Corso di laurea in Informatica
Insegnamento di Algoritmi II

Presentazione

ANGELO MONTI

SAPIENZA
UNIVERSITÀ DI ROMA

CORSO DI ALGORITMI 2

PANORAMICA E MOTIVAZIONI

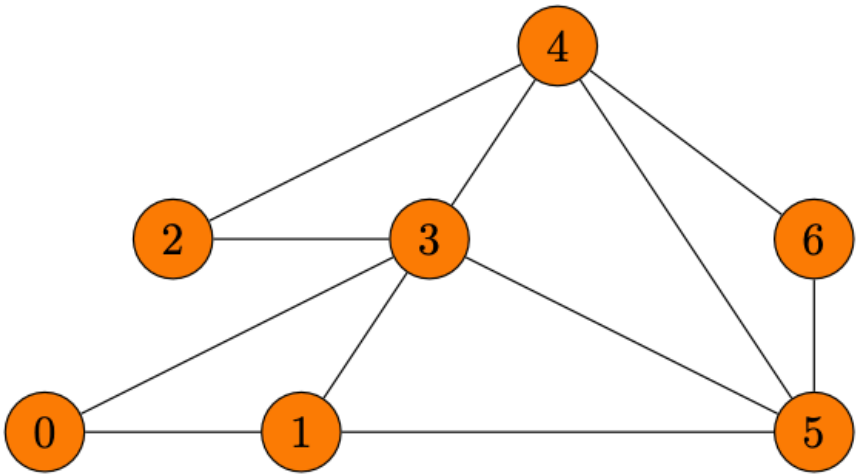
SOLUZIONE EFFICIENTE

Obiettivo: imparare a scegliere l'approccio giusto al problema.

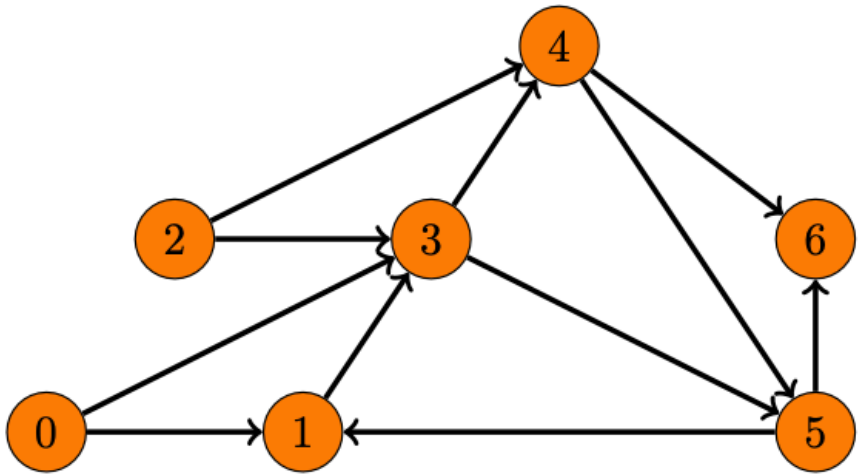
COS’E’ ALGORITMI 2

L’INGRANAGGIO DEL PENSIERO COMPUTAZIONALE

GRAFI: UN LINGUAGGIO UNIVERSALE



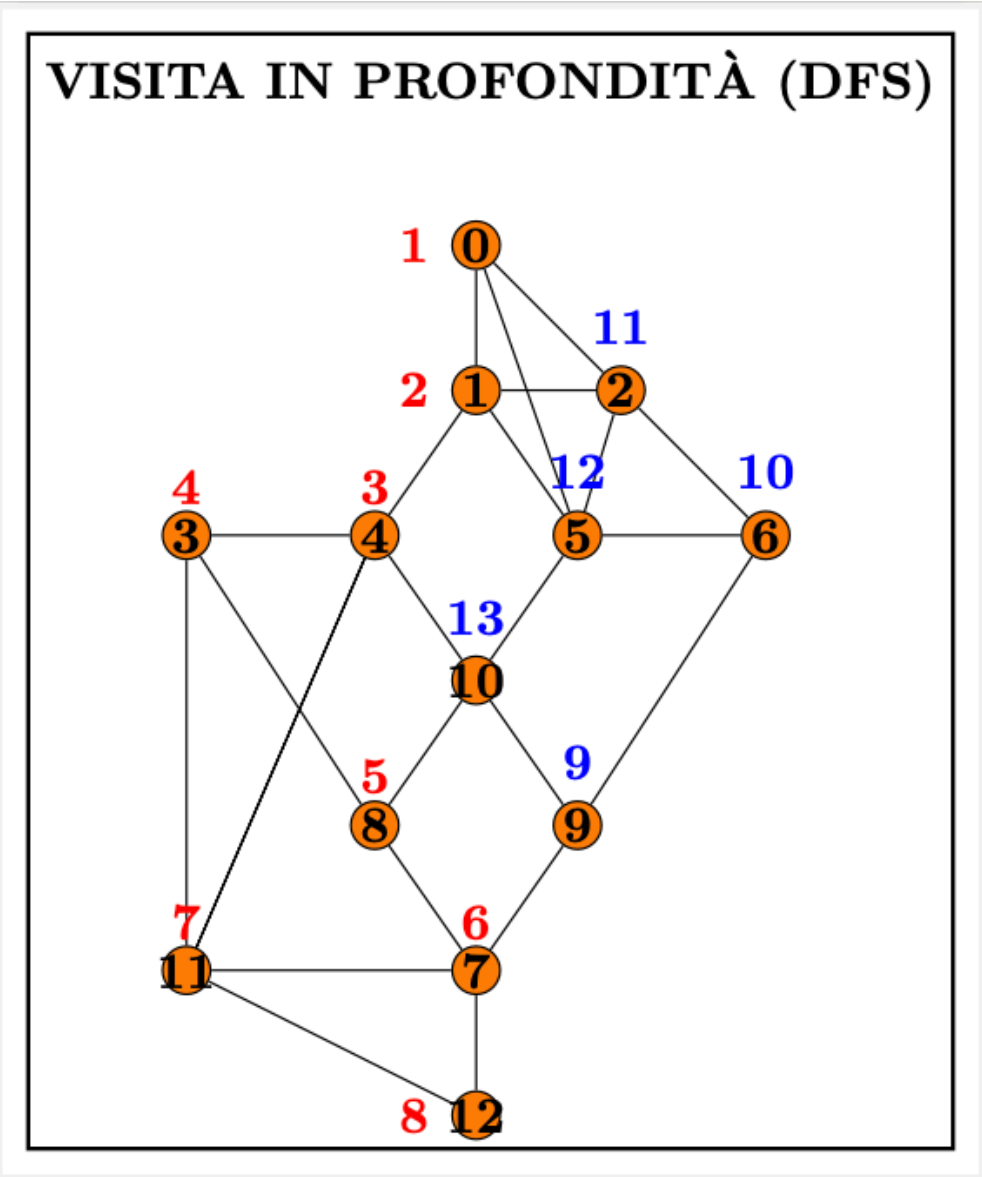
Grafo non orientato

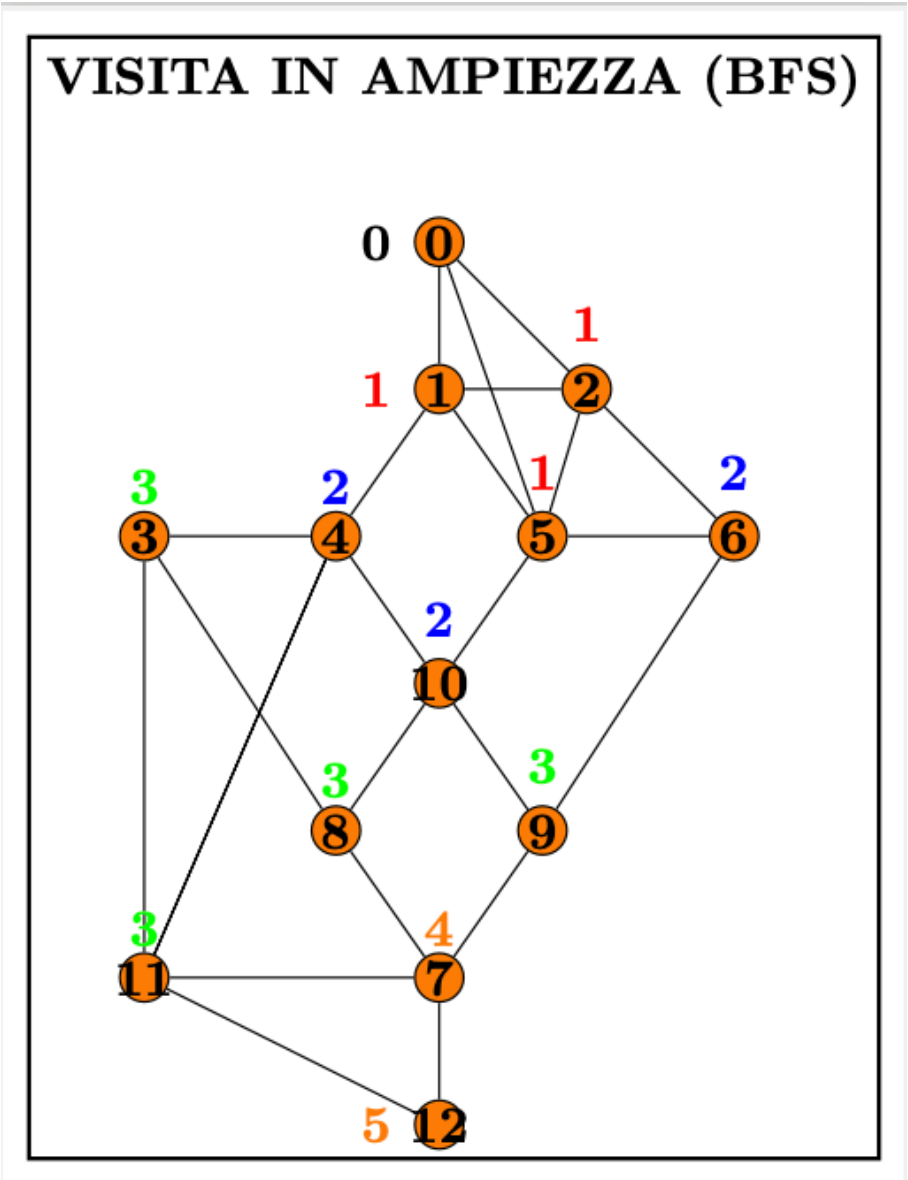


Grafo orientato

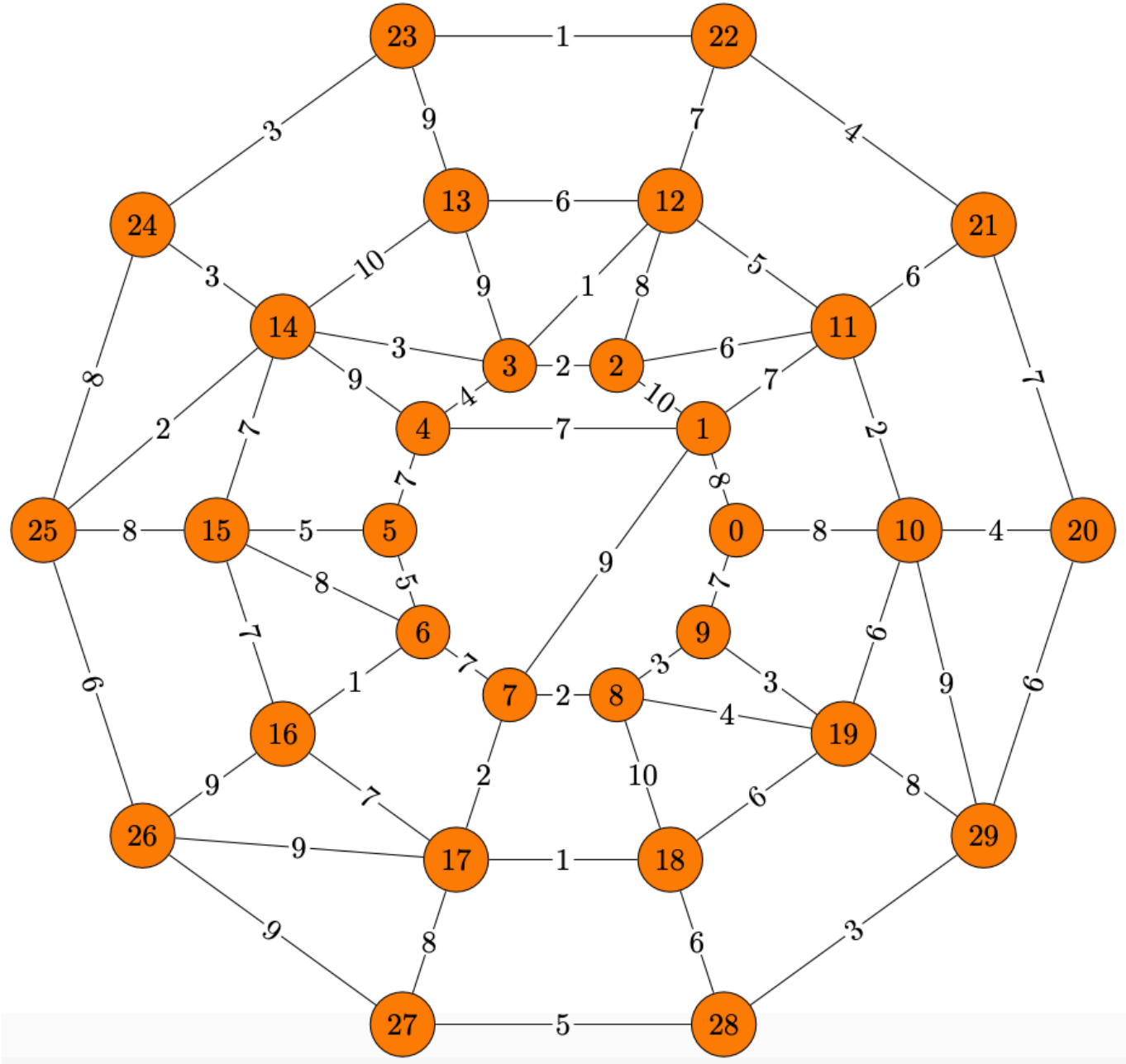
- Reti sociali, reti stradali, reti biologiche

- Due approcci di visita:
  - DFS = Esplorazione in profondità
  - BFS = Esplorazione in ampiezza

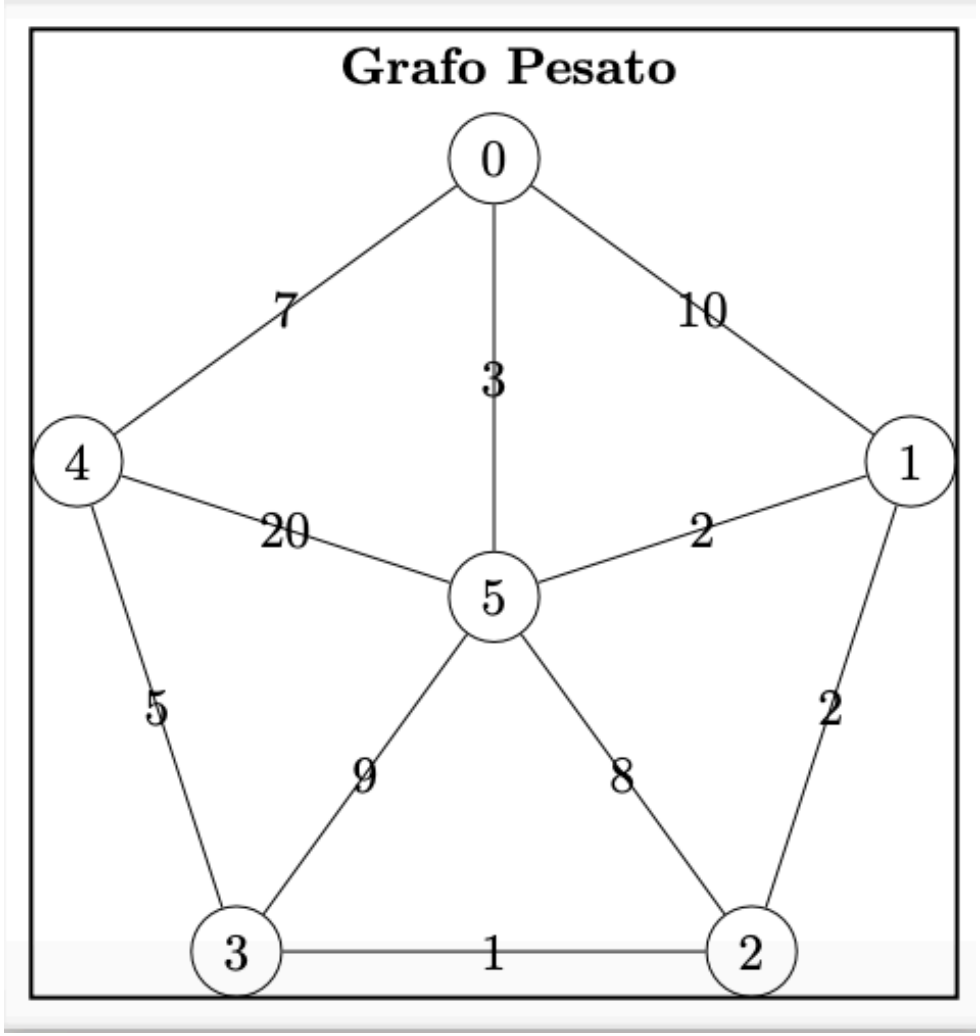


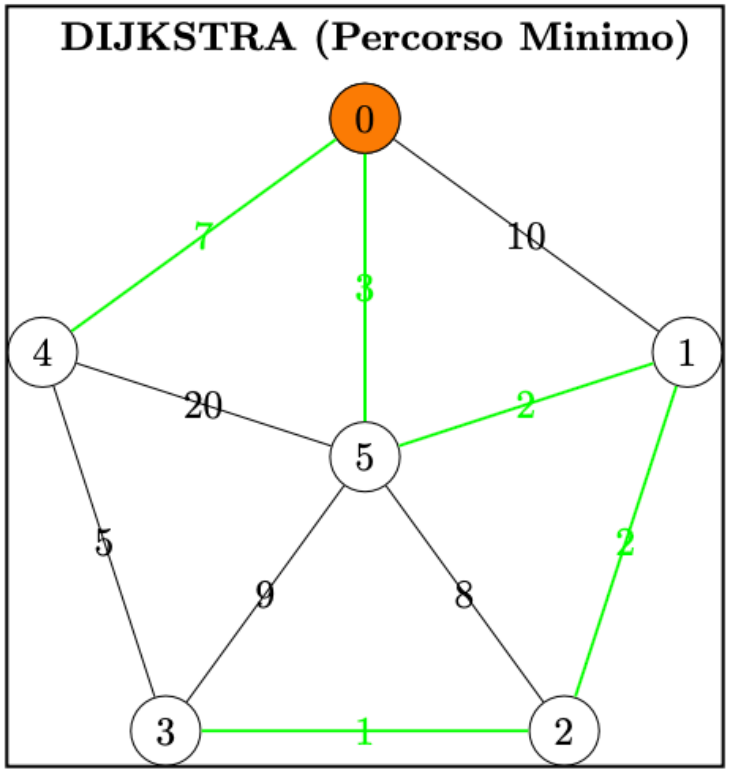


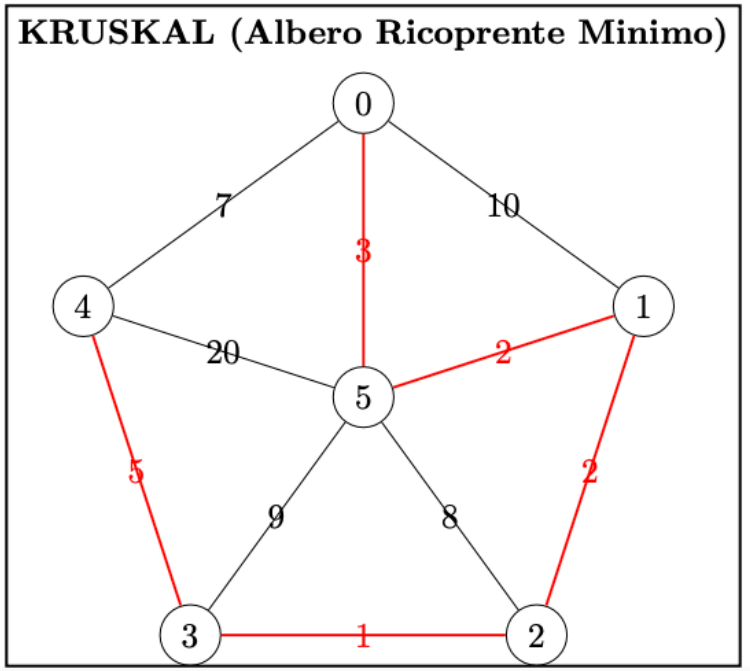
Grafi pesati



- Pesi = costi, distanze tempi
- **Dijkstra:** cammini minimi
- **Kruskal:** reti di connessione a costo minimo







Tecniche Algoritmiche Trasversali

Scelta Locale Ottimale

Scomponi e Ricomponi

Costruisci passo Dopo passo

Esplora e Torna Indietro

Tecnica Greedy: dalle decisioni locali alla soluzione globale

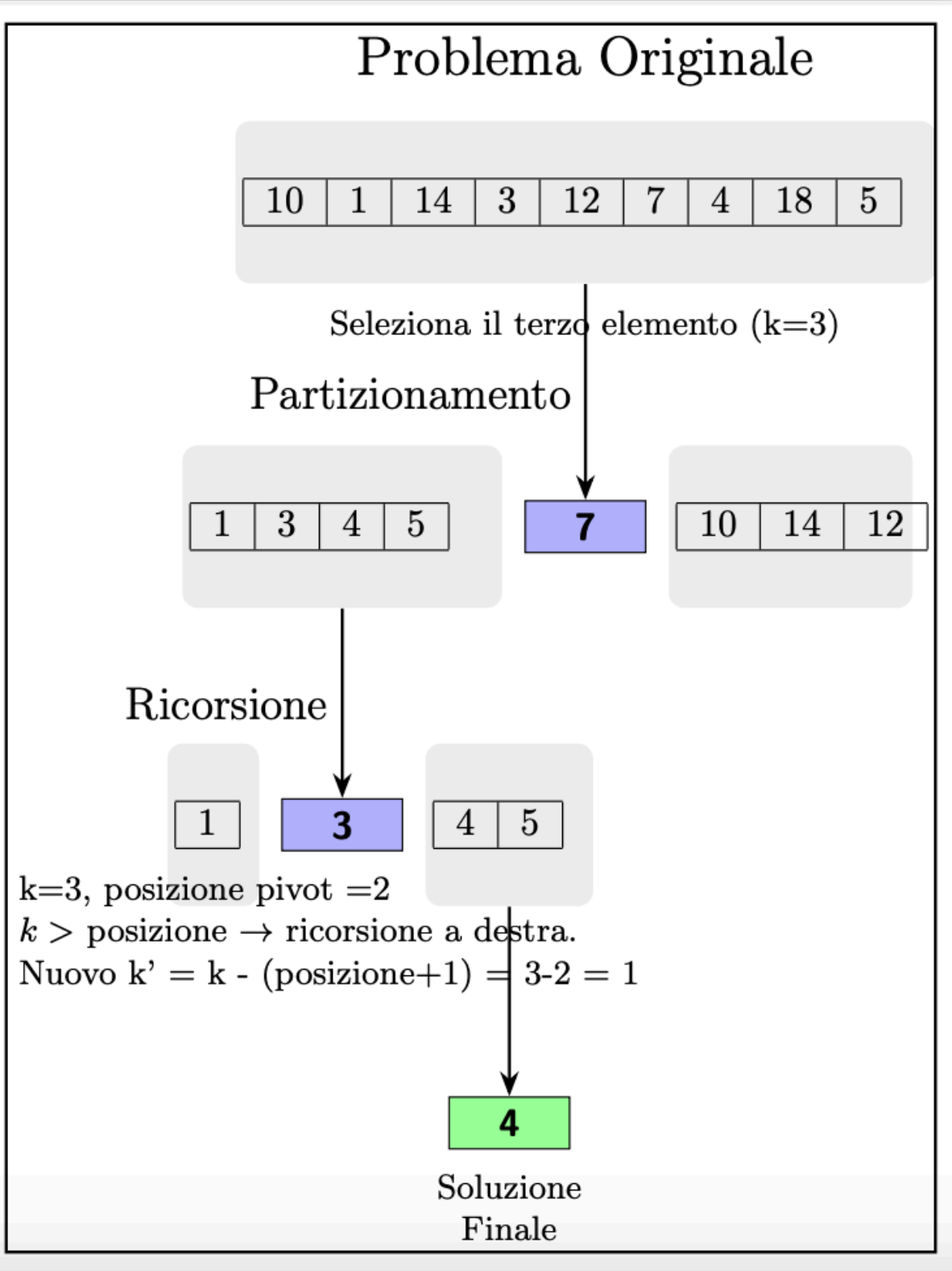
ESEMPIO: Massimizzazione del numero di file in un hard disk

Logica: scegli sempre il file che occupa meno spazio

Divide et impera

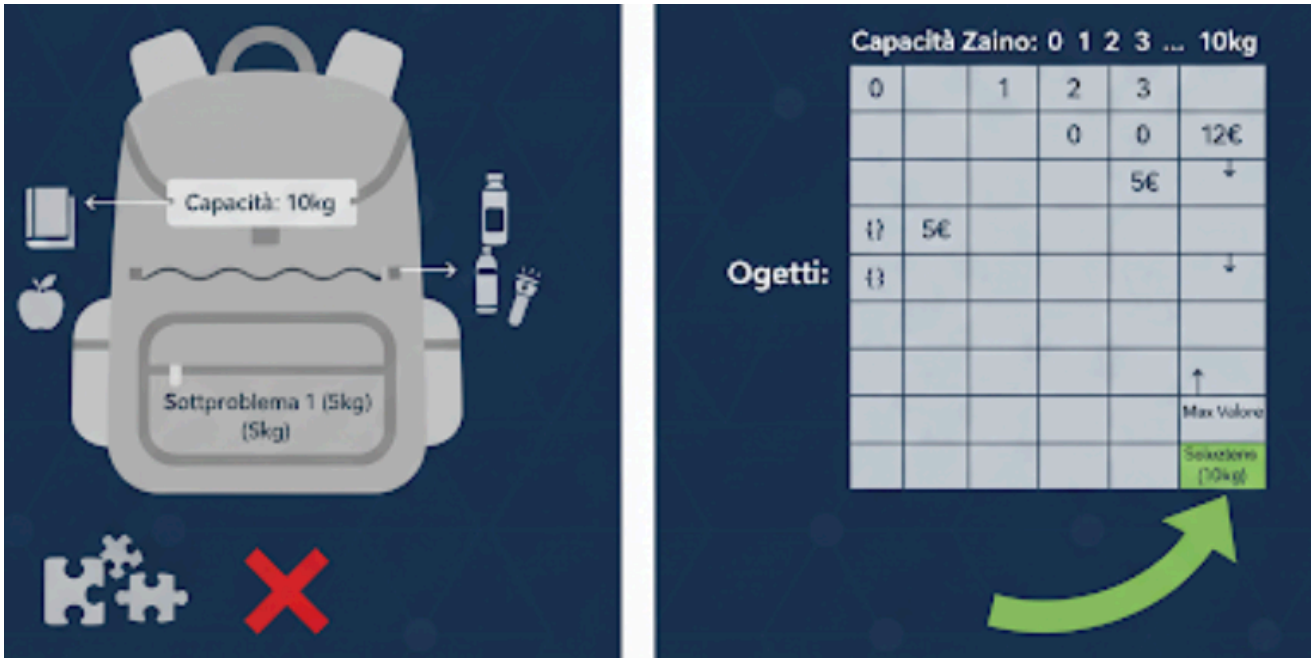
ESEMPIO: In un insieme di n interi selezionare quello che finirebbe al k-mo posto a seguito dell'ordinamento.

- Se $k = 1$ cerco il minimo $O(n)$.
- Se $k = n$ cerco il massimo $O(n)$.
- Per k generico $= O(????)$.



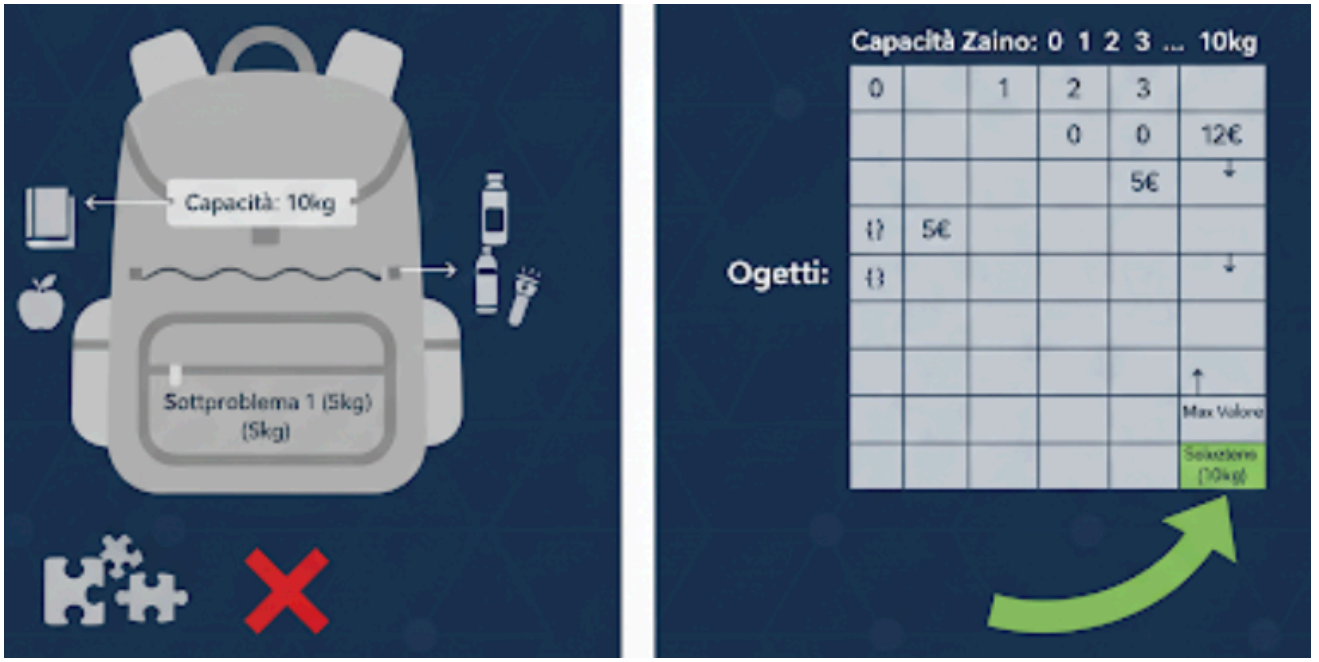
Programmazione Dinamica (1/2)

ESEMPIO: il problema dello zaino, dove ho uno zaino di capacità C ed un insieme di n oggetti, ognuno con un suo peso ed un suo valore, vogliamo selezionare un sottoinsieme degli oggetti di massimo valore il cui peso non eccede la capacità dello zaino.



- **Il divide et impera non funziona:** Se dividi gli oggetti in due sottoinsiemi (A e B) e cerchi la soluzione ottima per ogni sottoinsieme separatamente, la somma delle due soluzioni parziali potrebbe non essere la soluzione ottima per il problema completo.

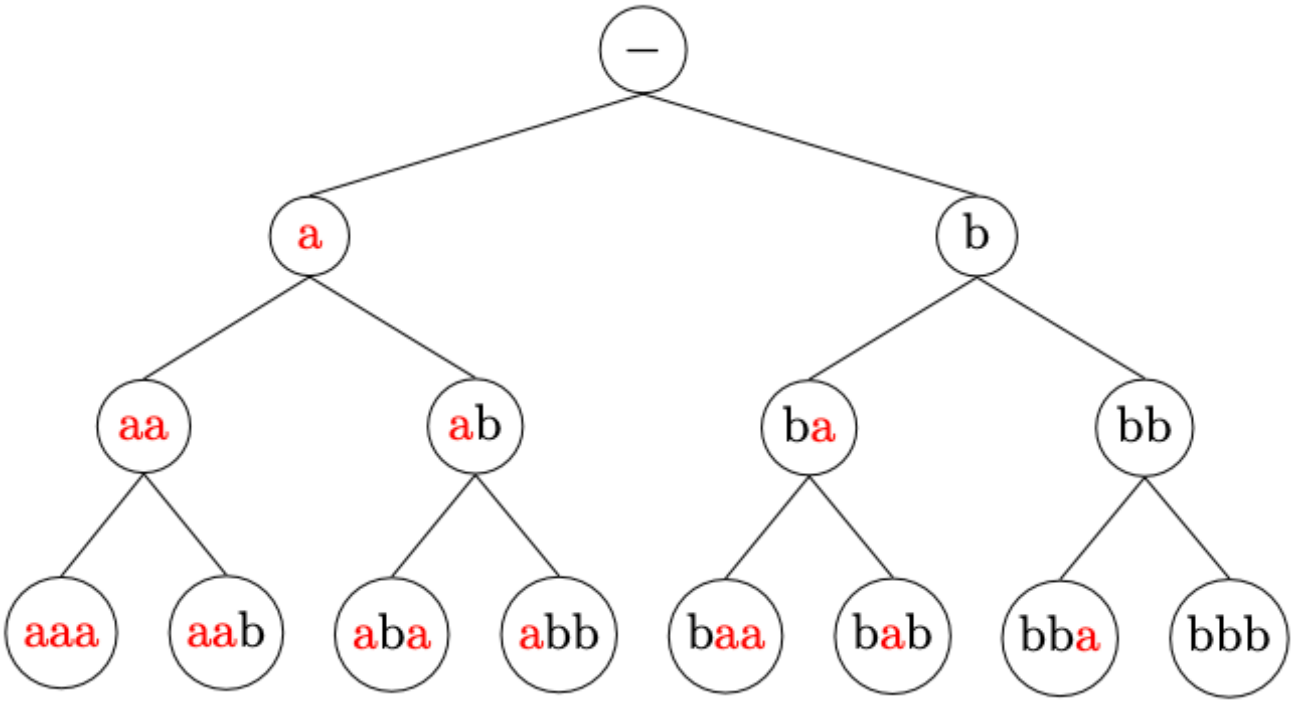
rogrammazione Dinamica (2/2)



- La **Programmazione Dinamica** risolve questo problema partendo dai sottoproblemi più piccoli e banali per costruire la soluzione finale.
  - Invece di dividere gli oggetti, si creano dei sottoproblemi in base alla **capacità dello zaino** e al **numero di oggetti** considerati. Si inizia con uno zaino piccolo (o con pochi oggetti) e si calcola la soluzione ottima per quella configurazione. Poi, si estende progressivamente il problema risolvendo sottoproblemi leggermente più grandi, riutilizzando le soluzioni già trovate.

Backtracking: enumerazione

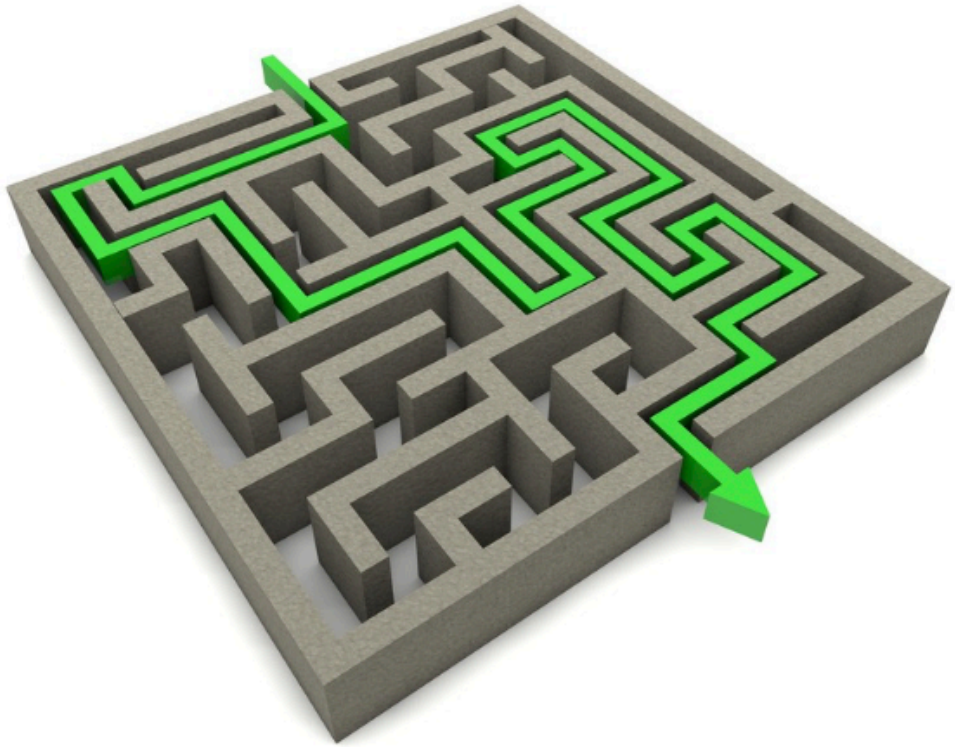
ESEMPIO: contare le stringhe binarie lunghe n



- Il backtracking e la **ricerca esaustiva**

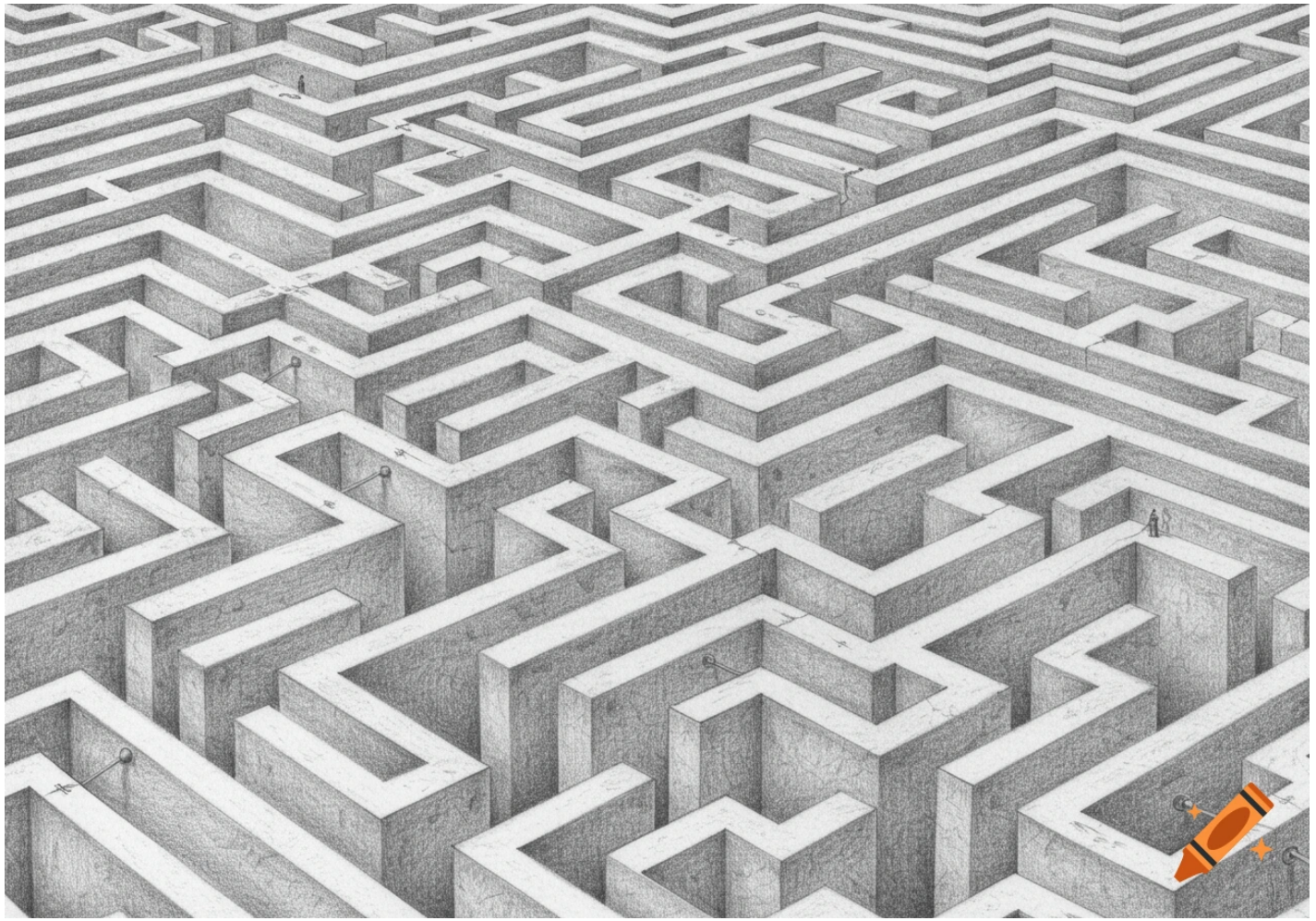
- L'enumerazione con backtracking produce in genere alberi di ricerca enormi. Introducendo vincoli e **funzioni di taglio**, si potano rami improduttivi, riducendo l'esplorazione e trovando più velocemente le soluzioni desiderate.

Backtracking: ricerca di una soluzione



Problemi Difficili

- Non tutti i problemi hanno soluzioni efficienti
- Importanza di riconoscere la difficoltà intrinseca



Non sempre il limite è l’algoritmo a volte è la difficoltà intrinseca del problema

Problemi difficili: Approssimazioni ed Euristiche

Che fare quando non si conoscono algoritmi efficienti?

- **Euristiche:** strategie pratiche quando non c'è l'algoritmo efficiente

- **Algoritmi approssimanti:** soluzioni vicine all'ottimo

ESERCIZI

Suggerimento: Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto. Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

Il Problema della Maggioranza Assoluta.

Data una lista A di n interi, bisogna determinare se in A esiste un elemento di **maggioranza assoluta**, cioè un elemento che compare più di $\lfloor n/2 \rfloor$ volte. Se tale elemento esiste, va restituito; altrimenti l'algoritmo deve restituire $None$.

La soluzione proposta deve avere complessità $O(n)$.

Soluzione:

```python
def esercizio1(A):
    B = []
    for x in A:
        if B != [] and B[-1] != x:
            # rimuove una coppia di elementi distinti
            B.pop()
        else:
            B.append(x)
    # verifica finale del candidato
    if B and A.count(B[0]) > len(A)//2:
        return B[0]
    return None
```