1 Programmazione Dinamica

La programmazione dinamica è una tecnica che si ispira al *divide-et-impera* e infatti anche qui un problema più grande viene preso e diviso in più sottoproblemi, ma si aggiunge una caratteristica. Normalmente nel *divide-et-impera* i problemi sono ben separati, mentre nella programmazione dinamica i sottoproblemi si ripetono. Per questo motivo si utilizza una tabella delle soluzioni, cioè si memorizzano le soluzioni dei problemi che sono già stati risolti.
Per fare una cosa del genere si deve capire come definire il sottoproblema e che informazioni memorizzare, dato che non si ha una memoria infinita e non si può memorizzare tutto, la memorizzazione deve essere efficiente. Nel caso in cui un sottoproblema debba nuovamente essere affrontato, si ottiene la sua soluzione dalla tabella, la quale è facilmente indirizzabile (il costo della procedura di lookup è $O(1)$).

1.1 Idea generale

Se non si sa con esattezza quale problema risolvere, si può provare a risolverli tutti e conservare i risultati ottenuti per poterli usare successivamente.
Rispetto al *divide-et-impera*, la programmazione dinamica presenta 3 differenze:

- È iterativa e non ricorsiva;
- Affronta i sottoproblemi dal "basso verso l'alto" (bottom-up) e non "dall'ato verso il basso" (top-down);
- Memorizza i risultati dei sottoproblemi in una tabella DP.

Procedendo dal basso verso l'alto, la programmazione dinamica risolve un sottoproblema comune una sola volta, mentre il *divide-et-impera* lo risolverebbe più volte.

1.1.1 Fasi della risoluzione del problema

Fasi principali

- Caratterizzare la struttura di una soluzione ottima;
- Definire iterativamente il valore di una soluzione ottima;
- Calcolare il **valore** di una soluzione ottima "bottom-up";
- Ricostruzione di una soluzione ottima.

1.2 Esempio - Coefficiente binomiale

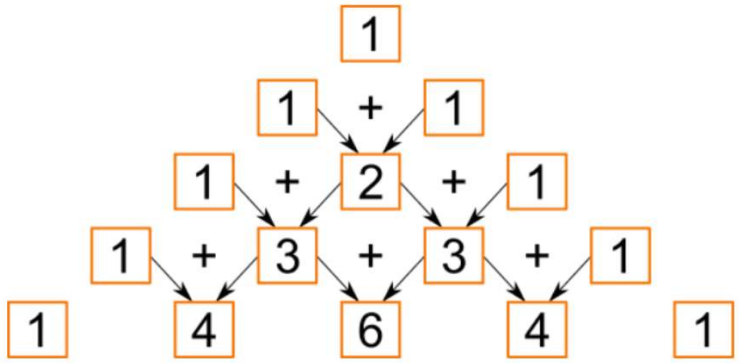
$C(n,k) = \frac{n!}{k!(n-k)!}$ rappresenta il numero di modi di scegliere k oggetti da un insieme di n oggetti, con $0 \leq k \leq n$, ed è definibile ricprsivamente come segue:

$$C(n,k) = \begin{cases} 1 \quad se\ k = 0 \ \vee \ n = k \\ C(n-1,k-1) + C(n-1,k) \quad altrimenti \end{cases}$$

Purtroppo, la complessità dell'algoritmo *divide-et-impera* cresce come il numero di chiamate ricorsive, che è proprio uguale a $C(n,k)$. Ciò è dovuto all'elevato numero di sottoproblemi identici che vengono risolti più volte. Ad esempio $C(n-2,k-1)$ è richiamato due volte: per calcolare $C(n-1,k)$ e $C(n-1,k-1)$.

```
int C(int n, int k)
1: if n=k or k=0 then
2:     return 1
3: end if
4: return C(n-1,k-1)+C(n-1,k)
```

Un algoritmo di programmazione dinamica invece risolve i sottoproblemi per valori crescenti degli argomenti seguendo lo schema "**Triangolo di Tartaglia**", che memorizza i risultati una volta per tutte in una matrice DP di dimensione nxn in $\theta(n^2)$.




```
Tartaglia(int n, int[][] DP)
1: for i do=0 to n do
2:     DP[i, 0] = 1
3:     DP[i, i] = 1
4: end for
5: for i do=2 to n do
6:     for j do=1 to i-1 do
7:         DP[i, j]=DP[i-1, j-1]+DP[i-1, j]
8:     end for
9: end for
```

In questo, il valore di $DP(m, k)$, per ogni coppia di interi "m, k" tali che $0 \leq k \leq m \leq n$ può essere successivamente letto direttamente in tempo $O(1)$.

1.3 Quando usare la programmazione dinamica

Perché la programmazione dinamica sia applicabile, occorre che:

1. sia possibile combinare le soluzioni dei sottoproblemi per trovare la soluzione del problema più grande;
2. le decisioni prese per risolvere in modo ottimo un sottoproblema rimangano valide quando il problema diviene un pezzo di un problema;
3. una tecnica *divide-et-impera* sia inutilizzabile dal punto di vista computazionale.

Le prime due proprietà definiscono una **sottostruttura ottima**. Ma non sono sufficienti. Infatti, affinché un algoritmo basato sulla programmazione dinamica abbia complessità

polinomiale, occorre anche che:

4. ci sia un numero polinomiale di problemi da risolvere;
5. per evitare di risolvere più di una volta lo stesso problema, si utilizzi una tabella in cui si memorizzano le soluzioni di tutti i sottoproblemi, senza preoccuparsi se la soluzione di un particolare sottoproblema verrà poi utilizzata oppure no;
6. il tempo per combinare le soluzioni dei sottoproblemi e trovare la soluzione del problema più grande sia polinomiale.

1.4 String Match Approssimato

Date:
- una stringa $P = p_1 \dots p_m$, detta **pattern**,
- una stringa $T = t_1 \dots t_n$, detta **testo**, con $m \leq n$,

un'occorrenza $k - approssimata$ di P in T, con $0 \leq k \leq m$, è una copia della stringa di P in T in cui sono ammessi k "errori" tra caratteri di P e caratteri di T, del seguente tipo:
1. corrispondenti caratteri P, T sono diversi (**sostituzione**)
2. un carattere in P non è incluso in T (**inserimento**)
3. un carattere in T non è incluso in P (**cancellazione**)

Problema - Approximated string matching

trovare un'occorrenza $k - approssimata$ di P in T con k minimo ($0 \leq k \leq m$).

1.4.1 Esempio

T = questoèunoscempio (dove $n = 17$)
P = unesempio (dove $m = 9$).
Un'occorrenza $2 - approssimata$ di P parte dalla posizione 8 di T: questo*èunoscempio*. Infatti, la prima e di P corrisponde ad una o di T (errore di tipo (1), **sostituzione**), mentre la c di T è un carattere non presente in P (errore di tipo (3), **cancellazione**).
Per applicare la programmazione dinamica, si deve trovare una formulazione ricorsiva che abbia la proprietà di sottostruttura ottima.

Definizione

Sia $DP[0 \dots m][0 \dots n]$ una tabella di programmazione dinamica tale che $DP[i][j]$ sia il minimo valore k per cui esiste un'occorrenza $k - approssimata$ di $P(i)$ in $T(j)$ che **termina nella posizione** j

1.4.2 4 possibilità:

- Se $P[i] = T[j]$, quel carattere è uguale, ci si sta chiedendo qual è la più corta occorrenza k-approssimata che termina in quel carattere. Ciò che si va a fare è dire: se il carattere è uguale non c'è errore, quindi non si somma un qualcosa qua e si va a vedere il carattere precedente. Il punto è che ci si sta limitando a considerare le sequenze che terminano in quella posizione. $DP[i-1][j-1]+0$.

- Se $P[i] \neq T[j]$, allora si può fare una **sostituzione**, cioè si cambia un carattere nell'altro e quindi si somma 1: $DP[i-1][j-1]+1$.

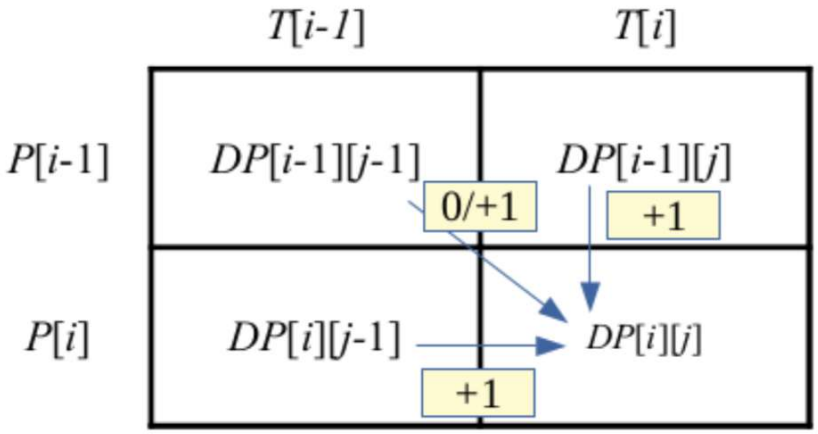
Queste due possibilità si escludono, perché non possono valere contemporaneamente, mentre le prossime due possono coesistere.

- $DP[i-1][j]+1$
- $DP[i][j-1]+1$

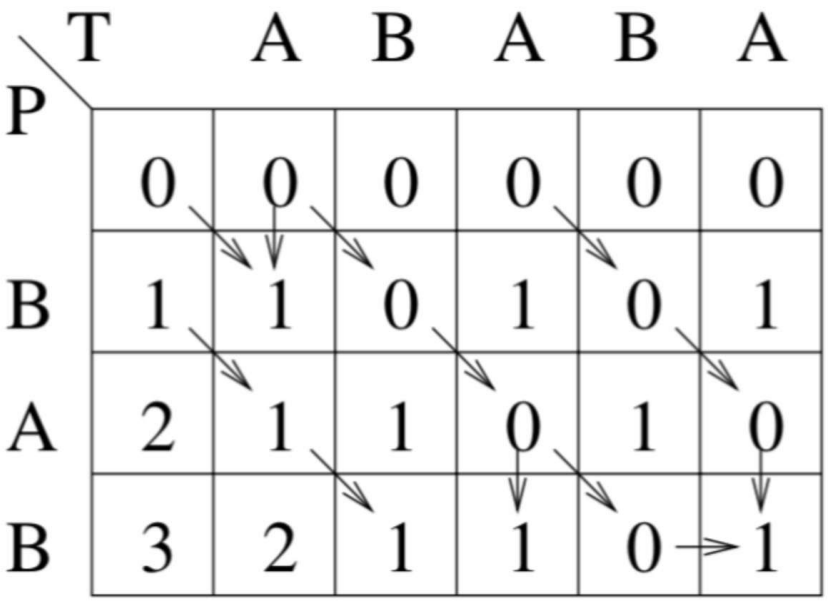
Si somma 1 perché si contano gli errori, non si sta massimizzando l'uguaglianza, ma si stanno minimizzando le differenze.

$$DP[i][j] = \begin{cases} 0 & i = 0 \\ i & j = 0 \\ \min\{DP[i-1][j-1] + \delta, & \delta = \text{iif}(P[i] = T[j], 0, 1) \\ \quad DP[i-1][j] + 1, & \\ \quad DP[i][j-1] + 1\} & \text{altrimenti} \end{cases}$$

Quello che succede è che la formula avrà la forma evidenziata da *, cioè il minimo di *1 dove δ è uguale a 0 o 1 a seconda che i caratteri siano uguali oppure diversi e poi *2 e *3, questo è il caso ricorsivo.

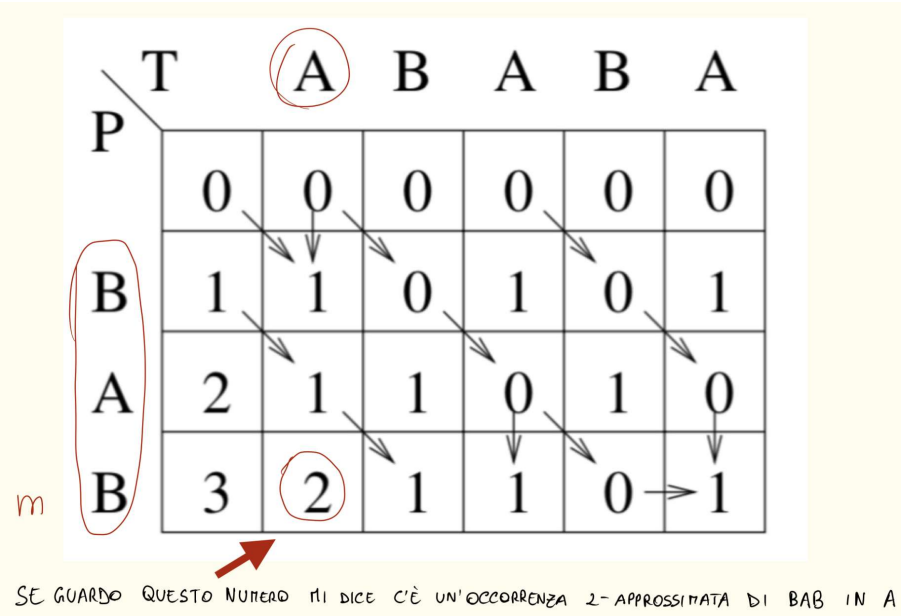


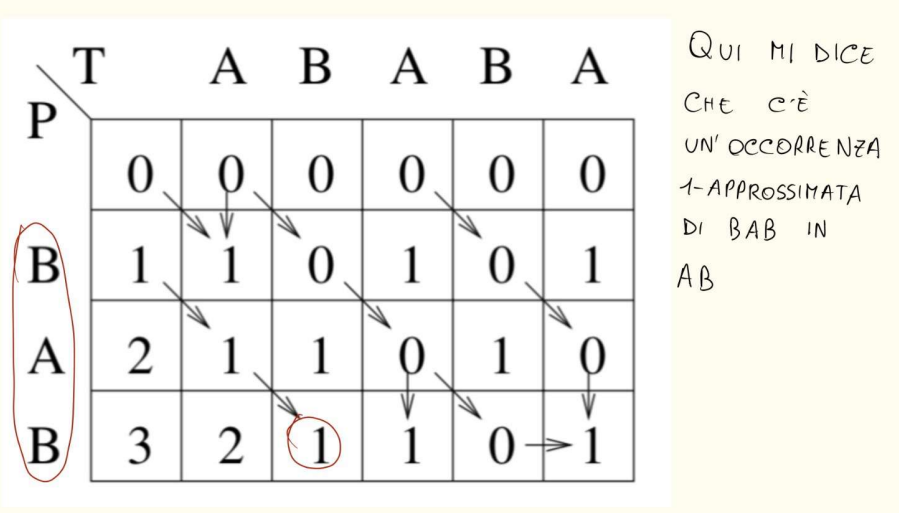
Nei primi due casi si nota la simmetria tra il testo e il pattern perché la i è per il pattern e la j per il testo. Se si cerca un pattern vuoto in un testo, qualunque sia la lunghezza del testo, il numero di cambiamenti che si devono fare è 0, perché in qualunque punto termina una stringa vuota di un testo. Invece se si cerca una parola in un testo vuoto, bisogna inserire tutti i caratteri di quella parola, quindi l'intera lunghezza del pattern, perciò si mette la lettera i.



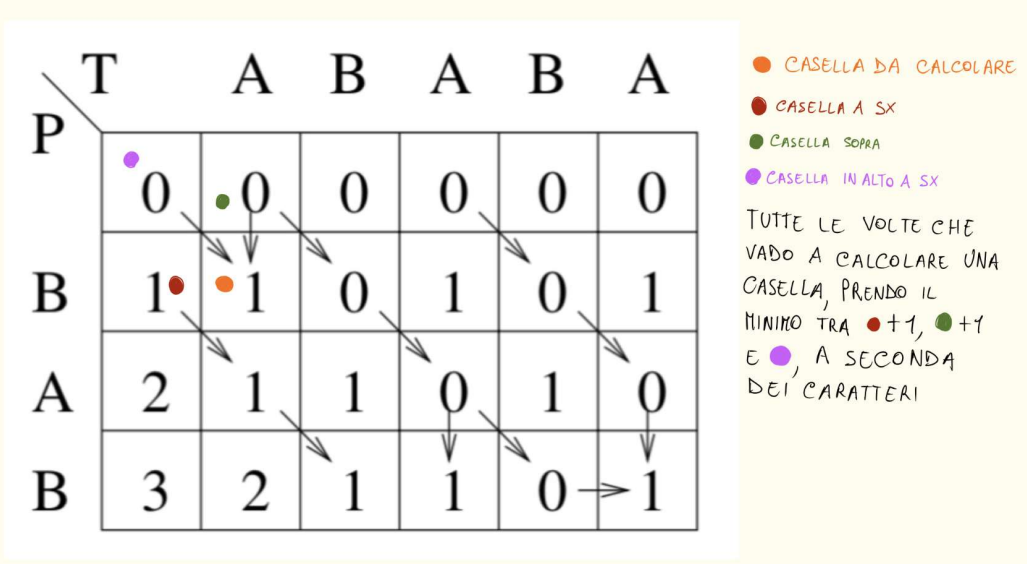
Le frecce indicano come prendere il "punteggio" da aggiungere.

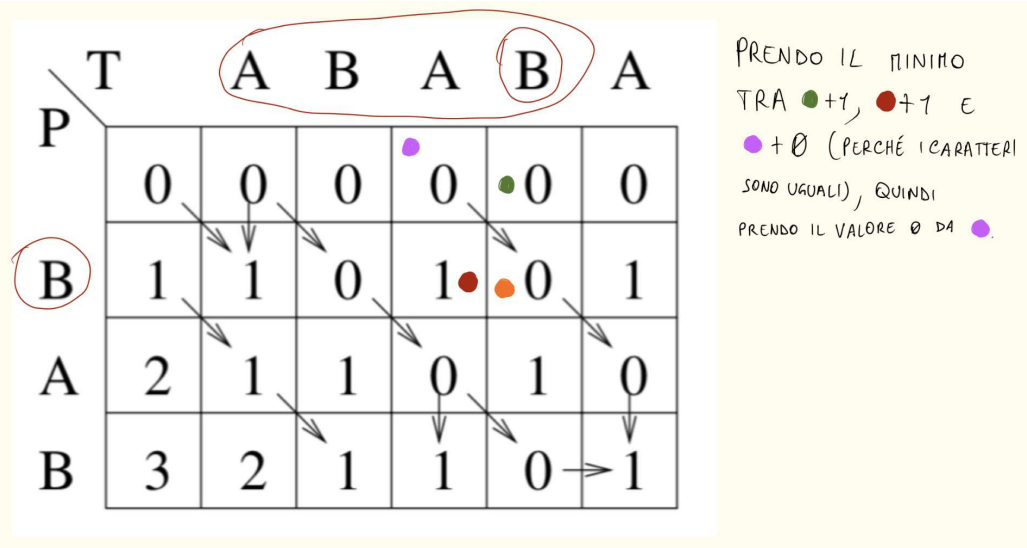
1.4.3 Esempio 1





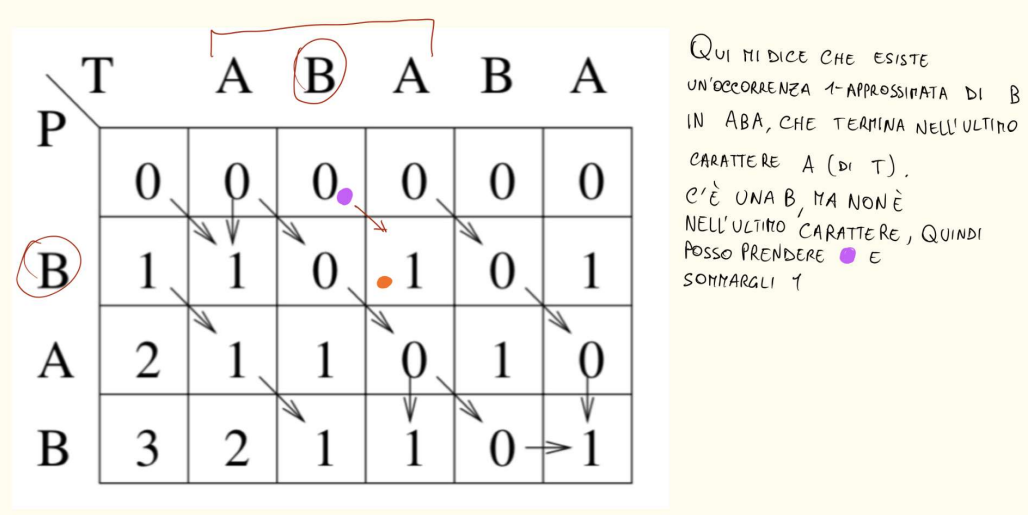
1.4.4 Esempio 2



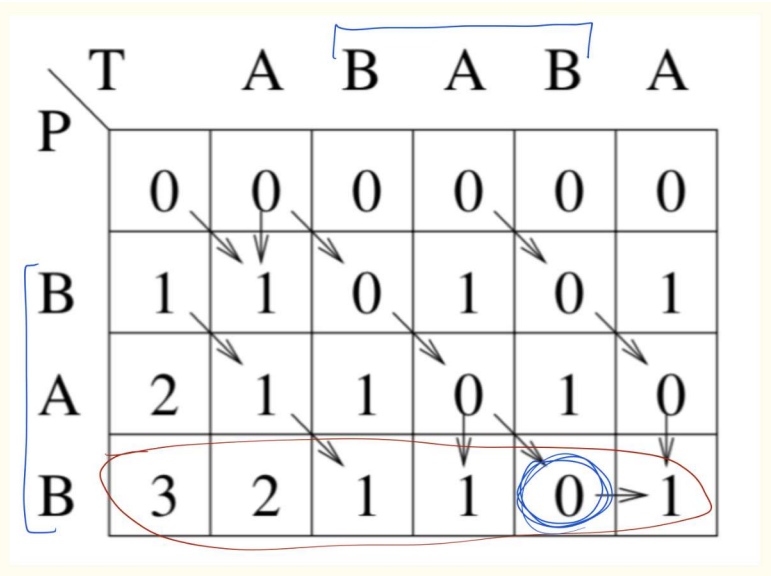


Se si va a vedere la semantica, questo mi sta dicendo che esiste un'occorrenza $0-approssimata$ di B in ABAB che termina nell'ultimo carattere. Se è $0-approssimata$ significa che è perfetta, perché in effetti la B del pattern ci sta perfettamente.

1.4.5 Esempio 3



1.4.6 Esempio 4



Queste sono sequenze $k - approssimate$ dell'intero pattern BAB in qualche parte del testo. Si deve restituire un'occorrenza della stringa BAB con k minore possibile, che qua si trova dove è lo 0.
Non è detto quindi che la soluzione finale si trovi nella casella in basso a destra. È possibile invece che la soluzione debba essere cercata (se esiste la mia occorrenza $k - approssimata$ e dov'è con k minore possibile).

Distanza di Levenshtein o distanza di editing

Date due stringhe, si vuole conoscere il numero minimo di operazioni necessario per trasformare una nell'altra (o viceversa). Es. la distanza di editing tra google e yahoo è 6 (cancellare gle e inserire ya, sostituire g con h). La relazione di ricorrenza è la stessa, ma $D[i, 0] = i$ e $D[0, j] = j$, perché $T(j)$ ha j caratteri in più di $P(0)$. Al termine, la distanza di editing sarà in $D[m, n]$.

1.4.7 Algoritmo

```
int stringMatching(ITEM[] P, ITEM[] T, int m, int n)
  int[][] DP=new int[0 ... m][0 ... n]
  for j do=0 to n do
      DP[0][j]=0
  end for
  for i do=0 to m do
      DP[i][0]=i
  end for
  for i do=1 to m do
      for j do=1 to n do
          DP[i][j]=min(DP[i-1][j-1]+iif(P[i]==T[j],0,1), DP[i-1][j]+1, DP[i][j-1]+1)
      end for
  end for
  int pos=0
  for j do=1 to n do
      if DP[m][j]<DP[m][pos] then
          pos=j
      end if
  end for
  return pos
```

Il primo `for` inizializza la prima riga con tutti gli zeri, il secondo `for` inizializza invece la prima colonna. Il resto è l'algoritmo che riempie la tabella e l'ultima parte serve a trovare il minimo.

1.4.8 Reality check

Approximate String Matching

Approximate String Matching è un esempio di string metric, si utilizza per misurare la distanza (come inverso della similarità) tra due stringhe. Può essere usato in:

- Fraud detection;
- Fingerprint analysis;
- Plagiarism detection;
- DNA-RNA analysis.

1.5 Insieme indipendente di intervalli pestai

Input

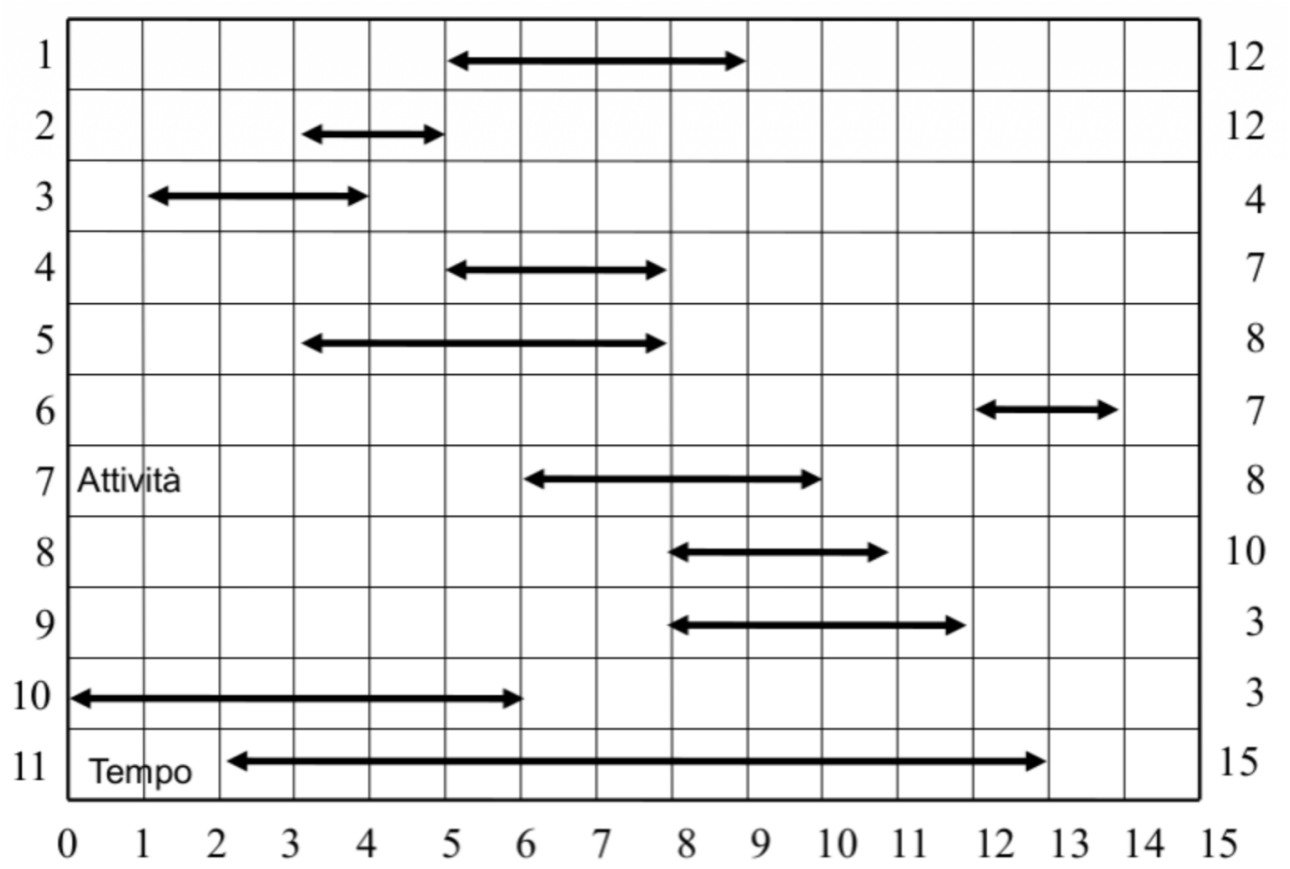
Siano dati n intervalli distinti $[a_1, b_1[, \ldots, [a_n, b_n[$ della retta reale, aperti a destra, dove all'intervallo i è associato un profitto w_i, $1 \leq i \leq n$

Si hanno n intervalli della retta reale aperti a destra, in modo che più intervalli possano essere compatibili.

Due intervalli i e j si dicono disgiunti se: $b_j \leq a_i$ oppure $b_i \leq a_j$. Questa versione del problema associa ad ogni intervallo un peso. Il problema consiste nel trovare un sottoinsieme

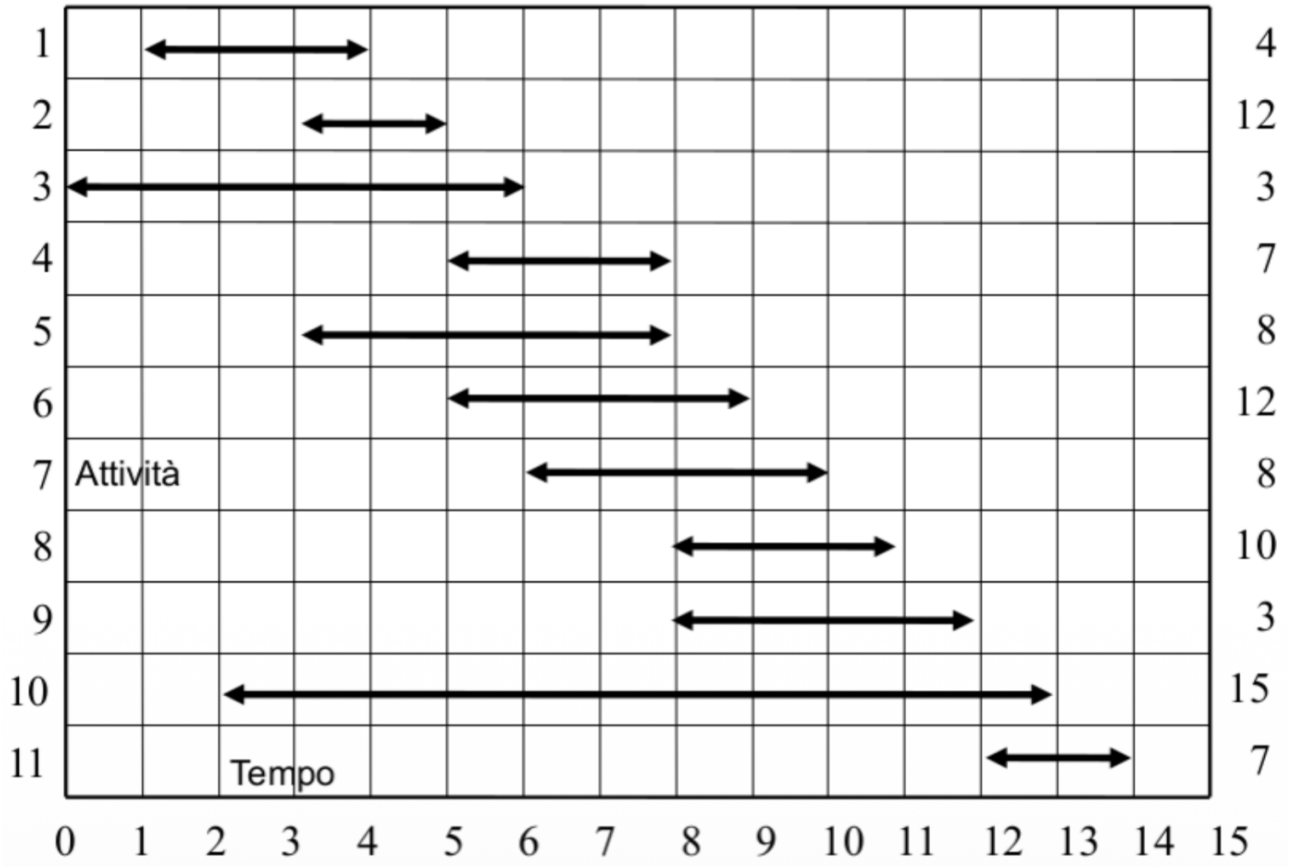
degli intervalli possibili, tale per cui essi siano disgiunti e la somma dei loro pesi sia il massimo possibile.

1.5.1 Esempio: insieme indipendente di peso massimo



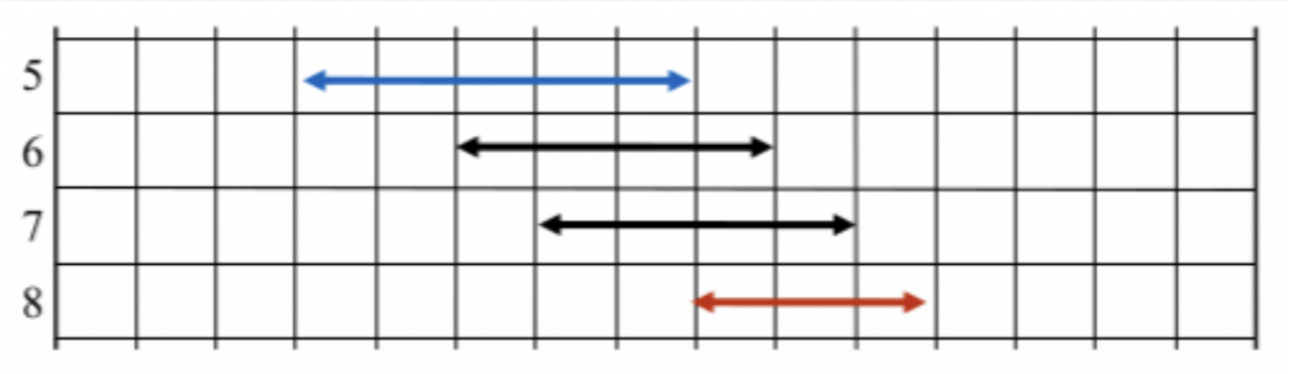
1.5.2 Pre-elaborazione

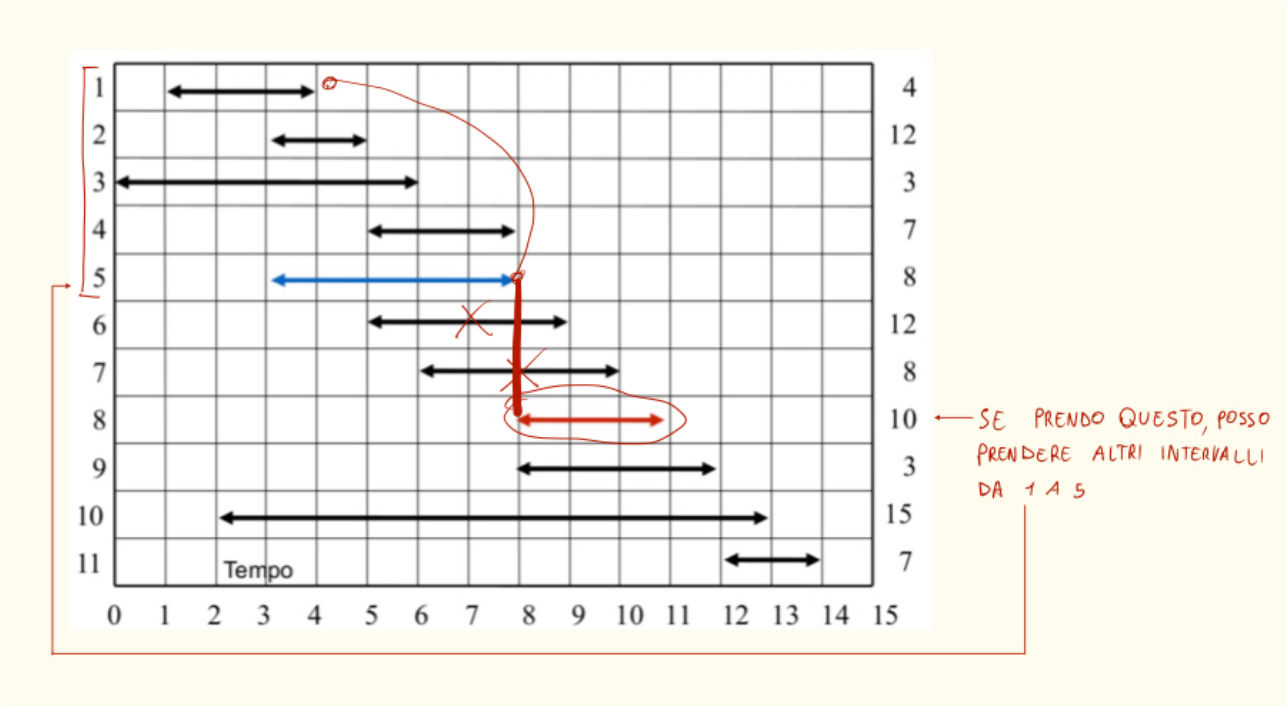
In alcuni problemi di programmazione dinamica, prima di operare sui dati, conviene ordinarli, ovvero fare una **pre-elaborazione**. In questo caso vengono ordinati gli intervalli per estremi finali non decrescenti e si ha un costo di ordinamento pari a $O(n \log n)$, perché si sta riordinando per tempo di fine.



Si fa anche una seconda pre-elaborazione che consiste nel definire il **predecessore** di i (quindi di ogni intervallo) come $p_i = j$, con $j < i$.

- j è il massimo indice tale che $[a_j, b_j[$ **non interseca** $[a_i, b_i[$;
- se non esiste $j \rightarrow p_i = 0$. Ovvero se non esiste l'indice j non c'è nessun predecessore.





1.5.3 Individuazione sottostruttura ottima (forma ricorsiva)

Siano $P[i]$ il sottoproblema dato dai primi i intervalli ed $S[i]$ una soluzione ottima di peso $D[i]$. Si hanno due opzioni:

- Se l'intervallo $i - esimo$ non fa parte della soluzione ottima $(i \notin S[i]) \rightarrow D[i] = D[i-1]$ (con $D[0] = 0$);
- Se l'intervallo $i - esimo$ fa parte della soluzione ottima $(i \in S[i]) \rightarrow D[i] = w_i + D[p_i]$

N.B. Si dimostrano per assurdo.

Il problema ha sottostruttura ottima

La relazione di programmazione dinamica che si ottiene è: $D[i] = maxD[i-1], w_i + D[p_i]$

Una volta definita la sottostruttura ottima, con una relazione ricorsiva di programmazione dinamica, la tabella di programmazione dinamica viene riempita attraverso un algoritmo iterativo. Si noti che $D[n]$ è il costo (guadagno) della soluzione ottima del problema iniziale. A partire da essa, si calcolano a ritroso gli intervalli che hanno portato a tale soluzione.

1.5.4 Algoritmo iterativo

```
SET maxInterval(int[] a, int[] b, int[] w)
 1: <ordinare gli intervalli per estremi finali crescenti>;<calcolare p_j>
 2: int[] D=new int[0...n]; D[0]=0
 3: for i=1 to n do
 4:     D[i]=max(D[i-1], w[i]+D[p_i])
 5: end for
 6: i=n
 7: SET S=Set()
 8: while i>0 do
 9:     if D[i-1]>w[i]+D[p_i] then
10:         i=i-1
11:     else
12:         S.insert(i); i=p_i
13:     end if
14: end while
15: return S
```

1.5.5 Costo computazionale

- Ordinamento intervalli: $O(n \log n)$;
- Calcolo predecessori: $O(n \log n)$;
- Riempimento tabella D: $O(n)$;
- Ricostruzione soluzione: $O(n)$;
- Costo totale: $O(n \log n)$.

Gli intervalli vano ordinati per tempo non decrescente di fine ed eventuali intervalli con lo stesso tempo di fine possono essere ordinati in qualunque modo. Questo perché $D[i]$ rappresenta il massimo profitto ottenibile con i primi i intervalli. È quindi possibile escludere l'intervallo $i - esimo$ se sceglierne uno precedente j (ma con lo stesso tempo di fine $b[i] = b[j]$ ha un valore $D[j]$ più alto).

1.6 Zaino (Knapsack)

Si ha un insieme di oggetti, caratterizzato da un peso e un profitto e si ha uno zaino con un limite di capacità. Si deve individuare un sottoinsieme di oggetti il cui peso totale sia inferiore alla capacità dello zaino e il valore totale degli oggetti sia massimale, ovvero il più alto o uguale al valore di qualunque altro sottoinsieme di oggetti.

Input

- Vettore w (weight), dove $w[i]$ è il peso dell'oggetto $i - esimo$;
- Vettore p (profit), dove $p[i]$ è il profitto dell'oggetto $i - esimo$;
- La capacità C dello zaino.

Si vuole un sottoinsieme $S \subseteq [1, \ldots, n]$ tale per cui il volume totale espresso come sommatoria dei pesi degli oggetti deve essere minore o uguale alla capacità: $w(S) = \sum_{i \in S} w[i] \leq C$ e il profitto totale deve essere massimizzato: $argmax_s p(S) = \sum_{i \in S} p[i]$

Quale è il valore massimo per questo zaino, con $C = 12$?

Item id	**1**	**2**	**3**	**4**
Weight	12	4	6	2
Profit	26	8	12	4

Quale è il valore massimo per quest'altro zaino, con $C = 12$?

Item id	**1**	**2**	**3**	**4**
Weight	12	4	6	2
Profit	29	9	13	5

1.6.1 Definizione matematica della soluzione

Le tabelle di programmazione dinamica utilizzate fino ad ora erano tutte sotto forma di vettori. Siccome in questo problema si ha una realtà a 2 parametri (il numero di oggetti e la capacità), in questo particolare caso si deve usare una tabella di programmazione dinamica che è una matrice, un array bidimensionale.
Si definisce quindi un sottoproblema, che si va ad identificare con due indici "i" e "c", $DP[i][c]$, come il massimo profitto che può essere ottenuto con i primi $i \leq n$ oggetti contenuti in uno zaino con capacità $c \leq C$.
Il problema originale è: si hanno "n" oggetti e una capacità "C", $DP[n][C]$, ma si vuole sapere cosa succede se si ha una capacità leggermente più piccola, se si hanno meno oggetti, quindi si considera tutto lo spazio di tutti i possibili sottoproblemi presenti.

1.6.2 Parte ricorsiva

Quello che si va a fare è scegliere se prendere o non prendere un oggetto. Se un oggetto **non viene preso** è *come se non fosse mai esistito* e quindi: $D[i][c] = DP[i-1][c]$, la capacità rimane

sempre la stessa e non c'è un'aggiunta del profitto.
Se invece l'oggetto viene preso: $DP[i][c] = DP[i-1][c-w[i]] + p[i]$, si sottrae il peso alla capacità e si aggiunge il profitto relativo. L'indice "i" è sempre "$i-1$" perché poi l'oggetto non si può più prendere, dato che ogni oggetto può essere preso solo una volta.

1.6.3 Casi base

Quali sono i casi base?

- Quando non si hanno più oggetti da scegliere, qual è il profitto massimo che si può ottenere? $\rightarrow 0$;
- Qual è il profitto massimo se non si ha più capacità? $\rightarrow 0$;
- Cosa succede se la capacità è negativa?

La capacità negativa può apparire dal fatto che "$c - w[i]$", se non si fa attenzione $w[i]$ potrebbe essere maggiore della capacità residua e quindi "$c - w[i]$" risulterebbe minore di 0.

$$D[i,c] = \begin{cases} 0 & se\ i = 0\ \vee\ c = 0 \\ -\infty & se\ c < 0 \\ max(DP[i-1][c-w[i]] + p[i], DP[i-1][c]) & altrimenti \end{cases}$$

Se il "$c - w[i]$" porta sullo spazio negativo allora si mette $-\infty$ e si farà un massimo tra $-\infty$ e un valore maggiore o uguale a 0, a questo punto vincerà il valore, altrimenti si possono fare i casi particolari:

$$\begin{cases} DP[i-1][c] & se\ w[i] > c \\ max(DP[i-1][c-w[i]] + p[i], DP[i-1][c]) & se\ w[i] \leq c \end{cases}$$

Una volta che si ha la formula la si traduce in codice (in maniera iterativa).

1.6.4 Zaino algoritmo

```
int knapsack(int[] w, int[] p, int n, int C)
 1: DP = new int[0 ... n][0 ... C]
 2: for i=0 to n do
 3:     DP[i][0] = 0
 4: end for
 5: for c=0 to C do
 6:     DP[0][c] = 0
 7: end for
 8: for i=1 to n do
 9:     for c=1 to C do
10:         if w[i] ≤ c then
11:             DP[i][c] = max(DP[i − 1][c − w[i]] + p[i], DP[i − 1][c])
12:         else
13:             DP[i][c] = DP[i − 1][c]
14:         end if
15:     end for
16: end for
17: return DP[n][C]
```

Nella riga **1**, la tabella ha dimensione "$(n+1) * (C+1)$" perché si ha bisogno di memorizzare anche i casi base. La complessità di questa operazione, ovvero della creazione della tabella, è: $\theta(1)$.
Nella righe **2-7**, siccome il caso base è 0 se $i = 0$ *o* $c = 0$ si inizializza a 0. La complessità di questi cicli `for` è rispettivamente: $\theta(n)$ e $\theta(C)$.
Poi si hanno due cicli `for` per riempire il resto della tabella, nell'`if`, se $w[i] \leq c$ si fa il *max* fra i due casi, quindi quando si prende un oggetto ($DP[i-1][c-w[i]] + p[i]$) e quando non lo si prende ($DP[i-1][c]$), altrimenti si passa al caso in cui $w[i] > c$. La complessità dei due `for` è $\theta(n * C)$, mentre quella dell'`if ...else` è $\theta(1)$.
Alla fine si ritorna la tabella $DP[n][C]$.

1.6.5 Esempio

- $w = [4, 2, 3, 4]$
- $p = [10, 7, 8, 6]$
- $C = 9$

	c									
i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	7	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

1.6.6 Complessità computazionale

C può essere arbitrariamente grande, deve quindi essere un dato appropriato. C non è la dimensione del problema, è infatti parte dell'input.
La dimensione del problema è data da n pesi più n profitti più 1 capacità, "$n + n + 1$".La dimensione del problema risulta come "$2n + 1$", ma la C si usa nel calcolo della complessità: $T(n) = O(nC)$.
Questo è un algoritmo **pseudo-polinomiale**, perché sono necessari $k = \lceil log\, C \rceil$ bit per rappresentare C e quindi la complessità risulta essere $T(n) = O(n2^k)$.

1.7 Memoization - Introduzione

La **Memoization** è una variante della programmazione dinamica, essa dice che "**Non è sempre necessario calcolare le soluzioni di tutti i sottoproblemi**".

1.7.1 Zaino con Memoization

Andando a riguardare la definizione ricorsiva del problema originale e il suo costo ($O(nC)$), non è detto che sia necessario risolvere tutti i sottoproblemi (dipende dai loro valori di volumi e capacità).
La **Memoization** permette un approccio alternativo: fonde l'approccio di memorizzazione della programmazione dinamica, ma è ricorsivo e risolve il problema "dall'alto verso il basso", top-down, si parte dal caso di interesse e su calcolano i sottoproblemi se non sono già stati risolti.
Gli approcci visti fin qui sono iterativi, dal basso verso l'alto e risolvono obbligatoriamente tutti i problemi.

Idea Base

È una procedura che utilizza la tabella come una cache per memorizzare le soluzioni dei problemi che viene inizializzata con un **valore speciale** che indica i sottoproblemi non ancora risolti (ad esempio un valore negativo).
Una procedura controlla se il sottoproblema è stato risolto:

- Se **si**: riutilizza il risultato;
- Se **no**: lo risolve (con chiamata ricorsiva sui sottoproblemi), memorizza il risultato in tabella e lo restituisce al chiamante.

1.7.2 Algoritmo zaino con Memoization

integer zaino(**integer**[] w, **integer**[] p, **integer** i, **integer** c, **integer**[][] DP)

```
1: if i==0 or c==0 then
2:     return 0
3: end if
4: if c<0 then
5:     return −∞
6: end if
7: if DP[i, c]=false then
8:     D[i,c] ← max(zaino(w, p, i − 1, c, DP), zaino(w, p, i − 1, c − w[i], DP)+p[i])
9: end if
10: return DP[i, c]
```

1.7.3 Complessità

Complessità della versione puramente ricorsiva dello Zaino Ricorsivo (senza controllo sulle operazioni già eseguite).

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n-1)+1 & n > 1 \end{cases}$$

$$T(n) = O(2^n)$$

1.7.4 Zaino Memoization: utilizzo tabella come cache

Non tutti gli elementi della matrice sono necessari alla risoluzione del problema.

- $w = [4, 2, 3, 4]$
- $p = [10, 7, 8, 6]$
- $C = 9$

	c									
i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	0	**0**
1	**0**	0	**0**	**0**	**10**	**10**	**10**	**10**	10	**10**
2	0	0	**7**	7	10	**10**	**17**	17	17	**17**
3	0	0	7	8	10	**15**	17	18	18	**25**
4	0	0	7	8	10	15	17	18	18	**25**

Come si ricostruisce la soluzione? Ossia, come si identificano gli oggetti che hanno portato al valore massimale $DP[n][C]$?

1.7.5 Dizionario vs Tabella

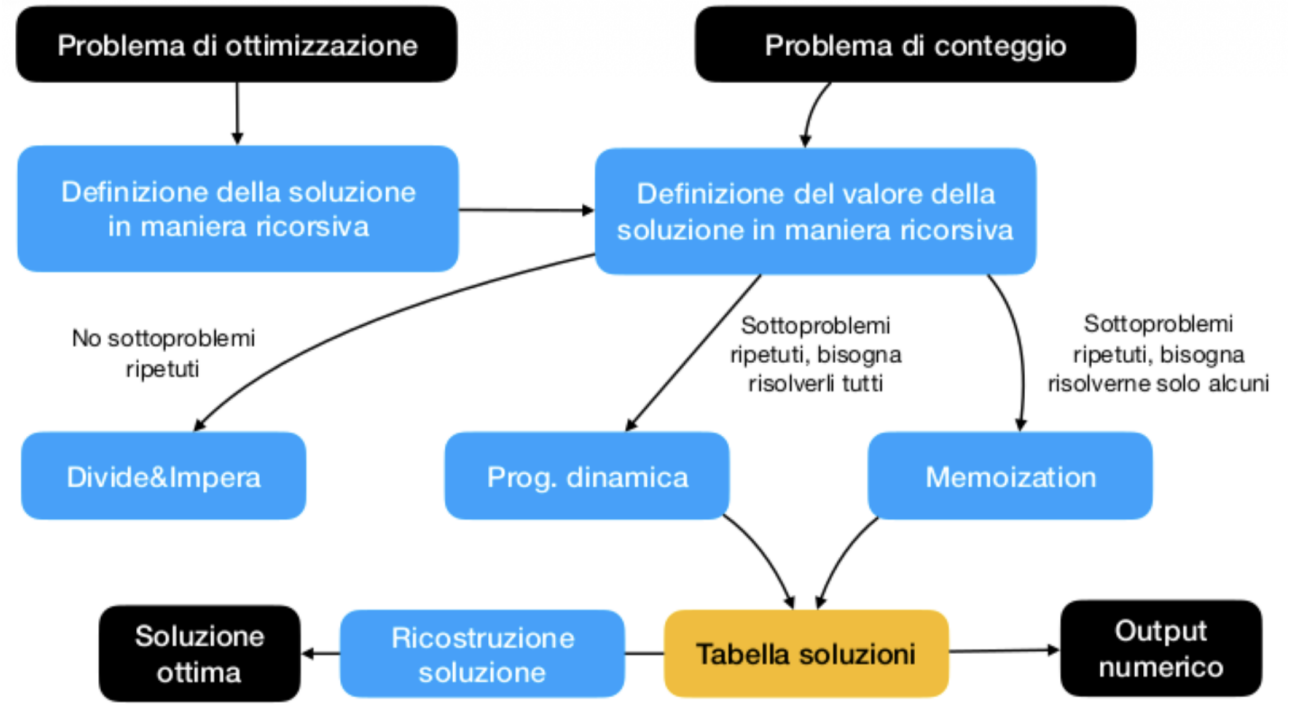
Inizializzazione tabella

- Il costo di inizializzazione è pari a $O(nC)$;
- Applicata in questo modo, non c'è alcun vantaggio nell'utilizzare la tecnica di memoization;
- Permette però di tradurre in fretta le espressioni ricorsive.

Utilizzo di un dizionario (hash table)

- Invece di utilizzare una tabella, si utilizza un dizionario;
- Non è necessario fare inizializzazione;
- Il costo di esecuzione è pari a $O(min(2^n,\ nC))$

1.7.6 Approccio generale



1.7.7 Riassunto: programmazione dinamica/memoization

Fasi:

- Caratterizzare la **struttura** di una soluzione ottima;
- Dimostrare che la soluzione gode di una **sottostruttura ottima**;
- Definire ricorsivamente il **valore** di una soluzione ottima;
- Calcolare il **valore** di una soluzione "bottom-up" (prog. dinamica)/ "top-down" (memoization);
- **Ricostruzione** di una soluzione ottima.