Ottimizzazione con la Ricorsione

Prof. Simone Faro
Dispensa didattica per il corso di Algoritmi e Laboratorio
Università di Catania

Anno Accademico 2025–2026

Indice

1 Il problema dello zaino

La ricorsione è uno strumento naturale per affrontare problemi di ottimizzazione in cui una scelta locale induce problemi residui della stessa natura. Tra questi, il *Problema dello Zaino* (Knapsack) è un classico: dato un insieme di oggetti, ciascuno con un valore e un peso, si vuole scegliere un sottoinsieme che massimizza il valore totale senza superare una capacità massima K.

Per comprendere intuitivamente il significato del problema, immaginiamo la scena di un racconto d'altri tempi. È notte fonda al museo del Louvre, e il celebre ladro *Arsenio Lupin* si è introdotto furtivamente in una sala colma di tesori. Davanti a lui si trovano gioielli, quadri, statuette, monete antiche: ogni oggetto ha un certo valore e un certo peso. Lupin non può portare via tutto — il suo zaino ha una capienza limitata, che indicheremo con K, e ogni scelta sbagliata potrebbe impedirgli di portare con sé gli oggetti più preziosi. Il suo obiettivo è dunque chiaro: riempire lo zaino nel modo migliore possibile, selezionando gli oggetti che gli garantiscono il massimo guadagno complessivo senza superare la capacità massima.

In questa prospettiva, il problema dello zaino diventa una sfida di equilibrio tra *costo* e *beneficio*: ogni decisione di includere un oggetto implica la rinuncia a un altro. È proprio qui che l'approccio ricorsivo mostra la sua forza: Lupin può considerare, a ogni passo, due possibilità — prendere un oggetto oppure lasciarlo — e ridurre così il problema a una versione più piccola di sé stesso. In questo modo, l'arte del furto perfetto si trasforma in un processo logico e rigoroso, dove la ricorsione diventa lo strumento per esplorare tutte le combinazioni possibili e scoprire la migliore strategia per massimizzare il bottino.

1.1 Versione 0.1: pesi unitari, valori diversi

Immaginiamo che Lupin si trovi in una sala del museo dove tutti gli oggetti esposti — monete d'oro, piccoli anelli, gemme e pietre preziose — abbiano lo stesso peso, ma valori differenti. Il suo zaino può contenere solo un numero limitato di pezzi, pari alla capacità K. Ogni oggetto occupa lo stesso spazio, ma non tutti hanno lo stesso valore: alcune gemme valgono poco, altre moltissimo. Fortunatamente, gli oggetti sono già disposti in ordine crescente di valore, dal meno prezioso al più prezioso. Questa semplice condizione cambia radicalmente il problema, poiché permette a Lupin di evitare ogni decisione complessa: è evidente che conviene riempire lo zaino partendo dagli oggetti più preziosi finché c'è spazio.

Questa situazione rappresenta la forma più semplice del *Problema dello Zaino*. Poiché tutti gli oggetti hanno lo stesso peso, la scelta ottima consiste nel prendere gli ultimi K oggetti della lista, quelli di valore più alto, oppure tutti gli oggetti se il numero totale è inferiore a K. L'approccio ricorsivo, in questo caso, può essere sostituito da una strategia *greedy* (golosa), poiché ogni decisione locale è anche globalmente ottimale: prendere un oggetto più prezioso al posto di uno meno prezioso migliora sempre il valore totale senza aumentare il peso complessivo.

Formalizzazione. Abbiamo n oggetti ordinati per valore crescente:

$$v_1 \leq v_2 \leq \cdots \leq v_n,$$

tutti di peso unitario. Lo zaino ha capacità K e può quindi contenere al massimo K oggetti. Vogliamo massimizzare il valore complessivo:

$$\max\Big\{\sum_{i\in S} v_i \;:\; S\subseteq\{1,\dots,n\},\ |S|\le K\Big\}.$$

Poiché i pesi sono unitari e i valori ordinati, la soluzione ottima è semplicemente:

$$S^* = \{n-K+1, n-K+2, \dots, n\}, \quad V^* = \sum_{i=n-K+1}^{n} v_i.$$

Il problema si risolve in un'unica scansione lineare degli ultimi K elementi, sommando i valori. Il tempo di esecuzione è $O(K)$ (e dunque $O(n)$ nel caso peggiore), poiché non sono necessarie né scelte condizionali né chiamate ricorsive.

```
int zainoLineare(int v[], int n, int K) {
    int somma = 0;
    for (int i = n - K; i < n; i++) {
        if (i >= 0) somma += v[i];
    }
    return somma;
}
```

Il procedimento è concettualmente semplice ma didatticamente rilevante, perché evidenzia un principio generale: quando le scelte locali non influenzano negativamente le successive (come in questo caso, in cui tutti i pesi sono uguali), la soluzione ottimale può essere trovata in modo diretto, senza necessità di esplorare lo spazio delle combinazioni.

Si osservi che se gli oggetti non fossero ordinati, si dovrebbe prima ordinarli in tempo $O(n \log n)$, ma anche in questo caso il passo di selezione finale resterebbe lineare. L'approccio ricorsivo non è necessario, ma rappresenta comunque un modello utile per comprendere come l'ottimalità globale possa emergere naturalmente da una struttura di decisioni semplici.

Possiamo comunque esprimere la stessa logica in forma ricorsiva, riducendo progressivamente il problema al sottoproblema con un elemento in meno e una capacità ridotta:

$$F(i,k) = \begin{cases} 0 & \text{se } k=0 \text{ o } i=0, \\ v_i + F(i-1,k-1) & \text{se } k>0. \end{cases}$$

Poiché gli oggetti sono già ordinati, non è necessario scegliere se includere o meno l'oggetto i: se c'è ancora spazio nello zaino ($k>0$), lo si prende automaticamente.

```
int F(int v[], int i, int k) {
    if (i == 0 || k == 0)
        return 0;                       // caso base
    return v[i-1] + F(v, i-1, k-1);     // riduzione diretta
}
```

La funzione viene inizialmente chiamata con $F(v, n, K)$ e termina dopo K chiamate ricorsive, ognuna delle quali aggiunge il valore di un elemento allo zaino virtuale di Lupin. Poiché ogni chiamata riduce il numero di oggetti e la capacità di uno, la profondità massima della ricorsione è K, e la complessità rimane lineare $O(K)$.
Questo esempio mostra come un algoritmo iterativo possa essere riscritto in forma ricorsiva anche quando la ricorsione non offre vantaggi in termini di efficienza. Ciò permette di comprendere meglio la struttura logica della ricorsione, in cui ogni chiamata rappresenta un "passo" della costruzione progressiva della soluzione. In questo caso, la ricorsione modella la naturale sequenza con cui Lupin aggiunge uno dopo l'altro gli oggetti più preziosi fino a riempire completamente il suo zaino.

1.2 Versione generale 0/1: pesi e valori variabili

Nella sala successiva del museo, la situazione di Lupin si fa più complessa. Ora gli oggetti non sono più simili tra loro: accanto a un piccolo anello di diamanti, leggero ma prezioso, c'è una statua di bronzo pesante ma di valore inferiore, e accanto a un delicato orologio d'oro, un antico quadro incorniciato. Ogni oggetto ha dunque il proprio peso w_i e un valore v_i differente. Lo zaino di Lupin ha sempre la stessa capacità K, ma scegliere cosa portare con sé diventa una questione di equilibrio tra valore e peso.

La strategia semplice della versione precedente — scegliere i K oggetti più preziosi — non funziona più: un oggetto di grande valore ma troppo pesante potrebbe impedire di portare con sé più oggetti leggeri complessivamente più vantaggiosi. In questa situazione, Lupin deve valutare tutte le combinazioni possibili: ogni oggetto può essere incluso o escluso, e la ricorsione diventa uno strumento fondamentale per esplorare sistematicamente tutte le scelte e individuare quella che massimizza il valore complessivo del bottino, rispettando il vincolo di capacità.

Formalizzazione. Per ogni oggetto i abbiamo *peso* $w_i \in \mathbb{N}_{>0}$ e *valore* $v_i \geq 0$; la capacità massima dello zaino è $K \in \mathbb{N}$. Vogliamo massimizzare

$$\max\Big\{\sum_{i\in S} v_i \;:\; S \subseteq \{1,\dots,n\},\ \sum_{i\in S} w_i \leq K\Big\}.$$

La strategia greedy non è più corretta in generale: occorre confrontare sistematicamente le scelte *includo/escludo*.

Definiamo $F(i, k)$ come prima: valore massimo usando solo $\{1, \dots, i\}$ con capacità residua k. I casi base sono:

$$F(i, k) = 0 \text{ se } i = 0 \text{ o } k = 0; \qquad F(i, k) = -\infty \text{ se } k < 0.$$

Il valore $-\infty$ (nella pratica, una costante molto negativa) serve a *squalificare* ricorsioni che superano la capacità. La ricorrenza:

$$F(i, k) \;=\; \max\big\{\, F(i-1, k),\ v_i + F(i-1, k-w_i) \,\big\}.$$

Rispetto alla versione 0.1, l'inclusione di i riduce la capacità di w_i.

Si osservi che se una soluzione ottima non include i, allora è ottima su $(i{-}1, k)$. Se include i, allora la soluzione residua è ottima su $(i{-}1, k - w_i)$. La massimizzazione tra i due casi produce il valore ottimo. Il caso $k < 0$ deve essere reso impossibile per impedire che soluzioni *infeasible* vengano scelte.

```
int F(int i, int k, int v[], int w[]) {
    if (k < 0)  return INT_MIN/4;    // rappresenta -infinito
    if (i == 0 || k == 0) return 0;
    int senza = F(i-1, k, v, w);
    int con   = v[i] + F(i-1, k - w[i], v, w);
    return (senza > con) ? senza : con;
}
```

La ricorsione, da sola, restituisce il *valore ottimo*. Per ottenere una soluzione concreta (l'elenco degli oggetti), si usa il confronto locale: se $F(i,k) = v_i + F(i-1, k-w_i)$ allora l'oggetto i è preso; altrimenti è escluso. Questo semplice criterio consente di ricostruire una soluzione ottima in tempo $O(n)$ dopo aver calcolato la tabella.

La versione 0.1 evidenzia come l'*ottimalità locale* (prendo i K valori maggiori) coincida con l'ottimo globale; la versione 0/1 generale mostra invece che occorre esplorare sistematicamente le alternative *includo/escludo*. In entrambi i casi, la ricorsione rende esplicita la struttura del problema e, con la memoizzazione, porta a un algoritmo efficiente dal punto di vista asintotico.

2 Esercizi sulla ricorsione e l'ottimizzazione

La ricorsione è uno strumento potente per affrontare problemi di ottimizzazione, in cui l'obiettivo è massimizzare o minimizzare una certa funzione di costo soggetta a vincoli. In questi problemi, la soluzione ottimale può spesso essere vista come il risultato della combinazione di soluzioni ottimali di sottoproblemi più piccoli. Gli esercizi seguenti mostrano come formulare ricorsioni efficaci e come interpretare la struttura del problema per individuare la strategia risolutiva.

Ciascuno di questi esercizi può essere affrontato seguendo la stessa logica generale:

- individuare i *casi base* che rendono la ricorsione terminante;
- determinare la *scomposizione del problema* in sottoproblemi più piccoli;
- stabilire una regola per *combinare le soluzioni parziali*

2.1 Problemi di ottimizzazione su array

Molti problemi di ottimizzazione possono essere formulati in termini di array, dove gli elementi rappresentano valori, costi o pesi associati a decisioni successive. La ricorsione si rivela uno strumento efficace per analizzare questi problemi, poiché ciascuna decisione su un elemento dell'array influenza in modo deterministico i sottoproblemi restanti. I seguenti esercizi propongono situazioni di natura diversa, ma tutte accomunate dall'idea di dover scegliere una sequenza ottimale di azioni o valori.

1. **(Somma massima di elementi non adiacenti)** Dato un array di numeri positivi $A = [a_1, a_2, \ldots, a_n]$, si desidera selezionare un sottoinsieme di elementi in modo che nessun elemento scelto sia adiacente a un altro. L'obiettivo è massimizzare la somma dei valori selezionati:
$$\max\Big\{\sum_{i\in S} a_i : S \subseteq \{1,\ldots,n\},\ |i-j|>1 \text{ per ogni } i,j \in S\Big\}.$$
Questo problema, noto come *maximum non-adjacent sum*, modella situazioni in cui la scelta di un elemento esclude automaticamente i suoi vicini.

2. **(Divisione ottimale di un array)** Dato un array $A = [a_1, a_2, \ldots, a_n]$, si vuole dividerlo in due segmenti contigui $A_1 = [a_1, \ldots, a_k]$ e $A_2 = [a_{k+1}, \ldots, a_n]$ in modo da massimizzare la differenza tra la somma dei valori nella parte sinistra e quella nella parte destra:
$$\max_{1\le k<n}\Big|\sum_{i=1}^{k} a_i - \sum_{i=k+1}^{n} a_i\Big|.$$
Il problema consiste nel trovare il punto di separazione k che genera il maggiore squilibrio tra le due metà dell'array.

3. **(Sottosequenza crescente di somma massima)** Dato un array di interi $A = [a_1, a_2, \ldots, a_n]$, si vuole determinare la somma massima ottenibile da una sottosequenza strettamente crescente. Formalmente:
$$\max_{i_1<i_2<\cdots<i_k} \sum_{j=1}^{k} a_{i_j} \quad \text{tale che } a_{i_1} < a_{i_2} < \cdots < a_{i_k}.$$

Questo problema richiede di bilanciare l'ordine crescente con la massimizzazione del valore totale.

4. **(Intervallo di somma minima)** Dato un array $A = [a_1, a_2, \ldots, a_n]$, si desidera trovare il sottointervallo contiguo la cui somma è la più piccola possibile:

$$\min_{1 \le i \le j \le n} \sum_{k=i}^{j} a_k.$$

L'obiettivo è individuare la regione "più negativa" dell'array, utile per analisi di costi, deviazioni o perdite localizzate.

5. **(Bilanciamento di carichi)** Un array $A = [a_1, a_2, \ldots, a_n]$ rappresenta il carico di n server. Si dispone di una risorsa di bilanciamento C che consente di ridurre o spostare parte dei carichi. Si desidera minimizzare la differenza tra il carico massimo e quello minimo dopo un numero limitato di operazioni. Formalmente:

$$\min\Big\{\max_i a'_i - \min_i a'_i : a'_i \text{ ottenuti da redistribuzioni entro limite } C\Big\}.$$

Il problema richiede di individuare una sequenza ottimale di redistribuzioni per livellare i carichi.

6. **(Massimizzazione della produttività)** Ogni elemento a_i di un array rappresenta la produttività giornaliera di una macchina. Una macchina non può essere utilizzata per più di due giorni consecutivi. Si vuole scegliere i giorni di utilizzo per massimizzare la somma della produttività complessiva. Il problema consiste nel trovare l'insieme di indici S che massimizza $\sum_{i \in S} a_i$ sotto il vincolo che nessuna tripla di indici consecutivi appartenga a S.

7. **(Suddivisione in segmenti bilanciati)** Dato un array $A = [a_1, \ldots, a_n]$ e un intero k, si desidera suddividere l'array in k segmenti contigui in modo che la somma massima di un segmento sia la più piccola possibile:

$$\min_{\text{partizioni in } k \text{ segmenti}} \max_{1 \le r \le k} \sum_{i \in \text{segmento}_r} a_i.$$

Il problema è tipico della pianificazione del lavoro su più processori o della distribuzione di carichi.

8. **(Rimozione ottimale di elementi)** Dato un array $A = [a_1, a_2, \ldots, a_n]$, si vogliono rimuovere esattamente r elementi in modo da massimizzare la somma dei valori rimanenti, mantenendo la loro posizione relativa:

$$\max_{S \subseteq \{1,\ldots,n\}, |S| = n-r} \sum_{i \in S} a_i.$$

L'obiettivo è scegliere quali elementi eliminare per ottenere il miglior valore complessivo residuo.

9. **(Scalatore di array)** Si consideri un array di interi positivi $A = [a_1, a_2, \ldots, a_n]$, dove ogni a_i rappresenta l'altezza di un gradino. Uno scalatore può muoversi da i a $i + 1$ o saltare a $i + 2$, ma il costo del movimento è proporzionale alla differenza

di altezza tra i gradini. Si vuole trovare il costo minimo per raggiungere l'ultimo gradino partendo dal primo:

$$\min_{\text{sequenze valide}} \sum |a_{i_{k+1}} - a_{i_k}|.$$

Il problema è un modello astratto di ottimizzazione di percorsi con vincoli di adiacenza.

10. **(Taglio ottimale di un array)** Dato un array di numeri positivi $A = [a_1, \ldots, a_n]$ e un numero K, si vuole dividere l'array in sottosequenze contigue in modo che la somma dei valori massimi di ciascun sottosegmento sia minima. In altre parole:

$$\min_{\text{partizioni in } K \text{ segmenti}} \sum_{r=1}^{K} \max_{i \in \text{segmento}_r} a_i.$$

Il problema combina decisioni di segmentazione e minimizzazione dei picchi locali.

2.2 Problemi di ottimizzazione su stringhe

Le stringhe costituiscono una delle strutture più studiate in informatica teorica e algoritmica, poiché permettono di rappresentare in modo compatto sequenze di simboli, caratteri o eventi. Molti problemi di ottimizzazione riguardano il confronto tra stringhe, la ricerca di pattern, o la trasformazione di una stringa in un'altra con il minimo costo possibile. La ricorsione si rivela uno strumento naturale per questi problemi, perché la struttura di una stringa è, per sua stessa natura, ricorsiva: ogni stringa può essere vista come un carattere iniziale seguito da una sottostringa più corta. Gli esercizi che seguono descrivono alcuni dei problemi classici di ottimizzazione su stringhe.

1. **(Distanza di edit minima)** Date due stringhe A e B, si vuole determinare il numero minimo di operazioni necessarie per trasformare A in B. Le operazioni ammesse sono: inserimento, cancellazione e sostituzione di un carattere. Si desidera trovare:

$$\min\{\text{numero di operazioni per trasformare } A \text{ in } B\}.$$

Questo problema misura la *distanza di Levenshtein* tra le due stringhe e rappresenta un modello fondamentale per l'analisi delle somiglianze testuali.

2. **(Sottosequenza comune massima)** Date due stringhe A e B, si vuole determinare la lunghezza massima di una sequenza di caratteri che appare in entrambe nello stesso ordine, ma non necessariamente in posizioni contigue. Formalmente:

$$\max\{|S| : S \text{ è una sottosequenza di } A \text{ e di } B\}.$$

Il problema richiede di identificare il grado di somiglianza strutturale tra le due stringhe.

3. **(Sottostringa comune massima)** Date due stringhe A e B, si desidera trovare la lunghezza massima di una sottostringa contigua che appare in entrambe. A differenza del caso precedente, i caratteri devono comparire in posizioni consecutive:

$$\max\{|S| : S \text{ è sottostringa di } A \text{ e di } B\}.$$

Questo problema ha applicazioni dirette nell'allineamento di sequenze biologiche e nella ricerca di pattern comuni.

4. **(Allineamento ottimale di sequenze)** Date due stringhe A e B, si vuole allinearle inserendo spazi (simboli di gap) in modo da massimizzare un punteggio di similarità complessivo. A ogni coppia di simboli (a, b) si assegna un punteggio $s(a, b)$, mentre un gap comporta una penalità g. L'obiettivo è determinare:

$$\max\Big\{\sum s(a_i, b_i) - g \cdot (\text{numero di gap})\Big\}.$$

Questo problema generalizza il concetto di distanza di edit ed è alla base dell'allineamento di sequenze in bioinformatica.

5. **(Palindromo più lungo)** Data una stringa A, si vuole determinare la lunghezza del palindromo più lungo contenuto al suo interno, cioè la più lunga sottostringa che si legge uguale in entrambe le direzioni:

$$\max\{|S| : S \text{ è sottostringa di } A,\ S = S^R\}.$$

Questo problema mette in evidenza la simmetria intrinseca delle stringhe e può essere visto come un caso di ottimizzazione strutturale.

6. **(Numero minimo di partizioni palindromiche)** Data una stringa A, si vuole suddividerla nel minor numero possibile di sottostringhe, ciascuna delle quali è un palindromo. L'obiettivo è minimizzare il numero di tagli necessari:

$$\min\Big\{k : A = P_1 P_2 \dots P_k,\ P_i = P_i^R\ \forall i\Big\}.$$

Questo problema combina la ricorsione sulla posizione dei tagli con la verifica delle proprietà palindromiche delle sottostringhe.

7. **(Combinazione di stringhe con costo minimo)** Si considerino due stringhe A e B e una funzione di costo $c(a, b)$ che rappresenta il costo di unire i caratteri a e b (ad esempio, la loro differenza di codice ASCII). Si vuole determinare un modo di interleaving (fusione) delle due stringhe che minimizzi il costo totale di combinazione:

$$\min_{\text{interleave}(A,B)} \sum c(a_i, b_j).$$

Il problema è utile per modellare la fusione di dati o segnali con disallineamento.

8. **(Sottosequenza con bilanciamento massimo)** Data una stringa binaria $A \in \{0, 1\}^n$, si desidera trovare la sottosequenza con il massimo bilanciamento tra zeri e uni, cioè quella in cui la differenza assoluta tra il numero di 0 e 1 è minima, e la lunghezza complessiva è massima. Formalmente:

$$\max\{|S| : S \text{ è sottosequenza di } A,\ ||S_0| - |S_1|| \text{ è minimo}\}.$$

Questo problema si collega alla nozione di equilibrio e simmetria in sequenze binarie.

9. **(Stringa di copertura minima)** Dato un insieme di stringhe $S = \{s_1, s_2, \dots, s_n\}$, si vuole costruire una singola stringa T che le contenga tutte come sottostringhe, minimizzandone la lunghezza complessiva:

$$\min\{|T| : s_i \text{ è sottostringa di } T \text{ per ogni } i\}.$$

Questo è il problema della *shortest common superstring*, che rappresenta un caso di ottimizzazione combinatoria di grande importanza teorica e pratica.

10. **(Ricostruzione di parola con costi di concatenazione)** Si dispone di un insieme di frammenti di testo $F = \{f_1, f_2, \ldots, f_m\}$, che devono essere concatenati nell'ordine corretto per ottenere una parola o frase completa. A ogni coppia di frammenti (f_i, f_j) è associato un costo di sovrapposizione $o(f_i, f_j)$ che indica quante lettere coincidono nella sovrapposizione. Si vuole determinare l'ordine di concatenazione che massimizza la somma totale delle sovrapposizioni:

$$\max_{\pi \in S_m} \sum_{k=1}^{m-1} o(f_{\pi(k)}, f_{\pi(k+1)}).$$

Il problema richiama, in forma stringente, la logica dei problemi di percorso minimo o massimo nei grafi.

2.3 Problemi di ottimizzazione su alberi binari

Gli alberi binari sono strutture intrinsecamente ricorsive: ogni sottoalbero è anch'esso un albero binario. Questo rende naturale pensare a strategie ricorsive per risolvere problemi di ottimizzazione su alberi. I problemi che seguono si basano su proprietà dei nodi (peso, valore, profondità, cammini) e vincoli strutturali (bilanciamento, discesa, taglio). Ciascuno invita a riflettere su come una scelta locale (prendere o escludere un nodo, confrontare rami) induca sottoproblemi analoghi su figli dell'albero.

1. **(Somma massima di un sottoalbero)** Dato un albero binario in cui ogni nodo u ha un valore $v(u)$ (posso essere positivo, zero o anche negativo), si desidera scegliere un sottoalbero (non necessariamente contenente la radice) che massimizza la somma dei valori dei nodi. In termini: trovare insieme di nodi S tale che l'insieme forma un sottoalbero e

$$\max_{S \text{ sottoalbero}} \sum_{u \in S} v(u).$$

2. **(Cammino di somma massima)** In un albero binario pesato, ogni nodo u ha un valore $v(u)$. Si vuole trovare un cammino (da un nodo qualsiasi a un altro) che massimizzi la somma dei valori dei nodi attraversati:

$$\max_{\text{cammini } u \to v} \sum_{x \text{ su cammino}} v(x).$$

3. **(Albero di copertura con vincolo di profondità)** Dato un albero binario radicato con valori $v(u)$ e un parametro limite di profondità D, si vogliono scegliere nodi tali che nessuna foglia selezionata superi profondità D, massimizzando la somma dei valori. Formalmente: selezionare nodo radice ed eventualmente sottorami, vincolando le altezze dei rami selezionati.

4. **(Taglio ottimale di rami)** In un albero binario, ogni nodo ha un valore $v(u)$ e un costo di "taglio" $c(u)$ se si decide di non includere quel ramo nel calcolo. Si desidera decidere quali archi troncare (ossia non esplorare quel sottoalbero) per massimizzare:

$$\sum_{u \text{ lasciati}} v(u) - \sum_{u \text{ tagliati}} c(u),$$

con la condizione che se un nodo è tagliato, tutti i suoi discendenti sono esclusi.

5. **(Sottostruttura equilibrata)** In un albero binario con valori $v(u)$, si vuole selezionare un sottoalbero bilanciato (numero di nodi simile tra figli) che massimizza la somma dei valori:
$$\max_{T' \subseteq T, T' \text{ bilanciato}} \sum_{u \in T'} v(u).$$

6. **(Cammino con vincolo di lunghezza)** Ogni nodo u ha valore $v(u)$. Si vuole trovare un cammino (da radice a qualche nodo) di lunghezza al più L che massimizza la somma:
$$\max_{\text{cammini lunghezza} \leq L} \sum_{u \in \text{cammino}} v(u).$$

7. **(Massima indipendenza con vincoli)** Dato un albero binario e valori $v(u)$, si vuole scegliere un insieme di nodi indipendenti (nessun nodo scelto ha un nodo figlio scelto) che massimizza la somma:
$$\max_{S \text{ indipendente}} \sum_{u \in S} v(u).$$

8. **(Somma massima per livello)** Ogni nodo u ha un valore $v(u)$. Si vuole scegliere un livello h (profondità) dell'albero tale che la somma dei valori dei nodi a profondità h sia massima:
$$\max_{h} \sum_{u : \text{depth}(u) = h} v(u).$$

9. **(Cammino massimo con somma vincolata)** In un albero binario con valori $v(u)$ e con capacità di costo K, ogni nodo ha un "peso" $w(u)$. Si vuole trovare un cammino dalla radice a una foglia, tale che la somma dei pesi non superi K, e la somma dei valori sia massima:
$$\max_{\text{cammini root} \to leaf, \sum w(u) \leq K} \sum v(u).$$

10. **(Rimozione minima per somma ottimale)** Dato un albero binario con valori $v(u)$, si vuole rimuovere un sottoinsieme minimo di nodi tali che la somma dei valori rimanenti soddisfi una soglia T. Formalmente: trovare $S \subseteq V$ di cardinalità minima tale che
$$\sum_{u \in V \setminus S} v(u) \geq T.$$

2.4 Altri problemi di ottimizzazione

1. **(Scelta di investimenti)** Un investitore dispone di un capitale iniziale C e di n progetti tra cui scegliere. Ogni progetto i richiede un investimento p_i e genera un rendimento atteso r_i. Non è possibile finanziare progetti parzialmente. L'obiettivo è selezionare i progetti che massimizzano il rendimento complessivo, mantenendo la somma dei costi entro il capitale disponibile:
$$\max\Big\{\sum_{i \in S} r_i : S \subseteq \{1, \ldots, n\},\ \sum_{i \in S} p_i \leq C\Big\}.$$

2. **(Taglio di una barra di metallo)** Una barra di lunghezza n può essere tagliata in segmenti più piccoli. A ogni lunghezza intera i è associato un valore p_i. Si vuole determinare come suddividere la barra per massimizzare il valore totale dei segmenti ottenuti. Formalmente, si cerca una partizione della lunghezza n in parti $i_1, i_2, \ldots, i_k$ tali che:

$$\sum_{j=1}^{k} i_j = n, \quad \text{e il valore totale } \sum_{j=1}^{k} p_{i_j} \text{ sia massimo.}$$

3. **(Scelta ottima di lavori compatibili)** Si hanno n lavori, ciascuno con tempo di inizio s_i, tempo di fine f_i e guadagno p_i. Due lavori non possono sovrapporsi nel tempo. L'obiettivo è determinare l'insieme di lavori compatibili che massimizza il guadagno complessivo:

$$\max\Big\{\sum_{i\in S} p_i : S \subseteq \{1,\ldots,n\},\ f_i \le s_j \text{ per ogni coppia consecutiva } (i,j)\Big\}.$$

   Il problema rappresenta un caso classico di ottimizzazione con vincoli di compatibilità temporale.

4. **(Cammino a somma massima in una griglia)** Data una matrice M di dimensioni $m \times n$, ogni cella (i, j) contiene un valore $M_{i,j}$. Si parte dalla cella $(1, 1)$ e si vuole raggiungere la cella (m, n) muovendosi solo verso destra o verso il basso. L'obiettivo è massimizzare la somma dei valori lungo il cammino:

$$\max_{\text{cammini da (1,1) a } (m,n)} \sum_{(i,j)\in\text{cammino}} M_{i,j}.$$

5. **(Cambio di monete)** Si dispone di monete di valore $c_1, c_2, \ldots, c_n$. Dato un importo S, si vuole determinare in quanti modi è possibile ottenere esattamente la somma S usando un numero arbitrario di monete. In una seconda variante del problema, si vuole minimizzare il numero di monete necessarie per raggiungere la somma esatta.

6. **(Sottosequenza contigua a somma massima)** Dato un array di interi $A = [a_1, a_2, \ldots, a_n]$, si vuole determinare la somma massima ottenibile da una sottosequenza contigua. Formalmente:

$$\max_{1\le i\le j\le n} \sum_{k=i}^{j} a_k.$$

   Il problema consiste nel individuare i limiti (i, j) che massimizzano la somma dei valori consecutivi.

7. **(Partizionamento bilanciato)** Dati n numeri interi positivi $a_1, a_2, \ldots, a_n$, si vuole stabilire se è possibile dividere l'insieme in due sottoinsiemi S_1 e S_2 tali che le loro somme siano uguali:

$$\sum_{i\in S_1} a_i = \sum_{i\in S_2} a_i.$$

   Si tratta di un problema di decisione che richiede di verificare se esiste una combinazione di elementi che equilibra perfettamente le due somme.

8. **(Cammino minimo in un grafo)** Dato un grafo orientato $G = (V, E)$ con pesi positivi $w(u, v)$ sugli archi, e due nodi $s, t \in V$, si vuole determinare il costo minimo di un cammino da s a t:

$$\min_{\text{cammini da } s \text{ a } t} \sum_{(u,v) \in \text{cammino}} w(u, v).$$

Il problema prevede di individuare un percorso ottimale che minimizza la somma dei pesi, evitando cicli e percorsi non ammissibili.