) = costo minimo per i punti p1, p2, ..., pj.**

**・ = SSE per i punti pi, pi+1, ..., pj.**

**Per calcolare OPT(j):  
・L'ultimo segmento utilizza i punti pi, pi+1, ..., pj per qualche i ≤ j.  
・Costo = + c + OPT(i - 1).  
  
**

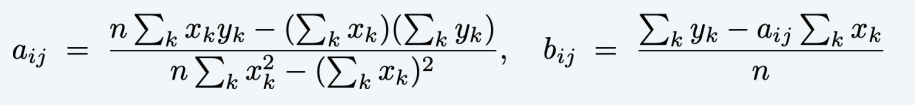
**Algoritmo Pseudocodice:  
  
**

**Segmented least squares analysis :**

***Teorema. [Bellman 1961]*   
L'algoritmo DP risolve il problema dei minimi quadrati segmentati in O() tempo e O() spazio.**

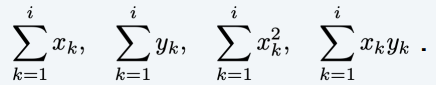
**Pf.**

**・Collo di bottiglia = calcolo di SSE per ogni i e j.**

****

**・O(n) per calcolare . ▪**

**Osservazione.   
Può essere migliorato in tempo O().**

**・Per ogni i: precalcolare le somme cumulative   
  
Utilizzando le somme cumulative, è possibile calcolare in tempo O(1).**

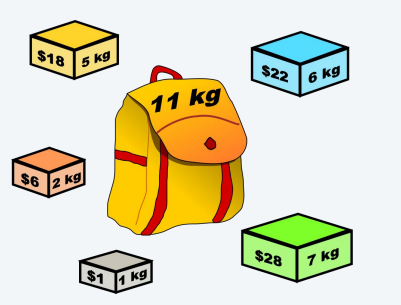
**‣ knapsack problem**

**Problema Knapsack**

**Obiettivo. Imballare lo zaino in modo da massimizzare il valore totale degli oggetti presi.**

* **Ci sono n oggetti: l'oggetto i fornisce il valore vi > 0 e pesa wi > 0.**
* **Il valore di un sottoinsieme di oggetti = somma dei valori dei singoli oggetti.**
* **Lo zaino ha un limite di peso pari a W**

**Ipotesi. Tutti i valori e i pesi sono integrali.**

****

**Es. Il sottoinsieme { 1, 2, 5 } ha valore 35 $ (e peso 10).**

**Es. Il sottoinsieme { 3, 4 } ha valore 40 $ (e peso 11).**

**Programmazione dinamica: falsa partenza**

**Definizione. OPT(i) = valore ottimale del problema knapsack con gli elementi 1,..., i.**

**Obiettivo. OPT(n).**

**Caso 1. OPT(i) non seleziona l'elemento i.**

* **OPT seleziona il migliore tra { 1, 2, ..., i - 1 }.**

**Caso 2. OPT(i) seleziona l'elemento i.**

* **La selezione dell'elemento i non implica immediatamente lo scarto di altri elementi.**
* **Senza sapere quali altri elementi sono stati selezionati prima di i, non sappiamo nemmeno se abbiamo abbastanza spazio per i.**

**Conclusione. Servono altri sottoproblemi!**

**Programmazione dinamica: due variabili**

**Definizione. OPT(i, w) = valore ottimale del problema dello zaino con gli elementi 1, ..., i, soggetto al peso limite w.**

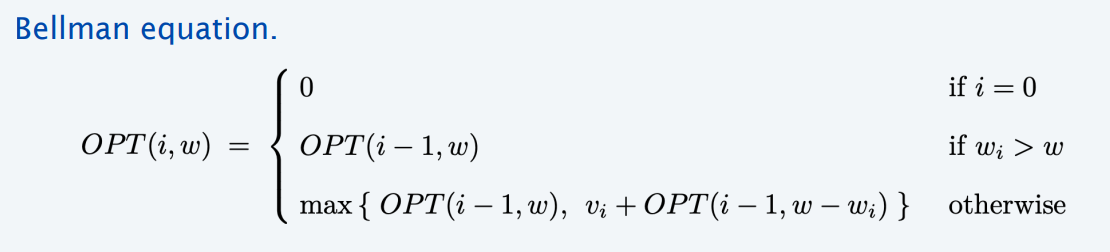
**Obiettivo. OPT(n, W).**

**Caso 1. OPT(i, w) non seleziona l'elemento i.**

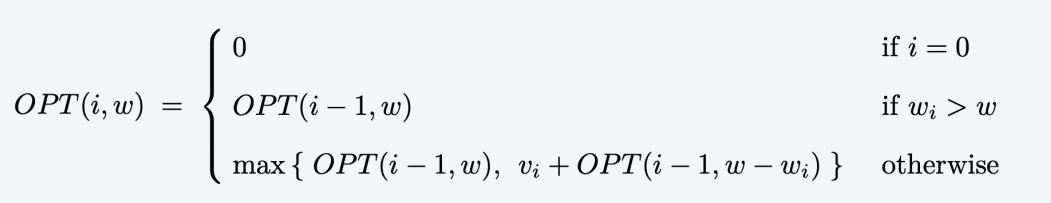
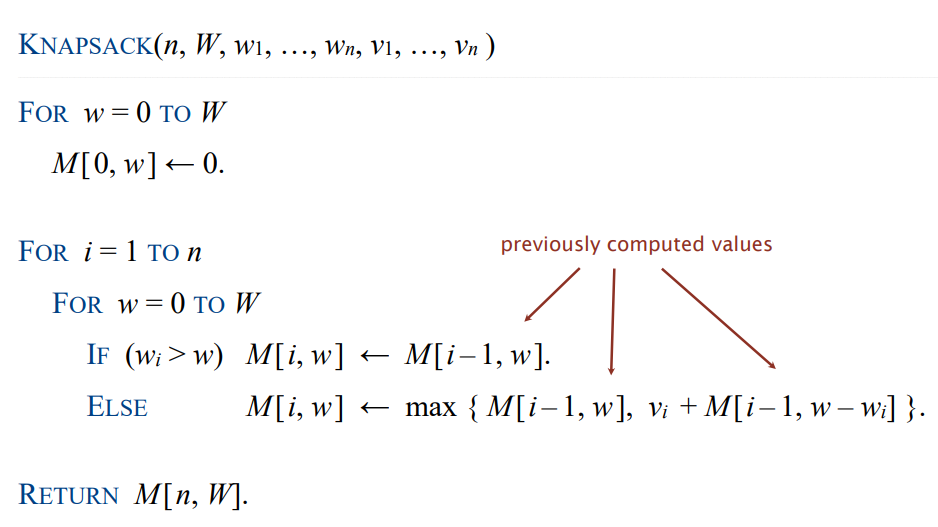
* **OPT(i, w) seleziona il migliore di { 1, 2, ..., i - 1 } soggetto al limite di peso w.**

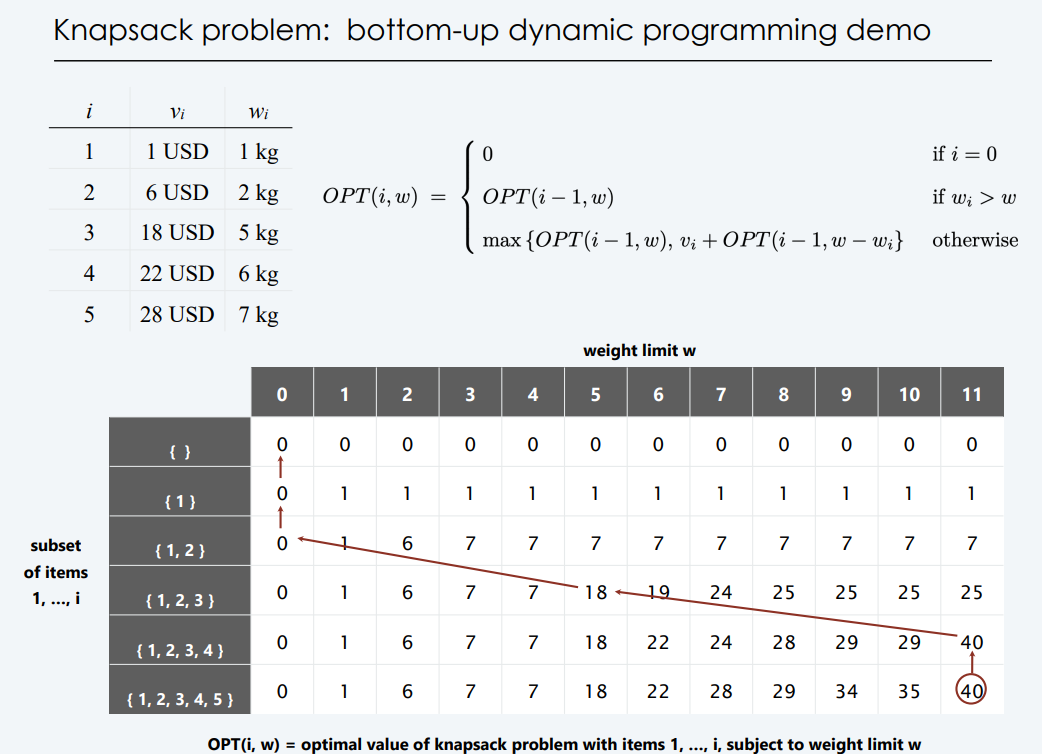
**Caso 2. OPT(i, w) seleziona l'elemento i.**

* **Raccoglie il valore vi.**
* **Nuovo limite di peso = w - wi.**
* **OPT(i, w) seleziona il migliore tra { 1, 2, ..., i - 1 } soggetto al nuovo limite di peso.**

****

**Problema Knapsack: programmazione dinamica bottom-up**

****

****

**Problema Knapsack: Running time.**

**Teorema. L'algoritmo DP risolve il problema di Knapsack con n elementi e peso massimo W in tempo Θ(n W) e spazio Θ(n W).**

**Pf.**

* **Richiede O(1) tempo per ogni voce della tabella.**
* **Ci sono Θ(n W) voci di tabella.**
* **Dopo aver calcolato i valori ottimali, è possibile risalire alla soluzione: OPT(i, w) prende l'elemento iff M [i, w] > M [i - 1, w]. ▪**

**Osservazioni.**

* **L'algoritmo dipende in modo critico dall'assunzione che i pesi siano integrali.**
* **L'ipotesi che i valori siano integrali non è stata utilizzata.**

**Il tempo di esecuzione dell'algoritmo DP per il problema knapsack è polinomiale?**

**- No, perché Θ(n W) non è una funzione polinomiale della dimensione dell'input.**

**- È pseudopolinomiale.**

**Algoritmo pseudopolinomiale: algoritmo il cui tempo di esecuzione è polinomiale per i valori dell'input (ad esempio, il più grande numero intero presente nell'input).**

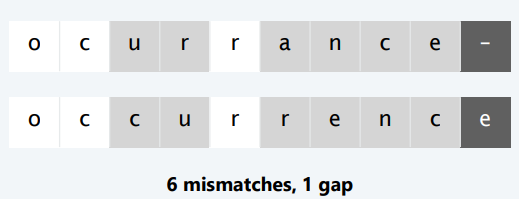
**- efficiente quando i numeri coinvolti nell'input sono ragionevolmente piccoli (ad esempio, nel problema del knapsack quando wi sono piccoli)**

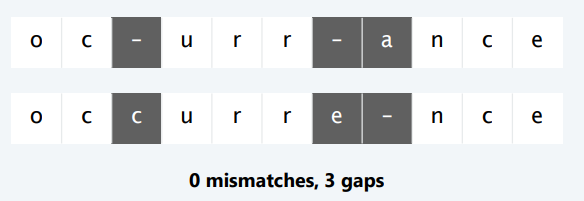
**- non è necessario che sia polinomiale nella dimensione dell'input (numero di bit richiesti per rappresentare l'input).**

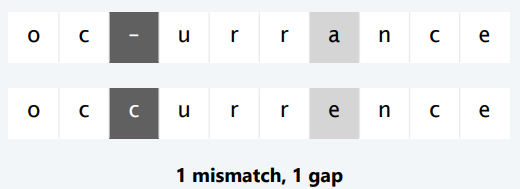
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**‣ sequence alignment**

**Utilizziamo un semplice esempio per illustrare il concetto di allineamento di sequenze. Consideriamo le stringhe "ocurrance" e "occurrence".**

****

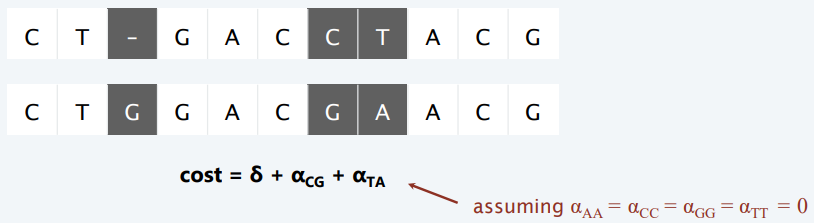
****

****

***Edit distance***

**Edit distance. [Levenshtein 1966, Needleman–Wunsch 1970]**

* **Penalità di gap δ; penalità di mismatch αpq.**
* **Costo = somma delle penalità di gap e mismatch.**

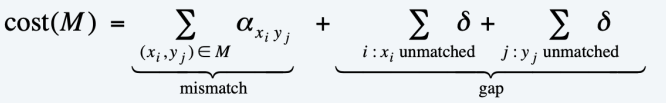
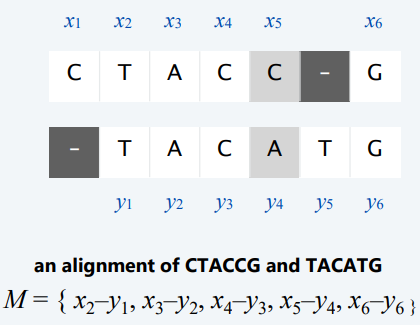
****

**sequence alignment .**

**Obiettivo. Date due stringhe x1 x2 ... xm e y1 y2 ... yn , trovare un allineamento a costo minimo.**

**Definizione. Un allineamento M è un insieme di coppie ordinate xi - yj tali che ogni carattere appaia al massimo in una coppia e non ci siano incroci.**

**xi - yj e xi′ - yj′ si incrociano se i < i ′ ma j > j ′**

**Def. Il costo di un allineamento M è:  
  
Esempio:  
 **

**Sequence alignment: problem structure:**

**Definizione. OPT(i, j) = costo minimo dell'allineamento delle stringhe di prefisso x1 x2 ... xi e y1 y2 ... yj.**

**Obiettivo. OPT(m, n).**

**Caso 1. OPT(i, j) corrisponde a xi - yj.**

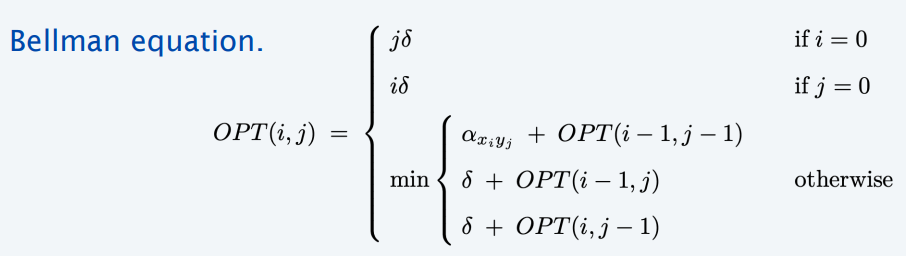
**Paga il disallineamento per xi - yj + il costo minimo dell'allineamento di x1 x2 ... xi-1 e y1 y2 ... yj-1.**

**Caso 2a. OPT(i, j) lascia xi non abbinato.**

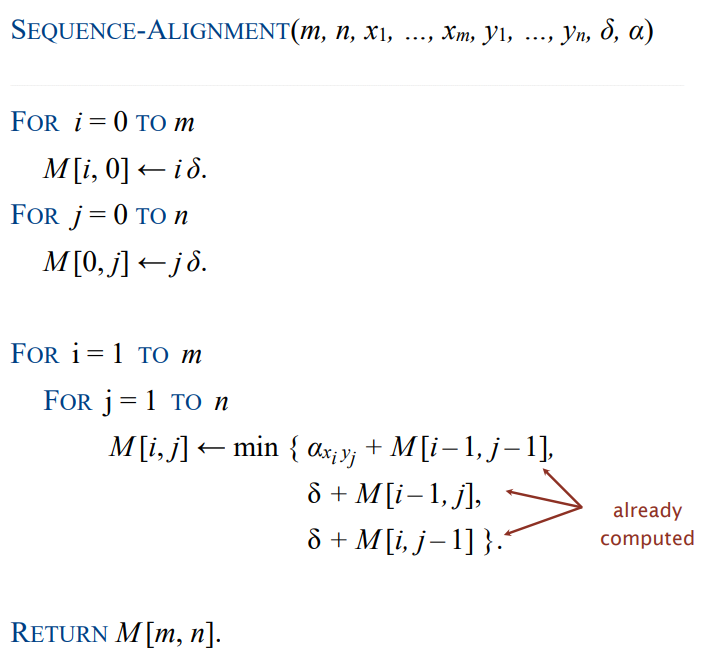
**Pagare il gap per xi + il costo minimo dell'allineamento di x1 x2 ... xi-1 e y1 y2 ... yj.**

**Caso 2b. OPT(i, j) lascia yj non abbinato.**

**Pay gap per yj + costo minimo dell'allineamento di x1 x2 ... xi e y1 y2 ... yj-1.**

****

**bottom-up algorithm**

****

**l'algoritmo memorizza i punteggi di allineamento di sottosequenze nella matrice M per evitare di ricalcolare gli stessi punteggi più volte.**

**Analisi:**

**Teorema. L'algoritmo DP calcola la distanza di modifica (e un allineamento ottimale) di due stringhe di lunghezza m e n in Θ(mn) tempo e spazio.**

**allineamento) di due stringhe di lunghezza m e n in Θ(mn) tempo e spazio.**

**Pf.**

* **L'algoritmo calcola la distanza di modifica.**
* **Può risalire all'estrazione dell'allineamento ottimale. ▪**

**‣ Hirschberg′s algorithm**

***Sequence alignment in linear space.***

**[Allineamento di sequenze nello spazio lineare]**

**Teorema. [Hirschberg] Esiste un algoritmo per trovare un allineamento ottimale in O(mn) tempo e O(m + n) spazio.**

**Combinazione geniale di divide et impera e programmazione dinamica.**

**Per calcolare la colonna/riga successiva della matrice abbiamo bisogno solo della colonna/riga precedente**

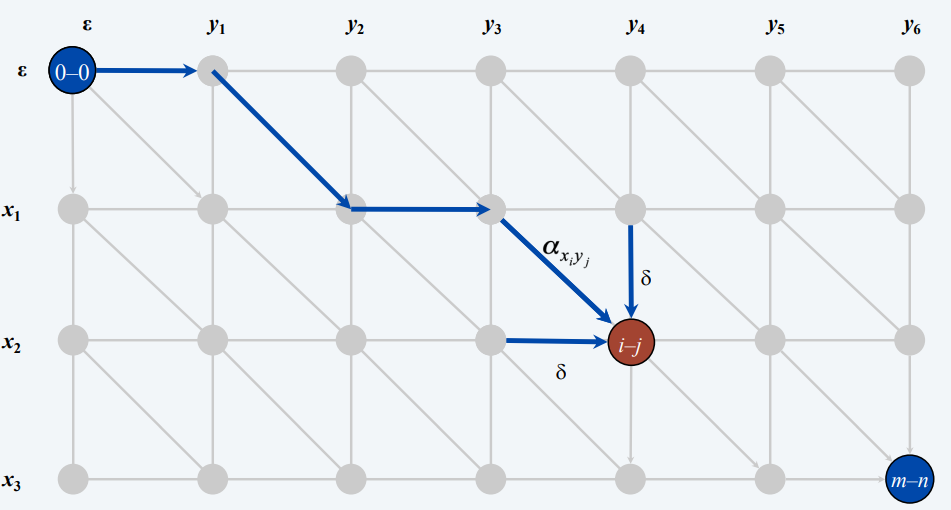
* **mantenere solo 2 colonne/riga alla volta**
* **Spazio O(m+n)**

****

**Edit distance graph.**

**・L'espressione f (i, j) indica la lunghezza del percorso più breve da (0,0) a (i,j).**

**・Lemma: f(i, j) = OPT(i, j) per tutti gli i e j.**

****

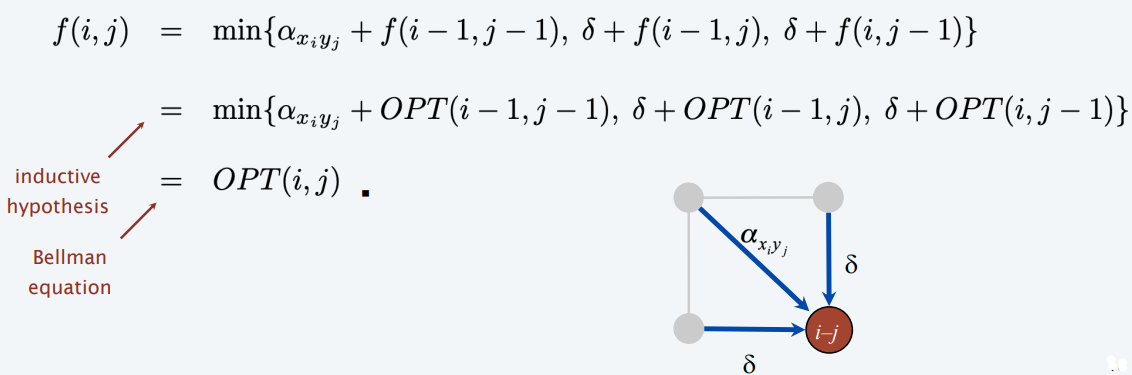
**Pf del Lemma. [ per induzione forte su i + j ]**

**Caso base: f(0, 0) = OPT (0, 0) = 0.**

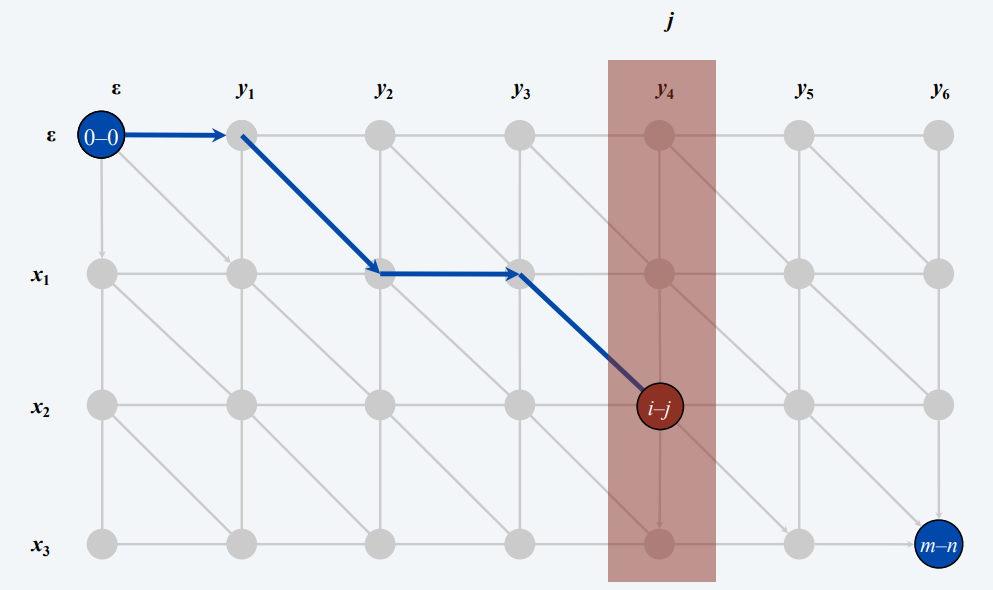
**・Ipotesi induttiva: si assume vera per tutti i (i′, j′) con i′ + j′ < i + j.**

**・L'ultimo bordo sul percorso più breve per (i, j) è da (i - 1, j - 1), (i - 1, j) o (i, j - 1).**

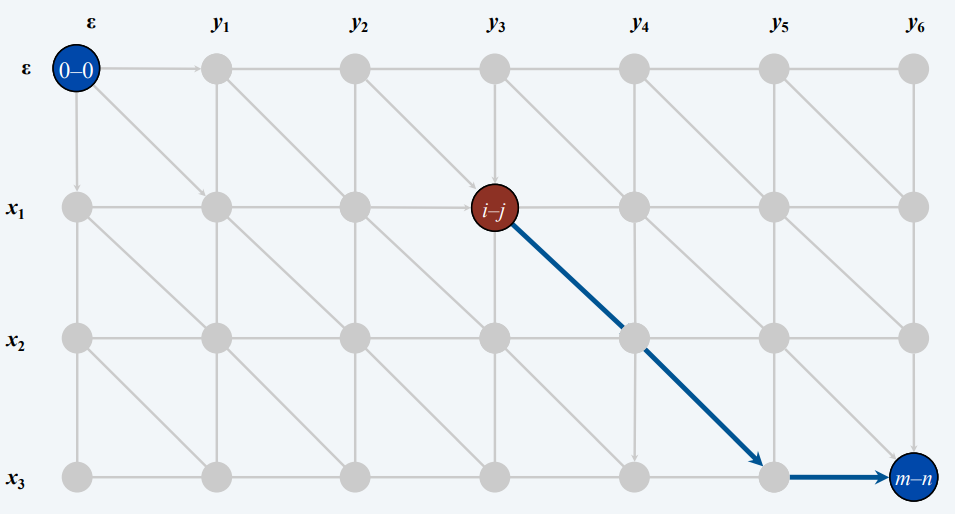
**・Quindi,**

****

**・Può calcolare f (-, j) per qualsiasi j in O(mn) tempo e O(m + n) spazio.**

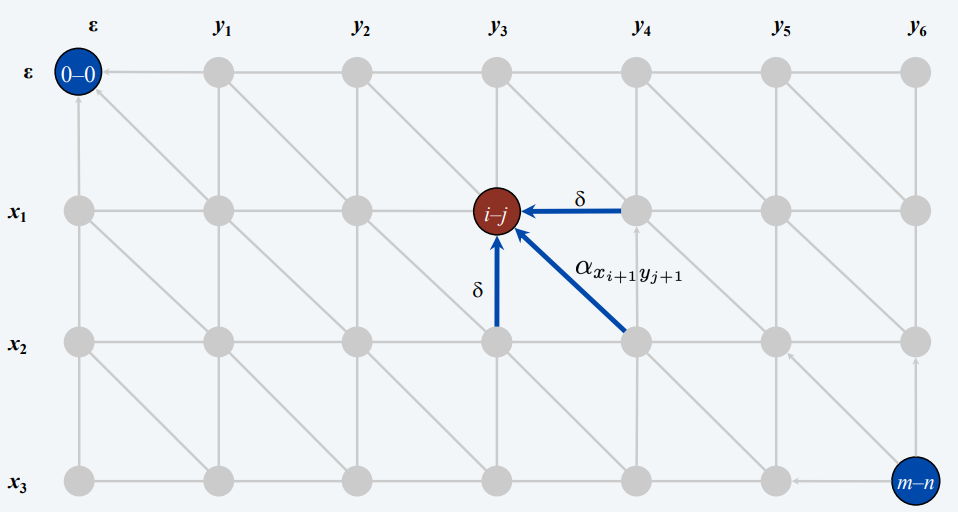
****

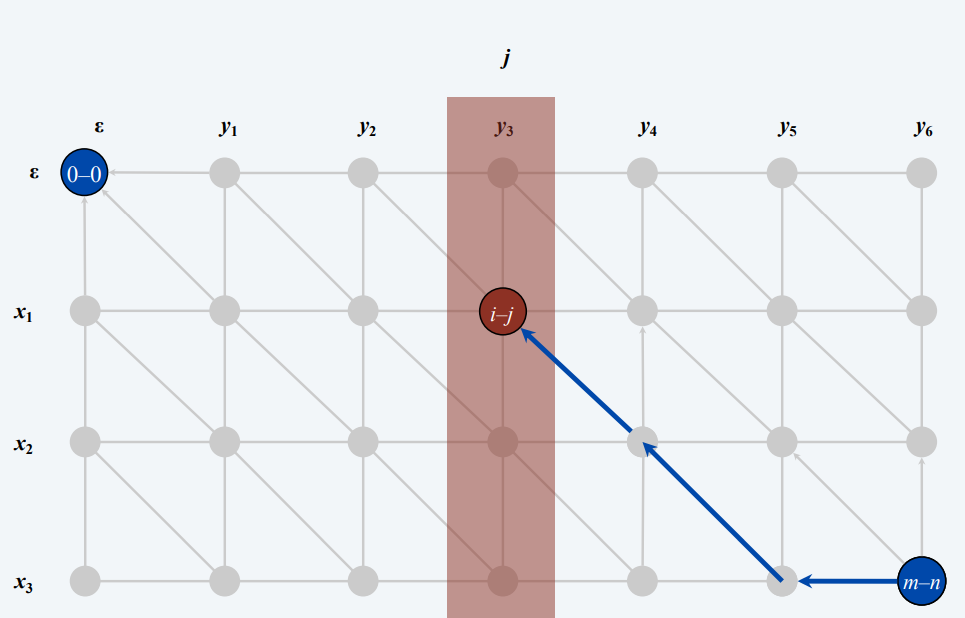
* **Sia g(i, j) la lunghezza del percorso più breve da (i, j) a (m, n).**

****

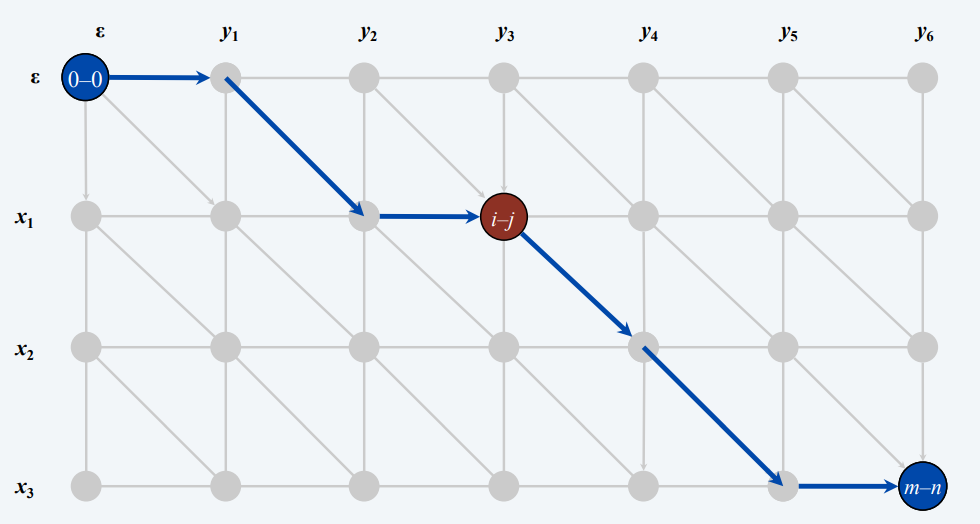
**・L'espressione g(i, j) indica la lunghezza del percorso più breve da (i, j) a (m, n).**

**・Può calcolare g(i, j) invertendo gli orientamenti degli spigoli e invertendo i ruoli di (0, 0) e (m, n).**

****

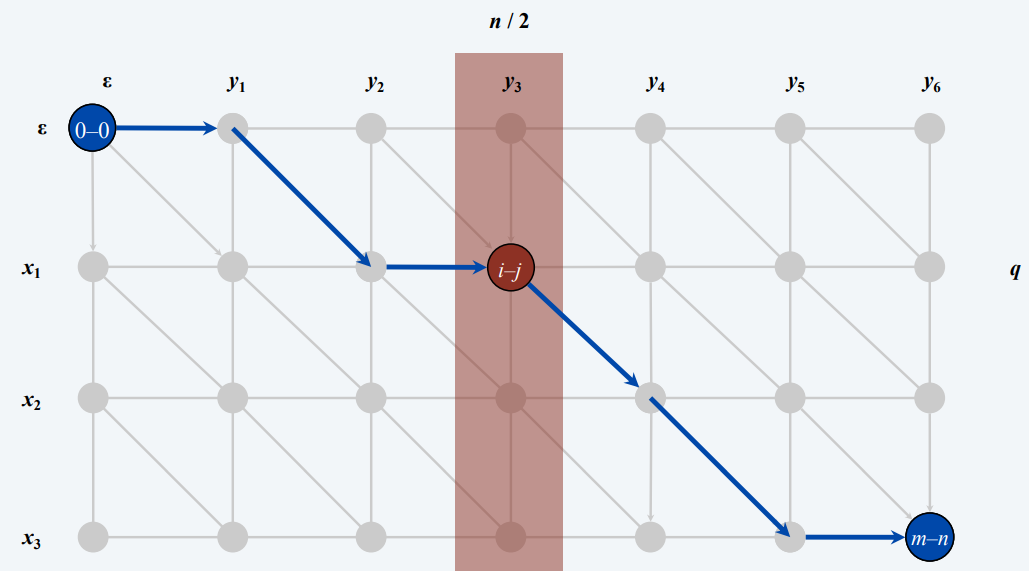
****

**Osservazione 1. La lunghezza di un percorso più breve che utilizza (i, j) è f (i, j) + g(i, j).**

****

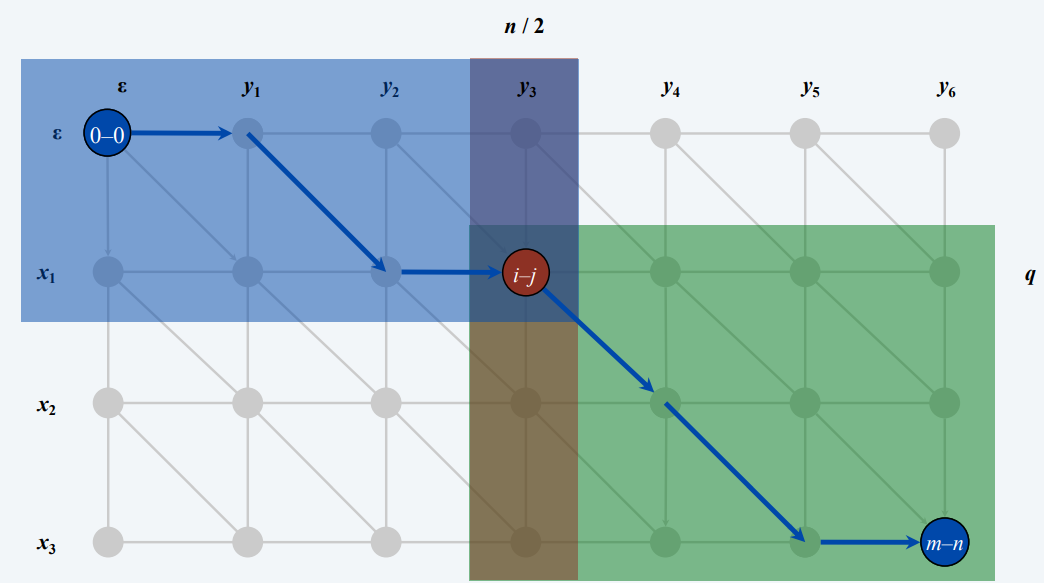
**Osservazione 2. Sia q un indice che minimizza f(q, n / 2) + g(q, n / 2).**

**Allora esiste un percorso più breve da (0, 0) a (m, n) che utilizza (q, n / 2).**

****

**Dividere. Trovare l'indice q che minimizza f (q, n / 2) + g(q, n / 2); salvare il nodo i-j come parte della soluzione.**

**Conquistare. Calcolo ricorsivo dell'allineamento ottimale in ogni pezzo.**

****

***space analysis***

**Teorema. L'algoritmo di Hirschberg utilizza lo spazio Θ(m + n).**

**Pf.**

* **Ogni chiamata ricorsiva utilizza lo spazio Θ(m) per calcolare f (-, n / 2) e g(-, n / 2).**
* **Per ogni chiamata ricorsiva è necessario mantenere solo Θ(1) spazio.**
* **Numero di chiamate ricorsive ≤ n. ▪**

***running time analysis (analisi dei tempi di esecuzione)***

**Teorema. Sia T(m, n) = tempo massimo di esecuzione dell'algoritmo di Hirschberg su stringhe di lunghezza massima m e n.**

**Allora T(m, n) = O(m n).**

**Pf. [ per induzione forte su m + n ]**

**・O(m n) tempo per calcolare f( -, n / 2) e g ( -, n / 2) e trovare l'indice q.**

**Tempo :**

**・T(q, n / 2) + T(m - q, n / 2) per due chiamate ricorsive.**

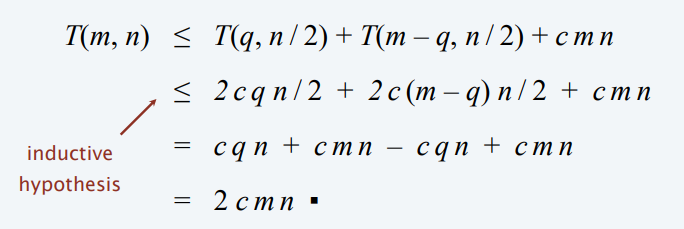
**・Scegliere la costante c in modo che:**

1. **T(m, 2) ≤ c m**
2. **2) T(2, n) ≤ c n**
3. **T(m, n) ≤ c m n + T(q, n / 2) + T(m – q, n / 2)**

**・Claim. T(m, n) ≤ 2 c m n.**

**・Caso base: m = 2 e n = 2.**

**・Ipotesi induttiva: T(m' , n') ≤ 2 c m' n' for all (m′, n′) with m′ + n′ < m + n.**

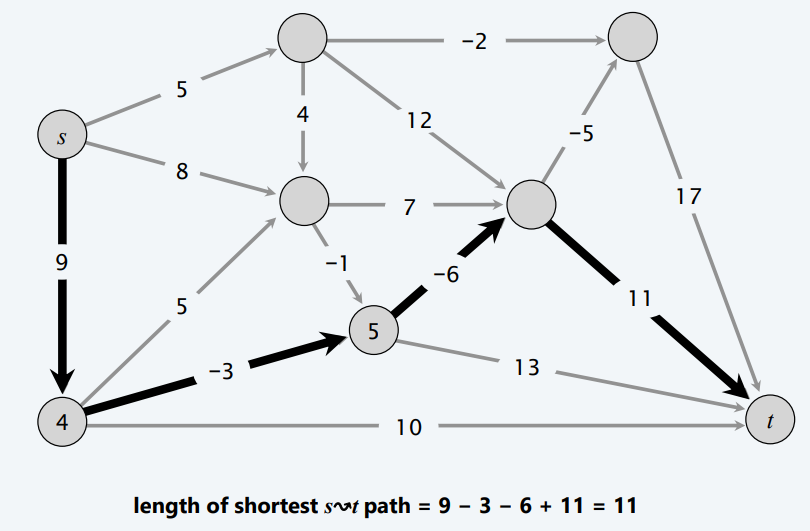
****

**‣ Bellman–Ford–Moore algorithm**

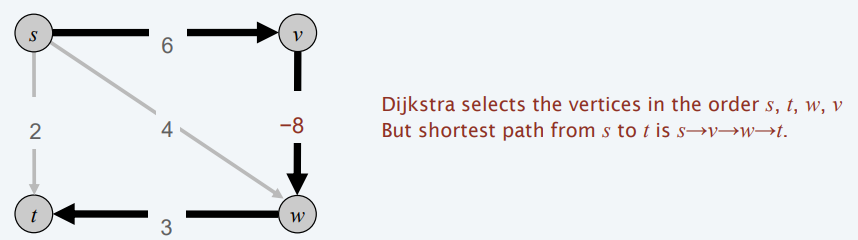
**[Shortest paths with negative weights]**

**Problema del percorso più breve.**

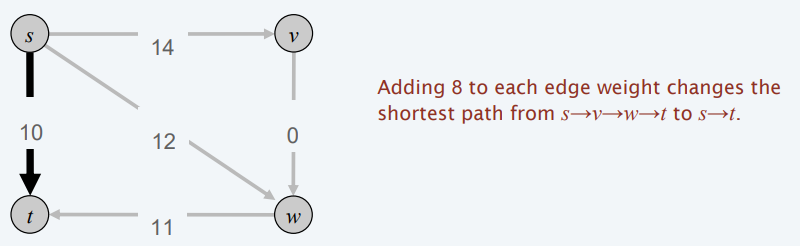
**Dato un digrafo G = (V, E), con bordi di lunghezza arbitraria ℓvw, trovare il percorso più breve dal nodo di origine s al nodo di destinazione t.**

****

**Dijkstra. Potrebbe non produrre percorsi più brevi quando le lunghezze dei bordi sono negative.**

****

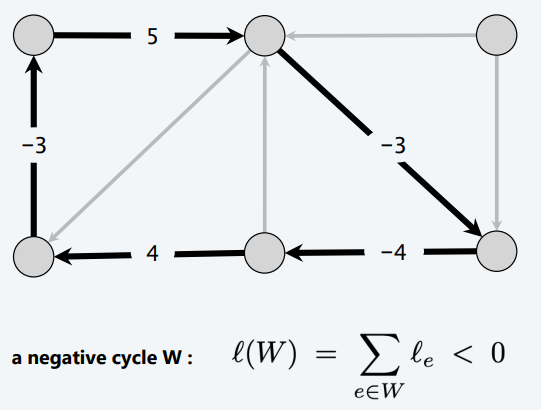
**Riponderazione. L'aggiunta di una costante alla lunghezza di ogni bordo non fa necessariamente sì che l'algoritmo di Dijkstra produca percorsi più brevi.**

****

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Negative cycles**

**Definizione. Un ciclo negativo è un ciclo diretto la cui somma delle lunghezze dei bordi è negativa.**

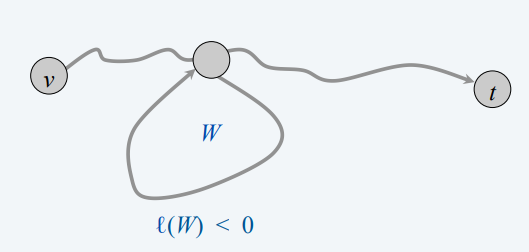
****

**Shortest paths and negative cycles**

**Lemma 1. Se un percorso v↝t contiene un ciclo negativo, allora non esiste un percorso v↝t più breve.**

**Pf.**

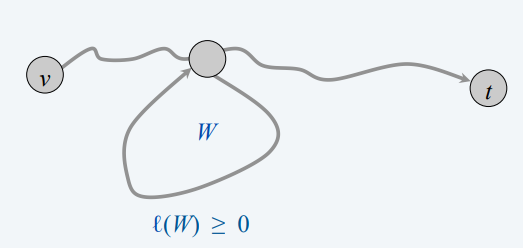
**Se esiste un ciclo W, è possibile costruire un percorso v↝t di lunghezza arbitrariamente negativa deviando attorno a W tutte le volte che si vuole. ▪**

****

**Lemma 2. Se G non ha cicli negativi, allora esiste un percorso v↝t più breve che è semplice (e ha ≤ n - 1 spigoli).**

**Pf.**

* **Tra tutti i percorsi v↝t più brevi, si consideri quello che utilizza il minor numero di spigoli.**
* **Se il percorso P contiene un ciclo diretto W, è possibile rimuovere la parte di P corrispondente a W senza aumentarne la lunghezza. ▪**

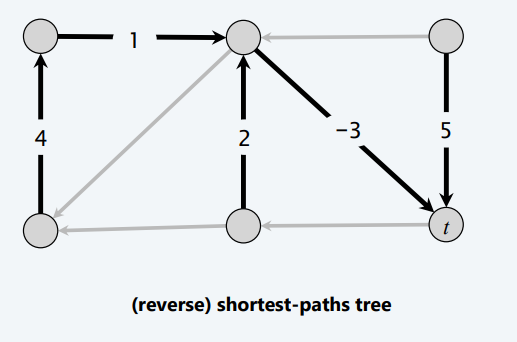
****

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

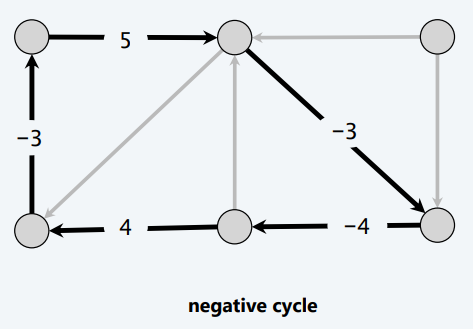
**Shortest-paths and negative-cycle problems**

**Problema dei percorsi più brevi a destinazione singola.**

**Dato un digrafo G = (V, E) con bordi di lunghezza ℓvw (ma senza cicli negativi) e un nodo distinto t, trovare un percorso v↝t più breve per ogni nodo v.**

****

**Problema dei cicli negativi. Dato un digrafo G = (V, E) con bordi di lunghezza ℓvw, trovare un ciclo negativo (se esiste).**

****

**Dynamic programming**

**Definizione. OPT(i, v) = lunghezza del percorso più breve v↝t che utilizza ≤ i bordi.**

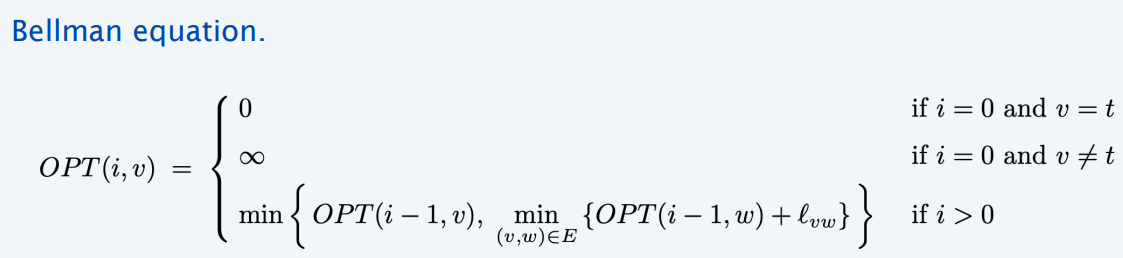
**Obiettivo. OPT(n - 1, v) per ogni v**

**Caso 1. Il percorso più breve v↝t utilizza ≤ i - 1 bordi.**

* **OPT(i, v) = OPT(i - 1, v).**

**Caso 2. Il percorso v↝t più breve utilizza esattamente i bordi.**

* **Se (v, w) è il primo spigolo del percorso più breve v↝t, il costo da sostenere è ℓvw.**
* **Quindi, selezionare il miglior percorso w↝t utilizzando ≤ i - 1 bordi.**

****

***implementation***

****

**Teorema 1. Dato un digrafo G = (V, E) senza cicli negativi, l'algoritmo DP calcola la lunghezza del percorso più breve v↝t per ogni nodo v in Θ(mn) tempo e Θ() spazio.**

**Pf.**

* **La tabella richiede Θ(n^2) spazio.**
* **Ogni iterazione i richiede Θ(m) tempo poiché esaminiamo ogni bordo una volta. ▪**

**Trovare i percorsi più brevi.**

**・Approccio 1: mantenere il successore[i, v] che punta al nodo successivo su un percorso più breve v↝t utilizzando ≤ i bordi.**

**・Approccio 2: Calcolare le lunghezze ottimali M[i, v] e considerare solo gli spigoli con M[i, v] = M[i - 1, w] + ℓvw. Qualsiasi percorso diretto in questo sottografo è un percorso più breve.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**miglioramenti pratici**

**Ottimizzazione dello spazio. Mantenere due matrici 1D (invece di una matrice 2D).**

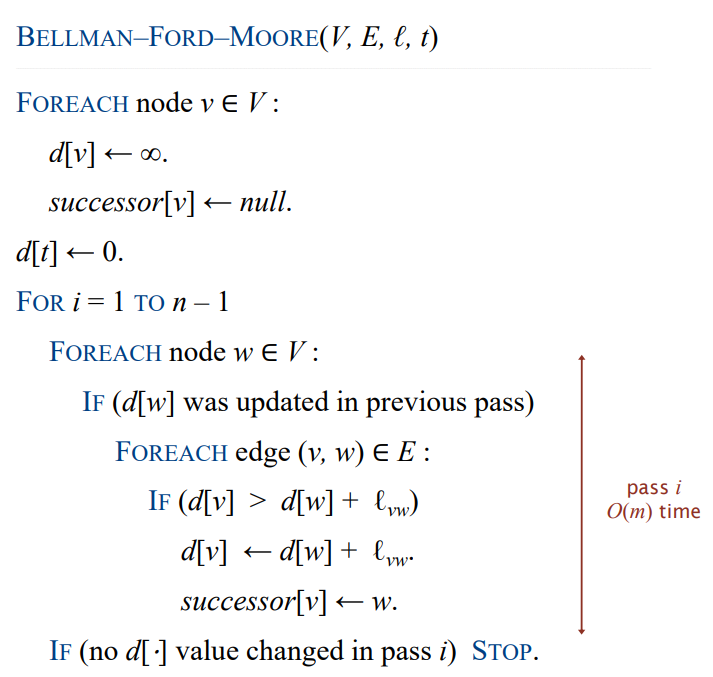
**・d[v] = lunghezza del percorso più breve v↝t trovato finora.**

**・successor[v] = nodo successivo in un percorso v↝t.**

**Ottimizzazione delle prestazioni. Se d[w] non è stato aggiornato nell'iterazione i - 1, non c'è motivo di considerare gli spigoli che entrano in w nell'iterazione i.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**efficient implementation**

****

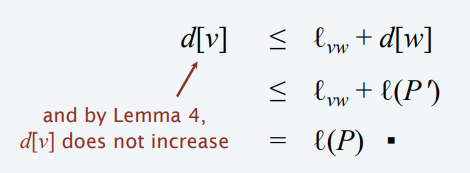
**Lemma 3. Per ogni nodo v : d[v] è la lunghezza di un percorso v↝t.**

**Lemma 4. Per ogni nodo v : d[v] è monotono non crescente.**

**Lemma 5. Dopo il passaggio i, d[v] ≤ lunghezza di un percorso v↝t più breve che utilizza ≤ i bordi.**

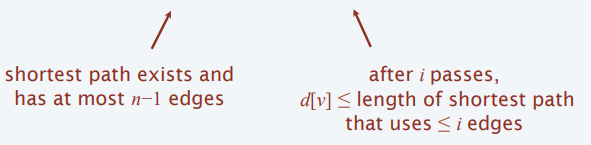
**Pf. [ per induzione su i ]**

* **Caso base: i = 0.**
* **Assumiamo vero dopo il passaggio i.**
* **P sia un qualsiasi percorso v↝t con ≤ i + 1 spigoli.**
* **Che (v, w) sia il primo bordo in P e che P′ sia il sottopercorso da w a t.**
* **Per ipotesi induttiva, alla fine del passaggio i, d[w] ≤ ℓ(P′) perché P′ è un cammino w↝t con ≤ i bordi.**
* **Dopo aver considerato lo spigolo (v, w) nel passaggio i + 1:**

****

**Teorema 2. Assumendo che non ci siano cicli negativi, Bellman-Ford-Moore calcola le lunghezze dei percorsi più brevi v↝t in O(mn) tempo e Θ(n) spazio extra.**

**Pf. Lemma 2 + Lemma 5. ▪**

****

**Osservazione. Bellman-Ford-Moore è tipicamente più veloce nella pratica.**

* **Edge (v, w) considerato nel passaggio i + 1 solo se d[w] viene aggiornato nel passaggio i.**
* **Se il percorso più breve ha k edges, l'algoritmo lo trova dopo ≤ k passaggi.**

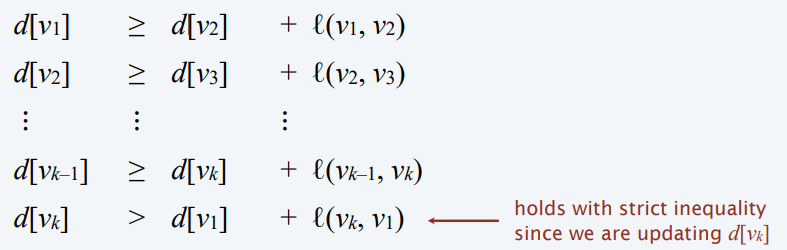
**Lemma 6. Ogni ciclo diretto W nel grafo successore è un ciclo negativo.**

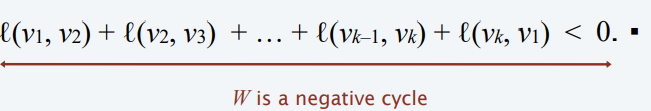
**Pf.**

**・Se successore[v] = w, dobbiamo avere d[v] ≥ d[w] + ℓvw. (LHS e RHS sono uguali quando successore[v] è impostato; d[w] può solo diminuire; d[v] diminuisce solo quando successore[v] è azzerato)**

**・Lasciare che v1 → v2 → ... → vk → v1 sia la sequenza di nodi in un ciclo diretto W.**

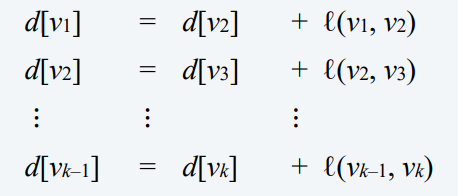
**・Assumiamo che (vk, v1) sia l'ultimo bordo in W aggiunto al grafo successore.**

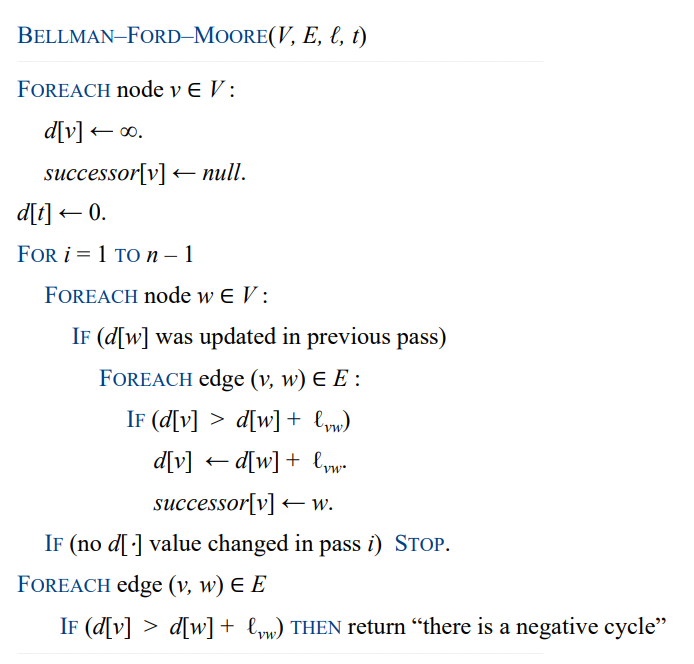
**・Poco prima di questo:**

**Aggiungendo le disuguaglianze si ottiene :**

**Teorema 3. Assumendo che non ci siano cicli negativi, Bellman-Ford-Moore trova i percorsi più brevi v↝t per ogni nodo v in O(mn) tempo e Θ(n) spazio extra.**

**Pf.**

* **Il grafo successore non può avere un ciclo diretto. [Lemma 6]**
* **Quindi, seguendo i puntatori dei successori da v si ottiene un percorso diretto verso t.**
* **Lasciate che v = v1 → v2→ ... → vk = t siano i nodi lungo questo percorso P.**
* **Al termine, se successore[v] = w, si deve avere d[v] = d[w] + ℓvw. (LHS e RHS sono uguali quando successore[v] è impostato; d[-] non è cambiato).**
* **Quindi,**
* **Aggiungendo le disuguaglianze si ottiene :**

**Bellman–Ford–Moore: checking for negative cycle**

**Lemma 6. Se esiste un ciclo negativo (che può raggiungere t) l'algoritmo (modificato) lo segnala.**

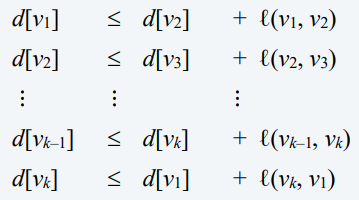
**Pf.**

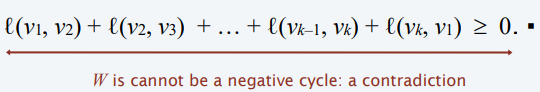
**・Se non c'è un ciclo negativo, il passaggio #n non fa nulla.**

**・Lasciare che v1 → v2 → ... → vk → v1 un ciclo negativo diretto W.**

**・assumiamo per contraddizione che l'algoritmo non lo restituisca**

**・Allora: la condizione dell'ultimo IF è sempre falsa.**

**Quindi:**

**・Aggiungendo le disuguaglianze si ottiene**