PROGRAMMAZIONE DINAMICA

(Dynamic Programming)

Prof. Dalla Pozza

ITT Chilesotti

Programmazione dinamica (Dynamic Programming)

Tecnica di OTTIMIZZAZIONE (risolve un problema di minimo o massimo) che permette di definire un algoritmo per la soluzione del problema.

Tecnica POTENTISSIMA e GENERICA: gli algoritmi «complessi» che vedrete a scuola/università si possono formulare con DP.

Difficoltà:

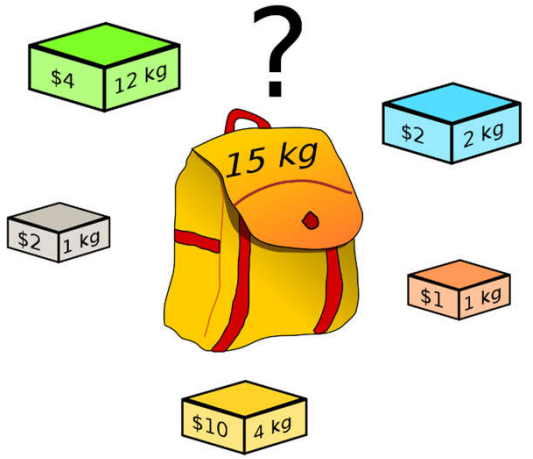
- Capire se è il caso di usarla o usare un'altra tecnica più semplice
- Scrivere l'algoritmo (ovvero riformulare il problema per usarla)

Esempio di problema: Knapsack Problem

- Il PROBLEMA DELLO ZAINO (knapsack problem) è risolvibile con DP:

  Un ladro entra a rubare in una casa con uno zaino che può portare M chili. Dentro la casa trova un'infinità di oggetti di N tipi diversi, ognuno con un peso P[i] e un valore V[i], i=1…N Come scegliere gli oggetti da inserire nello zaino in modo da MASSIMIZZARE IL VALORE TOTALE contenuto?

Nota: non si possono frazionare gli oggetti.



- Come risolvereste il problema?

Variante del problema (0-1 knapsack problem):
si possono prendere sono 0 o 1 oggetti per tipo.

Knapsack Problem: possibili approcci

- Approccio BRUTEFORCE (ESAUSTIVO): elenco e controllo tutte le possibili sequenze.
  - *IMPRATICABILE(se non per pochi oggetti)*

- Approccio GREEDY: ordino gli oggetti dal valore più grande al più piccolo, ed inizio a riempire lo zaino scegliendo di volta in volta l'oggetto che ci sta con il valore massimo.
  - *In generale non dà la soluzione ottima (dipende dai dati del problema), quindi per alcuni casi test potrebbe funzionare per altri no.*

Knapsack Problem: possibili approcci

Controesempio (ovvero un esempio che confuta l'ipotesi di ottimalità) per l'approccio greedy

$$M=9,\ P[0]=6,\ V[0]=8,\ P[1]=4,\ V[1]=7$$

L'approccio greedy mi direbbe di prendere l'oggetto con valore maggiore (sono già ordinati) ottenendo la soluzione

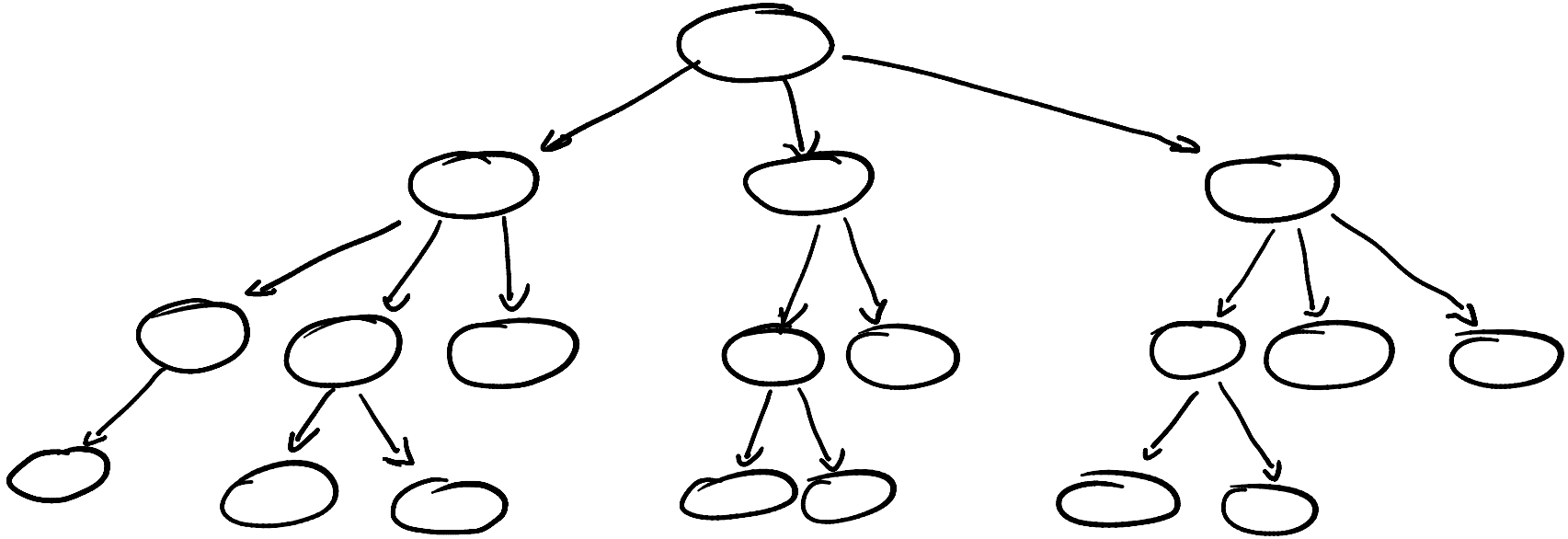
$$S=8 \text{ (totale peso 6)},$$

Ma inserendo due oggetti i=1 ottengo

$$S=14 \text{ (totale peso 8)}$$

Approccio Programmazione Dinamica

La programmazione dinamica sfrutta il fatto un problema si possa scomporre in più SOTTOPROBLEMI più facili da risolvere, e da queste soluzioni COSTRUIRE quella del problema originario. Spesso il sottoproblema è il problema originario «con meno dati», ovvero è la situazione che si presenta dopo aver fatto alcune scelte.

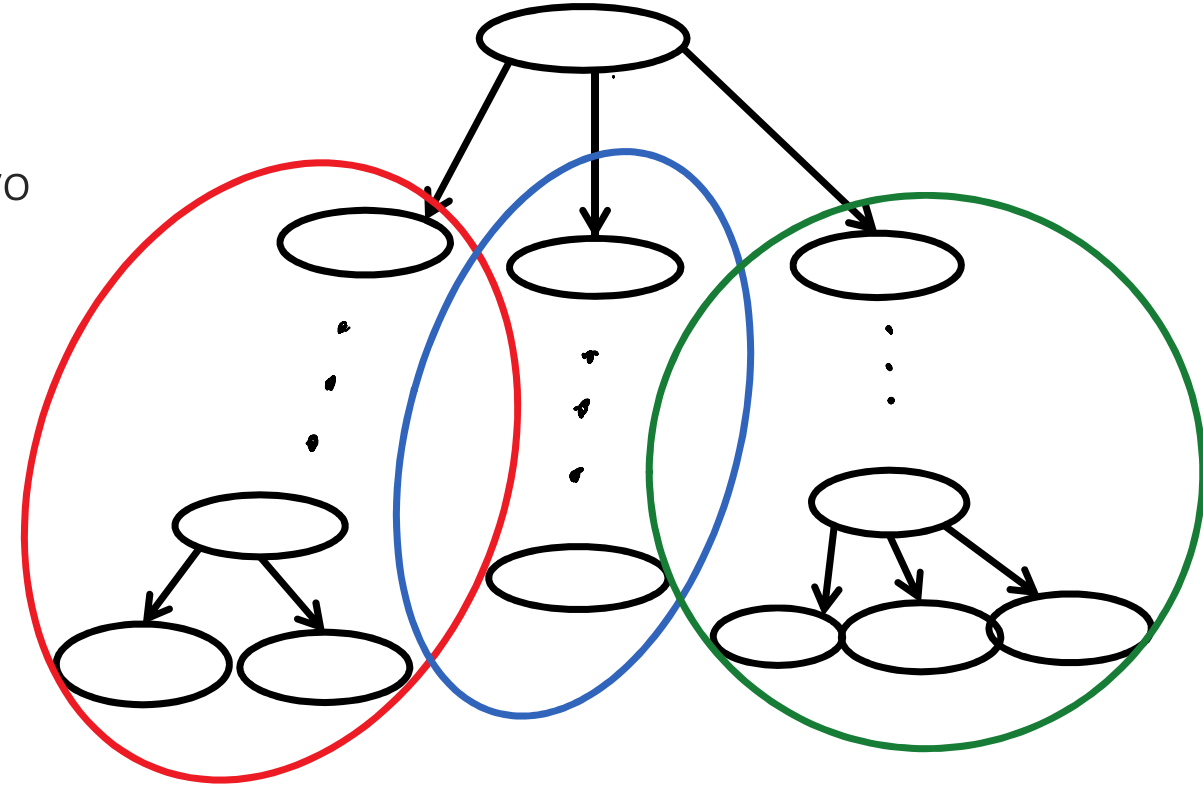


Approccio Programmazione Dinamica

Si può scomporre il problema con due approcci: TOP-DOWN (RICORSIVO) e BOTTOM UP (ITERATIVO).

TOP-DOWN significa che si parte dalla cima e si scende, partendo dall'obiettivo finale e scomponendo il problema in sottoproblemi sempre più piccoli.

Equivalentemente si scrive la soluzione finale a partire da soluzioni parziali.

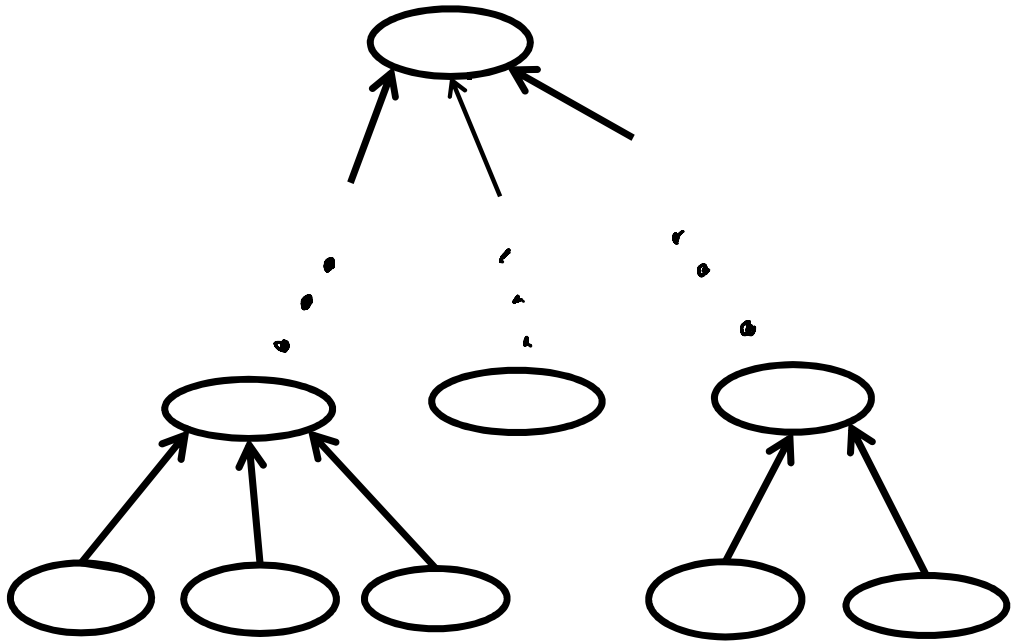


Approccio Programmazione Dinamica

Si può scomporre il problema con due approcci: TOP-DOWN (RICORSIVO) e BOTTOM UP (ITERATIVO).

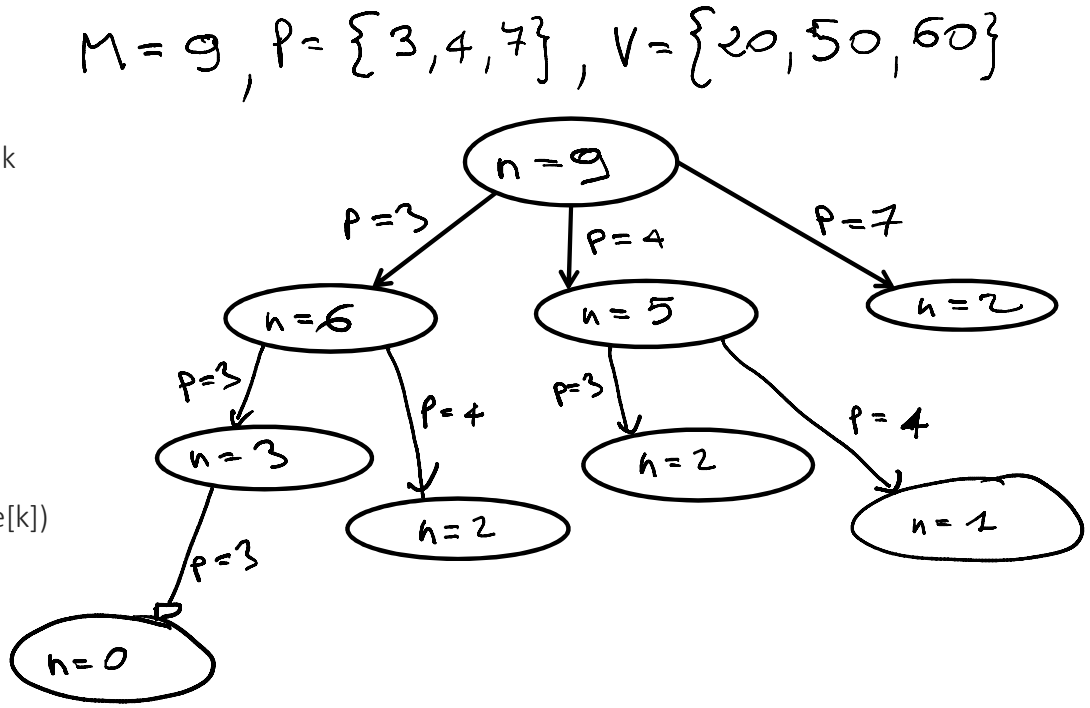
BOTTOM-UP significa invece partire dai casi particolari (dal dettaglio) e poi risalire verso l'alto per raggiungere l'obiettivo finale.

Equivalentemente, costruire la soluzione a partire dalle "prime scelte".



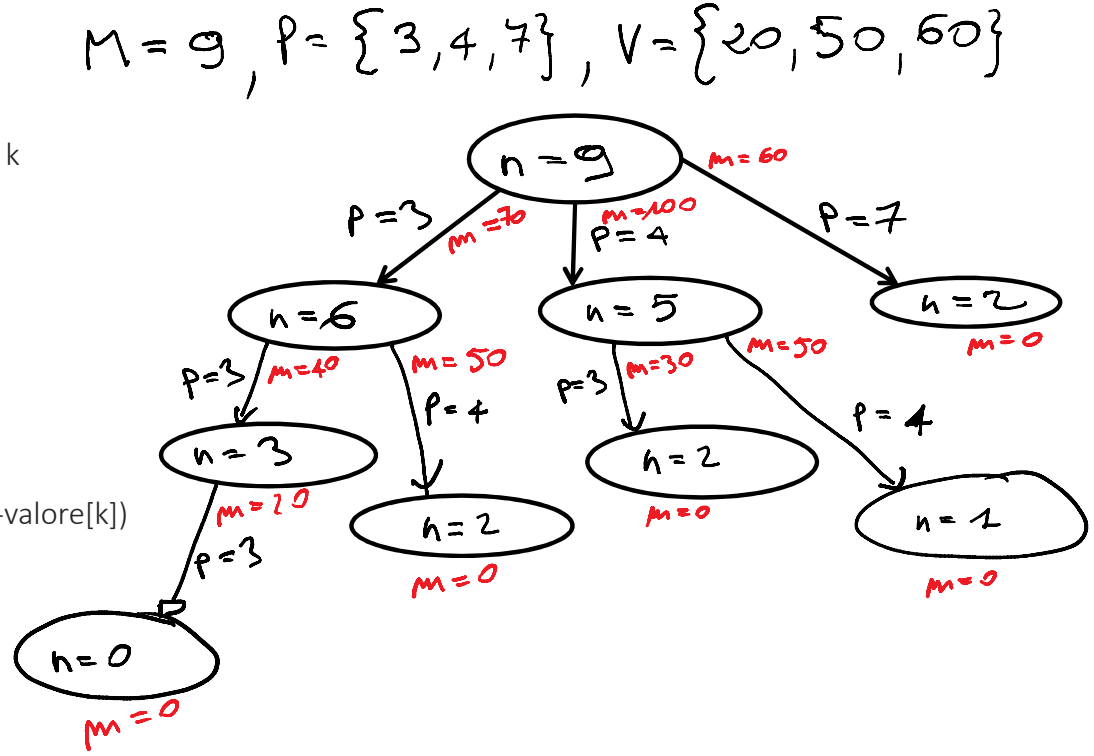
Approccio Programmazione Dinamica

```
// approccio top-down ricorsivo
int N, M; // numero oggetti, peso zaino
int peso[100], valore[100]; // peso[k]/valore[k] peso/valore dell'oggetto k


int knapsack_ricorsivo(int n) {
   if (n==0) return 0;  // caso base della ricorsione
   int massimo = 0;
   for (int k=0; k<N; k++) {
       if (n>=peso[k] && massimo < knapsack_ricorsivo(n-peso[k]) +valore[k])
          massimo = valore[k] + knapsack_ricorsivo(n-peso[k]);
  }
   return massimo;
}
int r = knapsack_ricorsivo(M); // chiamata iniziale nel main
```



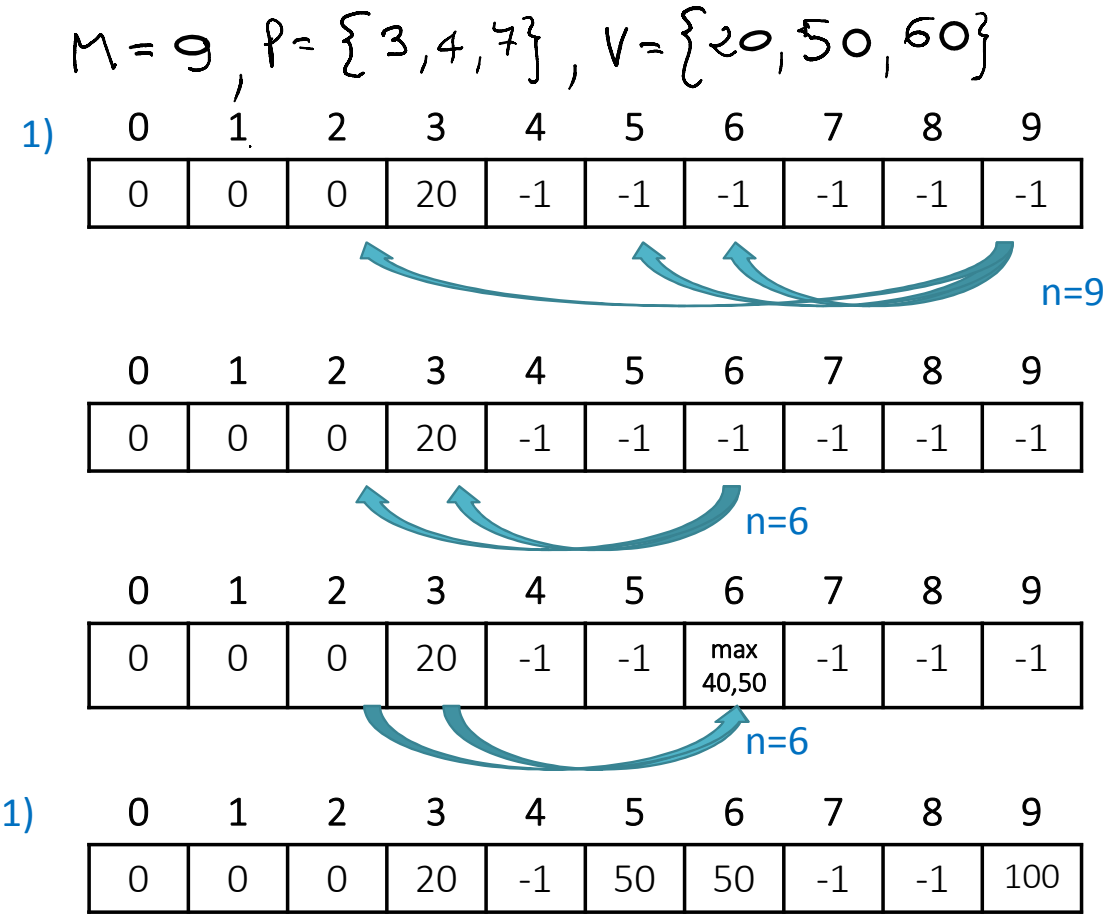
Approccio Programmazione Dinamica

```
// approccio top-down ricorsivo
int N, M; // numero oggetti, peso zaino
int peso[100], valore[100]; // peso[k]/valore[k] peso/valore dell'oggetto k


int knapsack_ricorsivo(int n) {
   if (n<peso[0]) return 0;  // se i pesi sono ordinati
   int massimo = 0, tmp;
   for (int k=0; k<N; k++) {
       if (n>=peso[k] && (tmp=knapsack_ricorsivo(n-peso[k])) > massimo-valore[k])
          massimo = valore[k] + tmp;    // con tmp risparmio chiamate
  }
   return massimo;
}
int r = knapsack_ricorsivo(M); // chiamata iniziale nel main
```



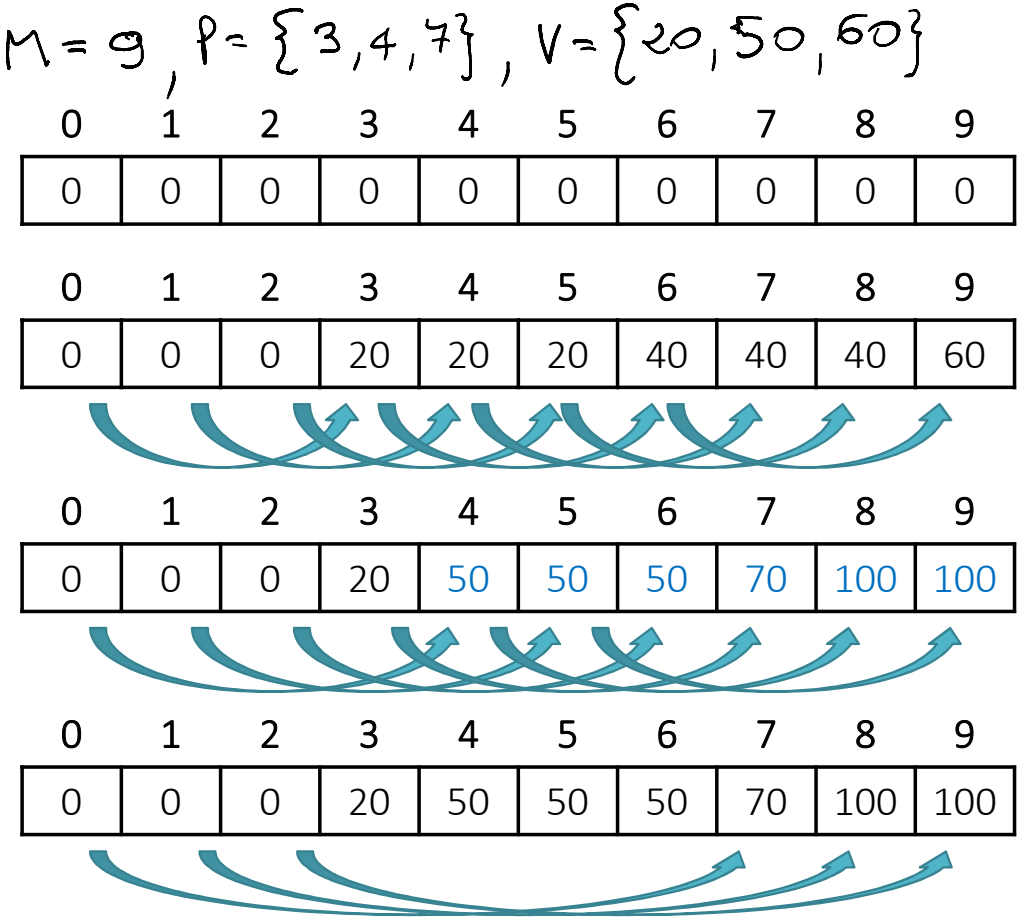
Approccio Programmazione Dinamica

```
// approccio top-down ricorsivo con MEMOIZATION
int N, M; // numero oggetti, peso zaino
int peso[100], valore[100]; // peso[k]/valore[k] peso/valore dell'oggetto k
int sol[100]; // sol[k] indica la soluzione del problema con peso dello zaino k
int knapsack_top_down(int n) {
   if (sol[n] != -1) return sol[n]; // memoization
   int massimo = 0;
   for (int k=0; k<N; k++)
      if (n>=peso[k] && valore[k] + knapsack_top_down(n-peso[k]) > massimo)
         massimo = valore[k] + knapsack_top_down(n-peso[k]);
   sol[n] = massimo; // memoization
   return massimo;
}
sol[peso_minimo] = valore[elemento_peso_minimo]; // caso base nel main
for (int k=peso_minimo+1; k<=M; k++)  {sol[k] = -1;}   // valori da calcolare
int r = knapsack_top_down(M); // nel main
```



Approccio Programmazione Dinamica

```
// approccio iterativo per unbounded knapsack problem
int N, M; // numero oggetti, peso zaino
int peso[100], valore[100]; // peso[k]/valore[k] peso/valore dell'oggetto k

int sol[100]; // sol[k] indica la soluzione del problema con peso dello zaino k
int knapsack_bottom_up() {
   for (int i = 0; i < N; i++)                  // per ogni oggetto i
      for (int j = 0; j <= M - peso[i]; j++)  // per ogni capacità dello zaino j
         if (sol[j+peso[i]] < sol[j] + valore[i])   // controllo sol[j+peso[i]]
            sol[j+peso[i]] = sol[j] + valore[i];   // se non è maggiore aggiorno
   return sol[M];
}
 int r = knapsack_bottom_up(); // chiamata iniziale nel main
```

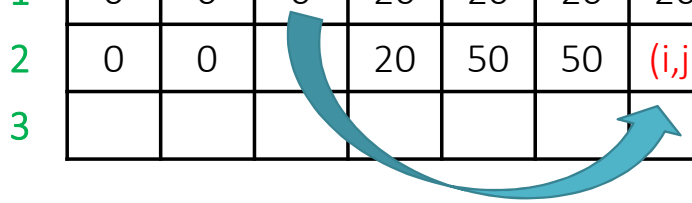


Approccio Programmazione Dinamica

```
// approccio iterativo per 0-1 knapsack problem
int N, M; // numero oggetti, peso zaino
// int peso[100], valore[100]; // peso[k]/valore[k] peso/valore dell'oggetto k
int sol[100][100]; // sol[i][j] la soluzione con oggetti 1...i (li conto da 1) con zaino j
int knapsack_bottom_up() {
   for (int i = 1; i <= N; i++) {  // per ciascun gruppo di oggetti fino a i
      for (int j = 0; j <= M; j++) { // per ciascuna capacità dello zaino
         sol[i][j] = sol[i - 1][j];
         if (j >= peso[i-1] && sol[i][j] < sol[i - 1][j - peso[i - 1]] + valore[i - 1]) {
            sol[i][j] = sol[i - 1][j - peso[i - 1]] + valore[i - 1];
         }
      }
   }
   return sol[N][M];
}
 int r = knapsack_bottom_up(); // chiamata iniziale nel main
```

$M = 9, P = \{3,4,7\}, V = \{20,50,60\}$

i \ j	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	20	20	20	20	20	20	20
2	0	0		20	50	50	(i,j)			
3										



0+50

Approccio Programmazione Dinamica

```
// approccio iterativo per 0-1 knapsack problem
int N, M; // numero oggetti, peso zaino
int peso[100], valore[100]; // peso[k]/valore[k] peso/valore dell'oggetto k
int sol[100][100]; // sol[i][j] la soluzione con oggetti 1...i (li conto da 1) con zaino j
int knapsack_bottom_up() {
   for (int i = 1; i <= N; i++)
      for (int j = 0; j <= M; j++) {
         sol[i][j] = sol[i - 1][j];
         if (j >= peso[i-1] && sol[i][j] < sol[i - 1][j - peso[i - 1]] + valore[i - 1])
               sol[i][j] = sol[i - 1][j - peso[i - 1]] + valore[i - 1];
      }
   return sol[N][M];
}
 int r = knapsack_bottom_up(); // chiamata iniziale
```

$M = 9, P = \{3,4,7\}, V = \{20,50,60\}$

i \ j	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	20	20	20	20	20	20	20
2	0	0	0	20	50	50	50	70	70	70
3	0	0	0	20	50	50	50	70	70	70

soluzione finale problema

Approccio Programmazione Dinamica

Gli approcci TOP-DOWN e BOTTOM UP sono entrambi validi e portano alla stessa soluzione

A volte potrebbe essere più semplice vedere un problema in un modo piuttosto che nell'altro.

Non sono proprio equivalenti dal punto di vista pratico. Le chiamate ricorsive ai sottoproblemi richiedono tempo e memoria (che è limitata) > MEMOIZATION

Approccio Programmazione Dinamica

Riassumendo:

1. Capire se è il caso di usare la DP o se usare un'altra tecnica più semplice (greedy?). Questo si può fare facilmente se si trova un controesempio che mostri che la soluzione greedy non è ottima. Molti problemi che richiedono di definire una sequenza di azioni richiedono la DP.
2. (STEP PIU' DIFFICILE) Riformulare il problema per la DP: vuol dire individuare i sottoproblemi (ovvero il dato su cui fare la ricorsione) e quali dati invece sono "globali", e capire quale "azione" collega il sottoproblema a quello generale.
3. Decidere se è più semplice l'approccio top-down o bottom-up.

(continua)

Approccio Programmazione Dinamica

(continua)

4. Scrivere l'algoritmo (BISOGNA IMPARARSI LA STRUTTURA DI UN ALGORITMO DP) e testarlo.

5. Se non riesco a formulare la DP, posso provare a ottenere qualche punto con una soluzione greedy o esaustiva, che risolve i casi più semplici.