Capitolo 13

Programmazione Dinamica

> *Those who cannot remember the past are condemned to repeat it*
>
> George Santayana, *1905*

Introduzione

La programmazione dinamica è un metodo per spezzare ricorsivamente un problema in sottoproblemi, i quali vengono risolti una sola volta e la loro soluzione viene memorizzata in una tabella. Nel caso il sottoproblema debba essere risolto nuovamente, si recupera la soluzione dalla tabella. La tabella è facilmente indirizzabile: la sua consultazione costa $\mathcal{O}(1)$.

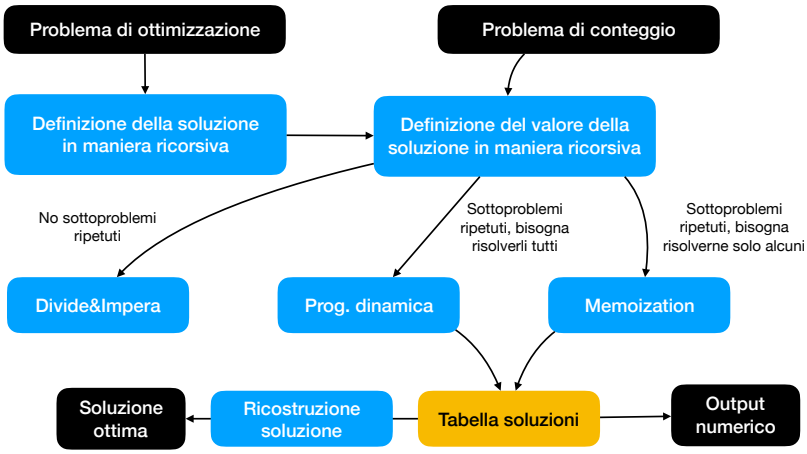

Figura 13.1: Approccio generale ad un problema

La programmazione dinamica nella storia

Il termine *Dynamic Programming* è stato coniato da Richard Bellman agli inizi degli anni '50, nell'ambito dell'ottimizzazione matematica. Inizialmente, si riferiva al processo di risolvere un problema compiendo le migliori decisioni una dopo l'altra. "*Dynamic*" doveva dare un senso "temporale", mentre "*Programming*" si riferiva all'idea di creare "programmazioni ottime", per esempio nella logistica. È possibile approfondire all'indirizzo `https://en.wikipedia.org/wiki/Dynamic_programming#History`.

La *programmazione dinamica* è caratterizzata da 4 fasi principali:

1. caratterizzare *matematicamente* la struttura di una soluzione ottima;
2. definire ricorsivamente il valore di una soluzione ottima;
3. calcolare il valore di una soluzione ottima con approccio *bottom-up* ed utilizzare una tabella per memorizzare la soluzione dei sottoproblemi ed utilizzarla per evitare di ripetere i calcoli più volte;
4. ricostruire la soluzione ottima.

13.1 Domino

Definizione del problema Il gioco del domino è basato su tessere di dimensione 2×1. Scrivere un algoritmo efficiente che prenda in input un intero n e restituisca il numero di possibili disposizioni di n tessere in un rettangolo $2 \times n$.

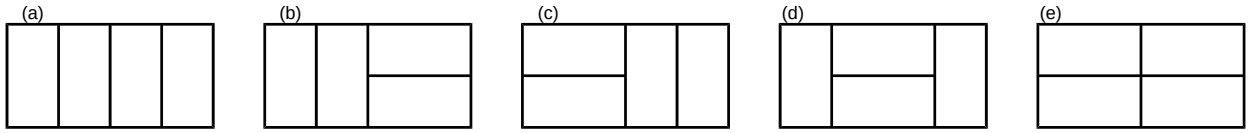

Figura 13.2: Rappresentazione delle cinque disposizioni possibili con cui è possibile riempire un rettangolo 2×4.

Definizione ricorsiva

Definiamo una formula ricorsiva $DP[n]$ che ci permetta di calcolare il numero di disposizioni possibili quando si hanno n tessere. Con nessuna tessera ($n = 0$) esiste una sola disposizione possibile. Avendo a disposizione una tessera ($n = 1$) è possibile disporla solo verticalmente.

Se posiziono una tessera in verticale, risolverò il problema di dimensione $n-1$, mentre se la posiziono in orizzontale ne devo mettere due, risolvendo così il problema di dimensione $n-2$. Queste ultime due possibilità si sommano insieme (conteggio).

$$DP(n) = \begin{cases} 1 & n \leqslant 1 \\ DP[n-2] + DP[n-1] & n > 1 \end{cases}$$

La serie generata è una successione di Fibonacci. $DP[n]$ infatti è pari al $n+1$-esimo numero della serie.

Algoritmo ricorsivo

L'algoritmo che viene scaturito dalla definizione ricorsiva del problema è il seguente:

Algoritmo 1: Algoritmo *ricorsivo* che risolve il problema Domino

```
int domino1(int n)
    if n ⩽ 1 then
        return 1
    else
        return domino1(n − 1) + domino1(n − 2)
```

Analisi della complessità L'equazione di ricorrenza associata a domino1 è la seguente:

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

È una ricorrenza lineare di ordine costante, per calcolare la sua complessità applichiamo quindi il *master theorem*: i fattori moltiplicativi di $T(n-1)$ e $T(n-2)$ sono $a_1 = 1$ e $a_2 = 1$ rispettivamente, possiamo raccogliergli in $a = a_1 + a_2 = 2$, il fattore β risulta pari a 0, in quanto $n^0 = 1$. Possiamo quindi concludere che la complessità dell'algoritmo è $\Theta(a^n \cdot n^\beta) = \Theta(2^n)$.

L'albero di ricorsione generato dall'algoritmo è il seguente:

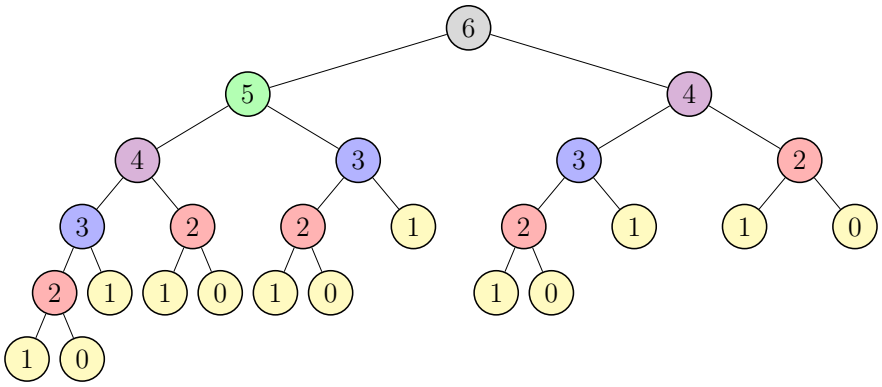

Figura 13.3: Albero di ricorsione per domino1. Possiamo notare che molti sottoproblemi vengono ripetuti.

Come evitare di risolvere un problema più di una volta

Dall'albero di ricorsione possiamo notare che molti sottoproblemi vengono ripetuti. Per evitare che questo avvenga memorizziamo il risultato ottenuto risolvendo un particolare problema in una **tabella DP**, la quale sarà un vettore, una matrice o dizionario dipendentemente dalle nostre esigenze. La tabella conterrà un elemento per ogni sottoproblema che dobbiamo risolvere. Memorizzeremo i casi base nelle relative posizioni. Dopodiché l'iterazione sarà *bottom-up*: partiremo dai casi base e andremo verso problemi via via sempre più grandi fino a risolvere il problema originale.

Algoritmo 2: Algoritmo *iterativo* che risolve il problema Domino

```
int domino2(int n)
    DP ← new int[0...n] // inizializzo la "tabella" DP
    DP[0] ← DP[1] ← 1 // inserisco i casi base
    // computo i valori successivi sulla base dei valori precedenti
    from i ← 2 until n do
        DP[i] ← DP[i − 1] + DP[i − 2]
    return DP[n] // restituisco il valore n-esimo richiesto
```

Analisi della complessità La complessità in tempo è $T(n) = \Theta(n)$, quella in spazio è $S(n) = \Theta(n)$. È possibile migliorare l'algoritmo riducendo lo spazio utilizzato.

Tabella 13.1: I casi base vengono inseriti manualemente nelle relative posizioni.
Ogni valore i-esimo successivo viene computato sulla base dei suoi valori precedenti $i-1$ e $1-2$.

n	0	1	2	3	4	5	6	7
$DP[n]$	*1*	*1*	2	3	5	8	13	21

Algoritmo 3: Algoritmo *iterativo che ottimizza lo spazio utilizzato* che risolve il problema Domino

```
int domino3(int n)
    int DP_0 ← 1
    int DP_1 ← 1
    int DP_2 ← 1
    from i ← 2 until n do
        DP_0 ← DP_1
        DP_1 ← DP_2
        DP_2 ← DP_0 + DP_1
    return DP_2
```

Analisi della complessità Questa implementazione ha costo costante nello spazio $S(n) = \Theta(1)$.

n	0	1	2	3	4	5	6	7
$DP[n]_0$	–	–	1	1	2	3	5	8
$DP[n]_1$	*1*	*1*	1	2	3	5	8	13
$DP[n]_2$	*1*	*1*	2	3	5	8	13	21

Sotto il modello di costo logaritmico, le tre versioni hanno le complessità mostrate nella tabella.

Tabella 13.2: Confronto delle varie versioni della funzione di fibonacci

Funzione	Tipologia	$T(n)$	$S(n)$
domino1	ricorsiva	$\mathcal{O}(n2^n)$	$\mathcal{O}(n^2)$
domino2	iterativa	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
domino3	finale	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$

Si può fare meglio di così utilizzando l'esponenziazione di matrici basata su quadrati (per approfondimenti consultare `https://brilliant.org/wiki/fast-fibonacci-transform/`).

13.2 Hateville

Descrizione del problema Hateville è un villaggio particolare, composto da n case, numerate da 1 a n lungo una singola strada. Ad Hateville ognuno odia i propri vicini della porta accanto, da entrambi i lati. Quindi il vicino i odia i vicini $i-1$ e $i+1$ (se esistenti). Hateville vuole organizzare una sagra e vi ha affidato il compito di raccogliere i fondi. Ogni abitante i ha intenzione di donare una quantità $D[i]$, ma non intende partecipare ad una raccolta fondi a cui partecipano uno o entrambi i propri vicini.

Dobbiamo scrivere un algoritmo che restituisca la quantità massima di fondi che può essere raccolta.

Consegne del problema I problemi che possono esserci posti sono due:

1. scrivere un algoritmo che restituisca la quantità massima di fondi che può essere raccolta;
2. scrivere un algoritmo che restituisca il sottoinsieme di indici $S \subseteq \{1, \ldots, n\}$ tale per cui la donazione totale $T = \sum_{i \in S} D[i]$ è massimale.

se risolviamo il primo siamo ad un passo dalla soluzione del secondo, mentre se risolviamo il primo abbiamo risolto necessariamente il primo.

Esempi di esecuzione Con un vettore di donazioni $D = [4, 3, 6]$ la raccolta massima è 10, dato dall'insieme di indici $\{1, 3\}$. Mentre con un vettore di donazioni $D = [10, 5, 5, 10]$ la raccolta massima è 20, dato dall'insieme di indici $\{1, 4\}$.

La domanda che dobbiamo porci è la seguente: è possibile ridefinire una formula ricorsiva che ci permetta di calcolare il sottoinsieme di case che, se selezionate, dà origine alla maggior quantità di donazioni?

13.2.1 Definizione ricorsiva

Ridefiniamo il problema caratterizzandolo matematicamente.

Definiamo $HV(i)$ uno dei possibili insiemi di indici da selezionare per ottenere una donazione ottimale delle prime i case di Hateville, numerate $1, \ldots, n$. $HV(n)$ diventa quindi la soluzione del problema originale.

Passo ricorsivo

Andiamo per passi. Consideriamo il vicino i-esimo.

- cosa succede se **non accetto** la sua donazione? Lo scarto. Proviamo ad esprimerlo in funzione dei problemi precedenti (dopotutto il problema viene risolto in maniera ricorsiva):

$$HV(i) = HV(i-1)$$

- cosa succede se **accetto** la sua donazione? Accetto la donazione i-esima e scarto i vicini. In simboli:

$$HV(i) = \{i\} \cup HV(i-2)$$

- a questo punto come faccio a **decidere** quale delle due opzioni scegliere? Semplicemente prendo quello che mi dà un guadagno maggiore. In simboli:

$$HV(i) = highest(HV(i-1), \{i\} \cup HV(i-2))$$

La funzione *highest* restituisce l'insieme di valore massimo.

Sottostruttura ottima

Quando voglio provare che una soluzione di programmazione dinamica è corretta devo riuscire a dimostrare che le mie possibili scelte sono quelle giuste. Per dimostrarlo utilizziamo il teorema di sottostruttura ottima, che dice sostanzialmente che il modo in cui ho applicato la ricorsione è corretto.

Proviamo quindi a dimostrare le scelte fatte. Il problema dato dalle prime i case è indicato con $HV_p(i)$ e una (ce ne può essere più di una) soluzione ottima per questo problema è indicato con $HV_s(i)$.

Se $i \notin HV_s i$ allora $HV_s(i) = HV_s(i-1)$ (ossia se l'i-esimo elemento non appartiene alla soluzione del problema $HV_s i$ allora abbiamo scartare quella donazione $HV_s(i-1)$), altrimenti se l'abbiamo accettata e quindi $i \in HV_s i$ allora $HV_s(i) = HV_s(i-2) \cup \{i\}$.

Se abbiamo la soluzione ottima, allora possiamo dimostrare che abbiamo la soluzione ottima per i rispettivi sottoproblemi (da qui sottostruttura).

Nota. Nella maggior parte dei casi non è necessaria una dimostrazione della soluzione, ma basta un'intuizione.

13.2.2 Dimostrazione sottostruttura ottima

Ricordiamo che indichiamo con $HV_p(i)$ il problema dato dalle prime i case e con $HV_s(i)$ una delle possibili soluzioni ottime per questo problema. Indichiamo inoltre con $|HV_s(i)|$ l'ammontare di donazioni per la soluzione ottima $HV_s(i)$.

Nota. Dobbiamo dimostrare separatamente il caso in cui prendiamo la donazione i-esima e il caso in cui non la prendiamo, in quanto sono eventi mutualmente esclusivi.

Entrambe le dimostrazioni procedono per assurdo.

Dimostrazione caso 1: $\boxed{i \notin HV_s(i)}$. Se non abbiamo preso la donazione i-esima vogliamo dimostrare che la soluzione ottima $HV_s(i)$ è una soluzione ottima anche per il problema precedente $HV_p(i-1)$.

Se così non fosse esisterebbe una soluzione (migliore) $HV_s^{'}(i-1)$ per il problema $HV_p(i-1)$ tale che l'ammontare delle donazioni del problema precedente sarebbe maggiore (in simboli $|HV_s^{'}(i-1)| > |HV_s(i)|$). Ma allora $HV_s^{'}(i-1)$ sarebbe una soluzione per $HV_p(i)$, che è assurdo. Quindi $HV_s(i)$ è una soluzione ottima anche per $HV_p(i-1)$. □

Dimostrazione caso 2: $\boxed{i \in HV_s(i)}$. Se abbiamo preso l'i-esima donazione vogliamo dimostrare che $i-1$ non appartiene alla soluzione ottima (in simboli $i-1 \notin HV_s(i)$), altrimenti non sarebbe una soluzione ammissibile. Quindi, se dalla soluzione ottima $HV_s(i)$ togliessimo la donazione i-esima ($HV_s(i) - \{i\}$) questa dovrebbe essere una soluzione ottima per $HV_p(i-2)$.

Se così non fosse, esisterebbe una soluzione $HV_s^{'}(i-2)$ per il problema $HV_p(i-2)$ tale che il suo guadagno sarebbe maggiore (in simboli $|HV_s^{'}(i-2)| > |HV_s(i) - \{i\}|$). Ma allora $HV_s^{'}(i-2) \cup \{i\}$ sarebbe una soluzione per $HV_p(i)$, che è assurdo. Quindi $i-1 \notin HV_s(i)$. □

13.2.3 Completare la ricorsione

Ragioniamo sui casi base. Se ho 0 case il mio guadagno è zero: $HV(0) = \varnothing$; se ho una casa prendo semplicemente la sua donazione: $HV(1) = \{1\}$.

Possiamo quindi scrivere la formula per calcolare la somma massima date i case:

$$HV(i) = \begin{cases} 0 & i = 0 \\ \{1\} & i = 1 \\ highest(HV(i-1), HV(i-2) \cup \{i\}) & i \geqslant 2 \end{cases}$$

Non vale la pena scrivere un algoritmo ricorsivo, basato su divide-et-impera, per risolvere il problema di Hateville poiché si risolverebbero molti sottoproblemi più volte.

13.2.4 Memorizzare una tabella

Facciamo qualche esempio di esecuzione. Nel primo il vettore delle donazioni è $D = [10, 5, 5, 8, 4, 7, 12]$, mentre nel secondo è $D = [10, 1, 1, 10, 1, 1, 10]$. Convincersi che gli insiemi risultanti sono corretti.

i	0	1	2	3	4	5	6	7
D		10	5	5	8	4	7	12
HV	$\varnothing$	$\{1\}$	$\{1\}$	$\{1,3\}$	$\{1,4\}$	$\{1,3,5\}$	$\{1,4,6\}$	$\{1,3,5,7\}$

i	0	1	2	3	4	5	6	7
D		10	1	1	10	1	1	10
HV	$\varnothing$	$\{1\}$	$\{1\}$	$\{1,3\}$	$\{1,4\}$	$\{1,4\}$	$\{1,4,6\}$	$\{1,4,7\}$

A questo punto dobbiamo risolvere ancora due problemi:

1. dobbiamo definire la funzione *highest* (ma è banale);
2. dobbiamo memorizzare gli insiemi nella tabella.

Memorizzare gli insiemi delle scelte nella tabella è costoso, quindi lo non faremo. Costruiremo invece il valore della soluzione. Così facendo eviteremo di memorizzare gli insiemi nella tabella e potremmo comunque ricostruire la soluzione a posteriori.

Tabella di programmazione dinamica

Indichiamo con $DP[i]$ il **valore** della massima quantità di donazioni che possiamo ottenere dalle prime i case di Hateville, e con $DP[n]$ il valore della soluzione ottima. Possiamo quindi riempire la tabella di programmazione dinamica nel seguente modo:

$$DP[i] = \begin{cases} 0 & i = 0 \\ D[1] & i = 1 \\ \mathsf{max}(DP[i-1], DP[i-2] + DP[i]) & i \geqslant 2 \end{cases}$$

Nel caso avessi 0 donatori allora la somma delle donazioni sarà 0. Mentre nel caso abbia un solo donatore ($i = 1$) allora prenderò la sua donazione ($D[1]$). Infine nel caso avessi 2 o più donatori allora prendere una scelta: o scarterò quella donazione, quindi non considererò più quell'indice ($DP[i-1]$), o la accetter ($+DP[i]$) e dovrò scartare a priori la scelta del suo vicino ($DP[i-2]$). La sceltà è rappresentata dalla funzione max che selezionerà quale fra le due scelte sarà quella più conveniente.

Nota. Non memorizziamo più insiemi, ma valori. Infatti non effettuiamo più l'unione di insiemi ma la somma fra i valori contenuti all'interno della tabella.

Dall'equazione di ricorrenza possiamo scrivere in modo naturale un algoritmo iterativo che risolve questo particolare problema. Nel caso volessimo implementare un algoritmo ricorsivo allora dovremmo utilizzare la tecnica della *memoization* che vedremo più avanti.

Algoritmo 4: Algoritmo iterativo che risolve il problema Hateville

```
int hateville(int[] D, int n)
    // creo la tabella, un vettore in questo caso
    int[] DP ← new int[0...n]

    // inserisco i casi base
    DP[0] ← 0 // nessun donatore
    DP[1] ← D[1] // un solo donatore

    // calcolo il valore n-esimo
    from i ← 2 until n do
        DP[i] ← max(DP[i-1], DP[i-2] + D[i])

    // restituisco il valore n-esimo
    return DP[n]
```

Stiamo calcolando la soluzione per ogni possibile sottoproblema $(n+1)$ qual è il valore massimo della soluzione. Questa soluzione ha complessità $\Theta(n)$ in quanto dobbiamo fare $\Theta(n)$ per ottenere il risultato.

Soluzione con linguaggi di programmazione

Vediamo un paio di implementazioni con "veri" linguaggi di programmazione. Gli indici differiscono dalla notazione matematica.

```java
public int hateville(int[] D, int n) {
    int[] DP = new int[n+1];
    DP[0] = 0;
    DP[1] = D[0]; // l'indice parte da 0
    for (int i=2; i <= n; i++) {
        DP[i] = max(DP[i-1], DP[i-2] + D[i-1]); // devo prendere la donazione i - 1
    }

    return DP[n];
}
```

Codice 13.1: Implementazione della soluzione in Java

```python
def hateville(D):
    DP = [ 0, D[0] ] # l'indice parte da 0

    for i in range(1,len(D)): # scrittura più elegante
        DP.append( max(DP[-1], DP[-2] + D[i]) )

    return DP[-1]
```

Codice 13.2: Implementazione della soluzione in Python

13.2.5 Ricostruire la soluzione originale

Questi sono i possibili risultati che possiamo ottenere applicando l'algoritmo.

i	0	1	2	3	4	5	6	7
D		10	5	5	8	4	7	12
DP	0	10	10	15	18	19	25	31

i	0	1	2	3	4	5	6	7
D		10	1	1	10	1	1	10
DP	0	10	10	11	20	20	21	30

A questo punto abbiamo il valore della soluzione massimale, ma non abbiamo la soluzione, ossia l'insieme degli indici.

Per riscostruire la soluzione guardiamo l'elemento i-esimo presente nella tabella nella posizione $DP[i]$, se la casa i-esima non è stata selezionata allora il valore di $DP[i]$ deriva da $DP[i-1]$, altrimenti (se la casa è stata selezionata) il suo valore deriva da $DP[i-2]+D[i]$. Se entrambe le equazioni sono vere, una vale l'altra.

Utilizziamo quindi questa informazione per ricostruire la soluzione in modo ricorsivo: per ricostruire la soluzione fino ad i, calcoliamo i valori fino a $i-2$ e aggiungiamo i (se la casa è stata selezionata), altrimenti li calcoliamo fino a $i-1$ senza aggiungere nulla.

Algoritmo 5: Ricostruire la soluzione generale di Hateville

```
int hateville(int[] D, int n)
    // creo la tabella
    int[] DP ← new int[0...n]
    // inserisco i casi base
    DP[0] ← 0
    DP[1] ← DP[1]
    // calcolo il valore i-esimo
    from i ← 2 until n do
        DP[i] ← max(DP[i-1], DP[i-2] + D[i])
    // restituisco il valore n-esimo
    return solution(DP, D, n)

// ricostruisce l'insieme degli indici dato il valore massimale
int solution(int[] DP, int[] D, int i)
    // i: indice di scorrimento
    if i == 0 then // nessun donatore
        return ∅
    else if i == 1 then// un solo donatore
        return {1}
    else if DP[i] == DP[i-1] then // se non c'è variazione fra valori consecutivi

        return solution(DP, D, i-1) // scarto l'indice
    else // c'è variazione fra valori consecutivi
        SET sol = solution(DP, D, i-2) // chiamo ricorsivamente l'algoritmo sull'indice i-2
        sol.insert(i) // inserisco l'indice nell'insieme

        // restituisco l'insieme degli indici
        return sol
```

Effettuo prima la chiamata ricorsiva e poi l'inserimento dell'indice all'interno dell'insieme così alla fine dell'esecuzione gli indici saranno nell'ordine corretto.

Analisi della complessità La complessità computazionale di solution è $T(n) = \Theta(n)$, quella spaziale è $S(n) = \Theta(n)$.

Nota. Non è possibile migliorare la complessità spaziale di hateville poiché è necessario ricostruire la soluzione.

13.3 Zaino

Definizione informale del problema Dato un insieme di oggetti, ognuno caratterizzato da un *peso* ed un *profitto*, e uno "zaino" con un limite di capacità, individuare un sottoinsieme di oggetti il cui peso sia inferiore alla capacità dello zaino e in cui il valore totale degli oggetti sia massimale, ossia il più alto o uguale al valore di qualunque altro sottoinsieme di oggetti.

Definizione formale del problema Dati un vettore w, dove $w[i]$ è il **peso** (*weight*) dell'oggetto i-esimo, un vettore p, dove $p[i]$ è il **profitto** (*profit*) dell'oggetto i-esimo, e la **capacità** C dello zaino. Bisogna trovare un insieme $S \subseteq \{1, \ldots, n\}$ tale che:

- il **valore totale** deve essere minore o uguale alla capacità;
$$w(S) = \sum_{i \in S} w[i] \leqslant C$$
- il **profitto totale** deve essere massimizzato.
$$p(S) = \sum_{i \in S} p[i]$$

Esempi di esecuzione Con un vettore dei pesi $w = [10, 4, 8]$ ed un vettore dei profitti $p = [20, 6, 12]$ ed una capacità $C = 12$. L'insieme degli indici da restituire è $S = \{1\}$. Mentre con un vettore dei pesi $w = [10, 4, 8]$ ed un vettore dei profitti $p = [20, 7, 15]$ ed una capacità sempre pari a $C = 12$. L'insieme degli indici da restituire è $S = \{2, 3\}$.

Definizione matematica del valore della soluzione

Dato uno zaino di capacità C e n oggetti caratterizzati da peso w e profitto p, definiamo $DP[i][c]$ come il massimo profitto che può essere ottenuto dai primi $i \leqslant n$ oggetti contenuti in uno zaino di capacità $c \leqslant C$. Il massimo profitto ottenibile dal problema originale è rappresentato da $DP[n][C]$.

Passo ricorsivo

Andiamo per passi. Consideriamo l'oggetto i-esimo.

- cosa succede se **non prendo** quell'oggetto? La capacità non cambia e non c'è profitto. Avanziamo con l'indice $(i-1)$.
$$DP[i][c] = DP[i-1][c]$$
- cosa succede se **prendo** quell'oggetto? Sottraiamo il peso dalla capacità $(c - w[i])$ e aggiungiamo il relativo profitto $(+p[i])$. Avanziamo con l'indice $(i-1)$.
$$DP[i][c] = DP[i-1][c - w[i]] + p[i]$$
- a questo punto come faccio a **decidere** quale delle due opzioni scegliere? Semplicemente prendo quello che mi dà un guadagno maggiore. In simboli:
$$DP[i][c] = \mathsf{max}(DP[i-1][c - w[i]] + p[i], DP[i-1][c])$$

Completare la ricorsione

Ragioniamo sui casi base. Se non abbiamo più oggetti o se abbiamo finito la capacità dello zaino allora il nostro guadagno sarà 0. E nel caso prendessimo un oggetto la nostra capacità diventasse negativa? In quel caso mettiamo come valore convenzionale $-\infty$.

Possiamo quindi scrivere la formula per calcolare il massimo guadagno dati i oggetti ed una capacità c:

$$DP[i][c] = \begin{cases} 0 & i = 0 \textbf{ or } c = 0 \\ -\infty & c < 0 \\ \mathsf{max}(DP[i-1][c - w[i]] + p[i], DP[i-1][c])) & \text{altrimenti} \end{cases}$$

13.3.1 Algoritmo iterativo

Dalla formula possiamo ricavare il seguente algoritmo iterativo.

Algoritmo 6: Algoritmo *iterativo* per la soluzione al proble

```
int knapsack(int[] w, int[] p, int n, int C)
    // w: vettore dei pesi
    // p: vettore dei profitti
    // n: numero di oggetti
    // C: capacità massima dello zaino
    // creo la tabella di programmazione dinamica
    DP ← new int[0...n][0...C]
    // la inizializzo
    from i ← 0 until n do
        DP[i][0] = 0 // capacità nulla
    from c ← 0 until C do
        DP[0][c] = 0 // nessun oggetto
    // calcolo caso per caso
    from i ← 1 until n do
        from c ← 1 until C do
            if w[i] ⩽ c then // se la capacità residua è
                DP[i][c] = max(DP[i-1][c-w[i]] +
                  scartarlo
            else
                DP[i][c] = DP[i-1][c] // lo scarto
    // restituisco il profitto massimo
    return DP[n][C]
```

Esempio di esecuzione Con un vettore dei pesi $w = [4, 2, 3, 4]$ e un vettore dei profitti $p = [10, 7, 8, 6]$ ed una capacità di $C = 9$, la tabella di programmazione generata è la seguente:

i \ c	**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

I valori in **grassetto** indicato numero di oggetti (i) e la capacità (c) dello zaino, mentre i valori in *corsivo* sono i valori inizializzati della tabella.

La tabella viene riempita per righe. Per capire come funziona l'algoritmo prendiamo in considerazione la seconda riga che corrisponde alla possibilità di prendere $i = 2$ oggetti: una volta raggiunta la capacità di $c = 2$ possiamo prendere l'oggetto con peso 7, una volta raggiunta la capacità di $c = 4$ devo scegliere il massimo profitto prendere l'oggetto 10 ($DP[i-1][c-w[i]]+p[i]$) e 7, ossia ignorarlo $DP[i-1][c]$. Quando raggiungo la capacità di $c = 6$ posso prendere entrambi gli oggetti per un profitto totale di 17.

13.3.2 Algoritmo ricorsivo

Complessità La complessità di knapsack è $\Theta(nC)$. Osserviamo che C è parte dell'input (non è la dimensione del problema). Applicchiamo quindi il criterio di costo logaritmico: per rappresentare C sono necessari $k = \log_2 C$ bit, quindi la complessità è $T(n) = \mathcal{O}(n2^k)$. knapsack è un algoritmo esponenziale.

Algoritmo 7: Algoritmo *ricorsivo* per la soluzione al problema dello zaino

```
// metodo wrapper
int knapsack(int[] w, int[] p, int n, int C)
    return knapsackRec(w, p, n, C)

int knapsackRec(int[] w, int[] p, int n, int c)
    // c: capacità residua
    if c < 0 then
        return −∞
    else if i == 0 or c == 0 then
        return 0
    else
        int nottaken ← knapsackRec(w, p, i − 1, c)
        int taken ← knapsackRec(w, p, i − 1, c − w[i]) +p[i]
        return max(nottaken, taken)
```

Analisi della complessità La funzione ricorsiva scaturita dalla funzione knapsack è la seguente:

$$T(n) = \begin{cases} 1 & n \leqslant 1 \\ 2T(n-1)+1 & n > 1 \end{cases} = \mathcal{O}(2^n)$$

Non si può fare meglio di così.

13.3.3 Memoization

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

Prendiamo l'esempio precedente.

i \ c	**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**

I valori segnati in **rosso** sono gli unici necessari alla computazione della soluzione.

La *memoization* (annotazione) è una tecnica che fonde l'approccio di *memorizzazione* della programmazione dinamica con l'approccio *top-down* di divide-et-impera.

Quando devo risolvere un sottoproblema prima controllo se l'ho già risolto, guardando nella cella corrispondente della tabella dove memorizzo i risultati, altrimenti lo calcolo al momento (*on-the-fly*) chiamando

ricorsivamente i sottoproblemi e scrivendo i rispettivi risultati nella tabella. In questo modo mi assicuro di fare il calcolo una volta sola ed evito di calcolare valori che non verranno mai usati.

Per indicare che il problema non è ancora stato risolto la inizializzo ad un valore speciale (-1).

Algoritmo 8: Zaino con memoization

```
// funzione wrapper
int knapsack(int[] w, int[] p, int n, int C)
    DP ← new int[1...n][1...C]
    from i ← 1 until n do
        from c ← 1 until C do
            DP[i][c] = −1 // non ho ancora risolto questo sotto-problema
    return knapsackRec(w, p, n, C, DP)
int knapsackRec(int[] w, int[] p, int n, int c, int[][] DP)
    if c < 0 then
        return −∞
    else if i == 0 or c == 0 then
        return 0
    else
        if DP[i][c] < 0 then
            int nottaken ← knapsackRec(w, p, i − 1, c, DP)
            int taken ← knapsackRec(w, p, i − 1, c − w[i], DP) +p[i]
            DP[i][c] ← max(nottaken, taken)
        return DP[i][c]
```

La tabella viene inizializzata esternamente, nella funzione *wrapper*, con un valore che non viene usato durante la procedura (-1 nel caso vengano usati *solo* valori positivi, $-\infty$ nel caso vengano usati sia valori positivi che valori negativi).

```python
def knapsackRec(w, p, i, c, DP):
    if c < 0:
        return -math.inf
    elif i == 0 or c == 0:
        return 0
    else:
        if DP[i][c] < 0:
            nottaken = knapsackRec(w, p, i-1, c, DP)
            taken = knapsackRec(w, p, i-1, c-w[i-1], DP) + p[i-1]
            DP[i][c] = max(nottaken, taken)
            return DP[i][c]

def knapsack(w,p,C):
    n = len(w)
    DP = [[-1]*(C+1) for i in range(n+1)]
    return knapsackRec(w,p,n,C,DP)
```

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

Scelta della struttura dati È meglio scegliere una tabella o un dizionario?

Il costo di inizializzazione della tabella è pari a $\mathcal{O}(nC)$. Applicata in questo modo, non c'è alcun vantaggio nell'utilizzare la tecnica di *memoization*. Permette tuttavia di tradurre in fretta le espressioni ricorsive.

Se invece di usare una tabella utilizzassimo un dizionario non dovremmo pagare il costo d'inizializzazione. Il costo di esecuzione è pari a $\mathcal{O}(min(2^n, nC))$.

```python
def knapsack(w,p,C):
    n = len(w)
    DP = {} # Hash-table dictionary
    return knapsackRec(w,p,n,C,DP)

def knapsackRec(w, p, i, c, DP):
    if c < 0:
        return -math.inf
    elif i == 0 or c == 0:
        return 0
    else:
        if not (i,c) in DP: # la soluzione non è contenuta nell'insieme delle soluzioni
        nottaken = knapsackRec(w, p, i-1, c, DP)
        taken = knapsackRec(w, p, i-1, c-w[i-1], DP) + p[i-1]
        DP[i,c] = max(nottaken, taken)
        return DP[i,c]
```

Codice 13.3: Implementazione di zaino con dizionario in Python

```python
from functools import wraps

def memo(func):
    cache = {}
    @wraps(func)
    def wrap(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrap

@memo
def knapsackRec(w, p, i, c):
    if c < 0:
        return -math.inf
    elif i == 0 or c == 0:
        return 0
    else:
        nottaken = knapsackRec(w, p, i-1, c)
        taken = knapsackRec(w, p, i-1, c-w[i-1]) + p[i-1]
        return max(nottaken, taken)
```

```python
def knapsack(w, p, C):
    n = len(w)
    return knapsackRec(w, p, n, C)
```

Codice 13.4: Implementazione di zaino con dizionario in Python con annotazione automatica

Nota. La ricostruzione della soluzione in Python è lasciata come esercizio.

13.4 Zaino senza limiti

Con *senza limiti* ci si riferisce alla scelta degli oggetti.

Definizione del problema Dato uno zaino di capacità C e n oggetti caratterizzati da peso w e profitto p, definiamo $DP[i][c]$ come il massimo profitto che può essere ottenuto dai primi $i \leqslant n$ oggetti contenuti in uno zaino di capacità $c \leqslant C$, *senza porre limiti al numero di volte che un oggetto può essere selezionato*.

Definizione matematica del valore della soluzione

Come modifichiamo la formula ricorsiva?

$$DP[i][c] = \begin{cases} 0 & i = 0 \textbf{ or } c = 0 \\ -\infty & c < 0 \\ \mathsf{max}(DP[i-1][c-w[i]] + p[i], DP[i-1][c]) & \text{altrimenti} \end{cases}$$

In un caso come questo, è possibile semplificare la formula riducendo lo spazio occupato.

Variante della soluzione

Dato uno zaino senza limiti di scelta di capacità C e n oggetti caratterizzati da peso w e profitto p, definiamo $DP[c]$ come il massimo profitto che può essere ottenuto da tali oggetti in uno zaino di capacità $c \leqslant C$.

$$DP[c] = \begin{cases} 0 & c = 0 \\ max_{w[i] \leqslant c}\{DP[c-w[i]] + p[i]\} & c > 0 \end{cases}$$

Algoritmo 9: Zaino senza limiti con *memoization*

```
// funzione wrapper
int knapsack(int[] w, int[] p, int n, int C)
    int[] DP ← new int[0...C]
    from i ← 0 until C do
        DP[i] = −1
    return knapsackRec(w, p, n, C, DP)
    return DP[C]
int knapsackRec(int[] w, int[] p, int n, int c, int[] DP)
    if c == 0 then
        return 0
    if DP[c] < 0 then
        DP[c] ← 0
        from i ← 1 until n do
            if w[i] ⩽ c then
                int val ← knapsackRec(w, p, n, c − w[i], DP) + p[i]
                DP[c] ← max(DP[c], val)
    return DP[c]
```

Complessità Non è detto che tutti gli elementi debbano essere riempiti. La complessità in spazio è pari a $\Theta(C)$. Questo approccio rende più difficile ricostruire la soluzione. Possiamo ispezionare tutti gli elementi

per capire da dove deriva il massimo. Conviene tuttavia memorizzare l'indice da cui deriva il massimo.

Algoritmo 10: Zaino senza limiti con *memoization* e ricostruzione della soluzione

```
// funzione wrapper
int knapsack(int[] w, int[] p, int n, int C)
    int[] DP ← new int[0...C]
    int[] pos ← new int[0...C] // vettori delle posizioni
    from i ← 0 until C do
        DP[i] = −1
        pos[i] = −1 // lo inizializzo
    knapsackRec(w, p, n, C, DP, pos)
    return solution(w, C, pos)

// calcolo dei valori
int knapsackRec(int[] w, int[] p, int n, int c, int[] DP, int[] pos)
    if c == 0 then
        return 0
    if DP[c] < 0 then
        DP[c] ← 0
        from i ← 1 until n do
            if w[i] ⩽ c then
                int val ← knapsackRec(w, p, n, c − w[i], DP, pos) + p[i]
                if val ⩾ DP[c] then
                    DP[c] ← val
                    pos[c] ← i
    return DP[c]

// ricostruzione della soluzione
solution(int[] w, int c, int[] pos)
    if c == 0 or pos[c] < 0 then
        return List()
    else
        List L ← solution(w, c − w[pos[c]], pos)
        L.insert(L.head(), pos[c])
        return L
```