Algoritmi & Strutture Dati

Fondamenti Teorici e Applicazioni Pratiche

15 novembre 2025

Indice

Prefazione

L'arte degli algoritmi

L'informatica moderna è fondata su due pilastri fondamentali: gli **algoritmi** e le **strutture dati**. Un algoritmo è una sequenza finita di istruzioni ben definite che, dato un input, produce un output risolvendo un problema computazionale. Le strutture dati, d'altra parte, sono modi organizzati di memorizzare e gestire i dati in modo che possano essere utilizzati efficacemente.

La relazione tra questi due concetti è profonda e inscindibile: un algoritmo efficiente richiede strutture dati appropriate, e viceversa, strutture dati sofisticate hanno senso solo nel contesto di algoritmi che le sfruttano adeguatamente.

Scopo di questo testo

Questo libro nasce con l'obiettivo di fornire una trattazione rigorosa ma accessibile dei fondamenti teorici e pratici degli algoritmi e delle strutture dati. Il testo è strutturato per essere utilizzato sia come manuale di studio per corsi universitari, sia come riferimento per professionisti che desiderano approfondire le proprie conoscenze.

Gli obiettivi principali sono:

1. **Rigore matematico**: Ogni concetto è introdotto con definizioni formali e, quando appropriato, corredato da teoremi e dimostrazioni. La matematica non è un ornamento, ma lo strumento essenziale per comprendere le proprietà degli algoritmi.
2. **Analisi di complessità**: Un algoritmo non è "buono" o "cattivo" in astratto, ma in relazione al suo costo computazionale. Dedicheremo ampio spazio all'analisi asintotica e ai diversi scenari (caso migliore, medio, peggiore).
3. **Implementabilità**: Ogni struttura dati e algoritmo è presentato con pseudocodice dettagliato, facilmente traducibile in linguaggi di programmazione reali.
4. **Visualizzazione**: Diagrammi, alberi e grafi sono rappresentati graficamente usando TikZ per facilitare la comprensione intuitiva dei concetti.
5. **Applicazioni pratiche**: Oltre alla teoria, vengono discusse le applicazioni reali e i casi d'uso di ogni struttura dati e algoritmo presentato.

Prerequisiti

Per trarre il massimo beneficio da questo testo, il lettore dovrebbe possedere:

- **Matematica di base**: Familiarità con algebra elementare, logaritmi, esponenziali, e nozioni di base di teoria degli insiemi.
- **Matematica discreta**: Conoscenza di elementi di logica matematica, tecniche di dimostrazione (induzione, dimostrazione per assurdo), combinatoria elementare.

- **Programmazione**: Esperienza con almeno un linguaggio di programmazione imperativo o orientato agli oggetti. La conoscenza di concetti come variabili, cicli, condizionali, funzioni e ricorsione è essenziale.
- **Curiosità intellettuale**: Forse il prerequisito più importante. Gli algoritmi sono oggetti di grande bellezza matematica, e lo studio di questo campo richiede pazienza e desiderio di comprendere in profondità.

Notazione matematica

In questo testo utilizzeremo una notazione matematica standard. Ecco alcune convenzioni importanti:

- $\mathbb{N}$ denota l'insieme dei numeri naturali $\{0, 1, 2, 3, \ldots\}$
- $\mathbb{Z}$ denota l'insieme dei numeri interi $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- $\mathbb{R}$ denota l'insieme dei numeri reali
- $\mathbb{R}^+$ denota l'insieme dei numeri reali positivi
- $\lg n$ denota il logaritmo in base 2 di n (logaritmo binario)
- $\ln n$ denota il logaritmo naturale di n (base e)
- $\log n$ denota generalmente $\lg n$ in contesti algoritmici
- $\lfloor x \rfloor$ denota la parte intera inferiore di x (floor)
- $\lceil x \rceil$ denota la parte intera superiore di x (ceiling)
- $n!$ denota il fattoriale di n: $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$
- $\binom{n}{k}$ denota il coefficiente binomiale: $\frac{n!}{k!(n-k)!}$

Per le sommatorie e i prodotti utilizzeremo:

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) + \cdots + f(n)$$

$$\prod_{i=1}^{n} f(i) = f(1) \cdot f(2) \cdots f(n)$$

Alcune sommatorie importanti che ricorreranno frequentemente:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \Theta(n^3)$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 = \Theta(2^n)$$

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1} \quad \text{per } r \neq 1$$

$$\sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1) \quad \text{(serie armonica)}$$

Pseudocodice

Gli algoritmi in questo testo sono presentati usando uno pseudocodice che mescola elementi di notazione matematica e costrutti di programmazione imperativi. Lo pseudocodice è progettato per essere:

- **Leggibile**: Deve essere comprensibile anche a chi non conosce un linguaggio di programmazione specifico.
- **Preciso**: Deve specificare esattamente cosa fa l'algoritmo, senza ambiguità.
- **Traducibile**: Deve poter essere facilmente convertito in codice eseguibile.

Convenzioni dello pseudocodice:

- L'**indentazione** indica la struttura dei blocchi
- `if-then-else`, `while`, `for` hanno il significato usuale
- `return` termina l'esecuzione e restituisce un valore
- Gli **array** sono indicizzati da 1 (salvo diversa indicazione)
- L'accesso a un elemento dell'array A di indice i è scritto come $A[i]$
- L'assegnamento è indicato con $\leftarrow$
- Il confronto di uguaglianza è indicato con $=$
- I commenti sono preceduti da `//`

Esempio di pseudocodice:

```
def SommaArray(A, n):
    """
    Calcola la somma degli elementi di un array
    Input: array A di n elementi
    Output: somma degli elementi
    """
    somma = 0
    for i = 1 to n:
        somma = somma + A[i]
    return somma
```

Struttura del libro

Il testo è organizzato in capitoli progressivi, ciascuno dei quali si basa sui concetti introdotti nei precedenti:

Capitolo 1: Introduzione e Complessità Introduciamo le notazioni asintotiche (Big-O, Omega, Theta) e i metodi per analizzare la complessità temporale e spaziale degli algoritmi. Discutiamo i casi migliore, medio e peggiore.

Capitolo 2: Array e Liste Esaminiamo le strutture dati fondamentali: array (statici e dinamici) e liste concatenate (semplici, doppie, circolari). Analizziamo le operazioni di base e le loro complessità.

Capitolo 3: Stack e Code Studiamo due strutture dati lineari fondamentali: stack (LIFO) e queue (FIFO), con le loro varianti e applicazioni pratiche come valutazione di espressioni e gestione di processi.

Capitolo 4: Alberi Approfondiamo gli alberi binari, alberi binari di ricerca (BST), alberi AVL (auto-bilancianti) e heap. Analizziamo le operazioni di inserimento, cancellazione e ricerca.

Capitolo 5: Grafi Introduciamo i grafi, le loro rappresentazioni (matrice di adiacenza, lista di adiacenza), e algoritmi fondamentali come BFS, DFS, e algoritmi per cammini minimi (Dijkstra, Bellman-Ford).

Capitolo 6: Tabelle Hash Studiamo le funzioni hash, la gestione delle collisioni tramite concatenamento (chaining) e indirizzamento aperto (open addressing), e l'analisi delle prestazioni.

Capitolo 7: Algoritmi di Ordinamento Analizziamo algoritmi di ordinamento classici: Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, con confronto delle complessità.

Capitolo 8: Algoritmi di Ricerca Studiamo la ricerca lineare e binaria, alberi di ricerca ottimali, e strutture dati avanzate per la ricerca.

Capitolo 9: Ricorsione Approfondiamo la ricorsione, le relazioni di ricorrenza, il Master Theorem, e tecniche per risolvere equazioni di ricorrenza.

Capitolo 10: Programmazione Dinamica Introduciamo il paradigma della programmazione dinamica, con esempi classici come il problema dello zaino, il calcolo di Fibonacci, e la longest common subsequence.

Capitolo 11: Algoritmi Greedy Studiamo gli algoritmi greedy (golosi) e le loro proprietà: scelta greedy, sottostruttura ottima. Esempi: problema dello zaino frazionario, algoritmo di Huffman.

Capitolo 12: Backtracking Esploriamo la tecnica del backtracking per problemi di ricerca esaustiva: N-regine, Sudoku, percorsi in grafi.

Come usare questo libro

Questo testo può essere utilizzato in diversi modi:

- **Corso universitario**: I capitoli sono pensati per coprire un semestre di un corso di Algoritmi e Strutture Dati. Ogni capitolo può corrispondere a 1-2 settimane di lezioni.
- **Studio individuale**: I capitoli sono auto-contenuti e possono essere studiati indipendentemente, anche se è consigliabile seguire l'ordine proposto. Ogni concetto è spiegato da zero.
- **Riferimento rapido**: Le definizioni formali, i teoremi e gli pseudocodici possono servire come riferimento rapido per chi ha già familiarità con gli argomenti.
- **Preparazione a colloqui tecnici**: Le sezioni sulle applicazioni pratiche e gli esercizi possono essere utili per prepararsi a colloqui di lavoro nel settore informatico.

Consigli per lo studio

Studiare algoritmi e strutture dati richiede un approccio attivo. Ecco alcuni consigli:

1. **Non limitarsi a leggere**: Cerca di implementare gli algoritmi in un linguaggio di programmazione. Solo scrivendo codice si comprende davvero il funzionamento degli algoritmi.
2. **Seguire le dimostrazioni**: Le dimostrazioni dei teoremi non sono opzionali. Contengono l'essenza della comprensione. Se una dimostrazione non è chiara, rileggila, scrivila su carta, discutila con altri.
3. **Visualizzare**: Disegna alberi, grafi, traccia l'esecuzione degli algoritmi passo passo. La visualizzazione aiuta enormemente la comprensione.
4. **Fare esercizi**: La comprensione teorica deve essere consolidata con esercizi pratici. Implementa le varianti, risolvi problemi, analizza nuovi algoritmi.
5. **Confrontare alternative**: Per ogni problema, considera diverse soluzioni e confrontale in termini di complessità temporale, spaziale, semplicità di implementazione.
6. **Approfondire**: Questo libro è un'introduzione. Per ogni argomento esistono trattazioni più avanzate. Usa la bibliografia per esplorare ulteriormente.

Bibliografia essenziale

Questo testo si basa su decenni di ricerca e didattica nel campo degli algoritmi. Alcune opere di riferimento fondamentali:

- **Cormen, Leiserson, Rivest, Stein**: *Introduction to Algorithms* (4a ed.). Il testo di riferimento per eccellenza, completo e rigoroso.
- **Knuth, Donald**: *The Art of Computer Programming* (volumi 1-4A). L'opera monumentale che ha definito il campo.
- **Sedgewick, Wayne**: *Algorithms* (4a ed.). Eccellente per le implementazioni pratiche e la visualizzazione.
- **Kleinberg, Tardos**: *Algorithm Design*. Ottimo per le tecniche di progettazione di algoritmi.
- **Skiena, Steven**: *The Algorithm Design Manual* (3a ed.). Ricco di esempi pratici e war stories.
- **Aho, Hopcroft, Ullman**: *Data Structures and Algorithms*. Un classico intramontabile.

Ringraziamenti

Lo studio degli algoritmi è un viaggio intellettuale che non si compie mai in solitudine. Questo testo è il frutto di innumerevoli discussioni, lezioni, letture e riflessioni.

Ringrazio i pionieri del campo—Dijkstra, Knuth, Hoare, Tarjan, Floyd, e tanti altri—le cui idee brillanti sono diventate patrimonio comune dell'informatica. Ringrazio i docenti che hanno reso accessibile questa materia complessa, e gli studenti le cui domande hanno stimolato chiarimenti e approfondimenti.

Questo libro è dedicato a chiunque sia affascinato dalla bellezza matematica degli algoritmi e voglia comprendere come piccole sequenze di istruzioni possano risolvere problemi di enorme complessità.

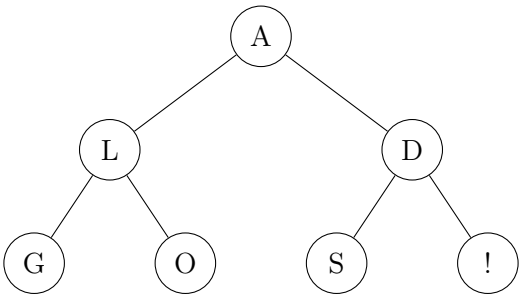
Nota finale

Gli algoritmi non sono solo oggetti di studio accademico. Ogni volta che usiamo un motore di ricerca, un navigatore GPS, un social network, un sistema di raccomandazione, stiamo beneficiando di algoritmi sofisticati che operano dietro le quinte.

Comprendere gli algoritmi significa comprendere i fondamenti del mondo digitale in cui viviamo. È un potere intellettuale che apre porte, sia professionali che culturali.

Buono studio, e benvenuti nel meraviglioso mondo degli algoritmi e delle strutture dati.

L'Autore



Un albero per iniziare il viaggio...

Capitolo 1

Introduzione e Analisi di Complessità

1.1 Introduzione

L'analisi di complessità è il cuore della teoria degli algoritmi. Non basta che un algoritmo sia corretto—deve anche essere **efficiente**. Ma cosa significa efficienza in senso formale? Come possiamo confrontare due algoritmi che risolvono lo stesso problema? Come possiamo prevedere il comportamento di un algoritmo su input di grandi dimensioni?

Queste domande trovano risposta nell'analisi asintotica, uno strumento matematico che ci permette di caratterizzare il comportamento degli algoritmi al crescere della dimensione dell'input.

Definizione 1.1 (Algoritmo)**.** *Un* ***algoritmo*** *è una sequenza finita di istruzioni non ambigue che, dato un input appartenente a un insieme specificato, termina dopo un numero finito di passi producendo un output.*

Le proprietà fondamentali di un algoritmo sono:

- **Input**: Dati iniziali provenienti da un insieme specificato
- **Output**: Risultato prodotto dall'algoritmo
- **Definitezza**: Ogni passo è definito precisamente
- **Finitezza**: L'algoritmo termina dopo un numero finito di passi
- **Efficacia**: Ogni operazione è sufficientemente basilare da poter essere eseguita

1.2 Modello di calcolo

Per analizzare gli algoritmi abbiamo bisogno di un modello di calcolo. Utilizziamo il modello della **macchina RAM** (Random Access Machine):

- Memoria infinita organizzata in celle, ciascuna contenente un numero intero
- Accesso a qualsiasi cella in tempo costante $O(1)$
- Operazioni aritmetiche di base (+, -, *, /, %) in tempo costante
- Operazioni logiche e di confronto in tempo costante
- Un processore che esegue un'istruzione alla volta

Questo modello è un'astrazione semplificata ma ragionevole dei computer reali.

1.3 Complessità temporale e spaziale

Definizione 1.2 (Complessità temporale)**.** *La **complessità temporale** di un algoritmo è una funzione $T(n)$ che rappresenta il numero di operazioni elementari eseguite dall'algoritmo su un input di dimensione n.*

Definizione 1.3 (Complessità spaziale)**.** *La **complessità spaziale** di un algoritmo è una funzione $S(n)$ che rappresenta la quantità di memoria utilizzata dall'algoritmo su un input di dimensione n.*

In questo testo ci concentreremo principalmente sulla complessità temporale, anche se entrambe sono importanti nella pratica.

1.4 Notazioni asintotiche

Le notazioni asintotiche ci permettono di descrivere il comportamento di una funzione per valori grandi del suo argomento, ignorando costanti moltiplicative e termini di ordine inferiore.

1.4.1 Notazione O-grande (Big-O)

Definizione 1.4 (Notazione O-grande)**.** *Date due funzioni $f(n)$ e $g(n)$ da $\mathbb{N}$ a $\mathbb{R}^+$, diciamo che $f(n) = O(g(n))$ se esistono costanti positive c e n_0 tali che:*

$$0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

In altre parole, $f(n) = O(g(n))$ significa che $f(n)$ cresce al più come $g(n)$ a meno di una costante moltiplicativa, per n sufficientemente grande.

La notazione O-grande fornisce un **limite superiore asintotico**.

Esempi:

- $3n + 5 = O(n)$ perché $3n + 5 \leq 4n$ per $n \geq 5$ (scegliamo $c = 4$, $n_0 = 5$)
- $n^2 + 100n + 50 = O(n^2)$ perché $n^2 + 100n + 50 \leq 151n^2$ per $n \geq 1$
- $\log n = O(n)$ perché il logaritmo cresce più lentamente di n
- $2^n = O(3^n)$ ma $3^n \neq O(2^n)$

Visualizzazione grafica:

$$n^2 + 3 * x + 2; [red, dashed, thick] 2 * x^2; ^2 + 3n + 2 g(n) = 2n^2$$

1.4.2 Notazione Omega-grande (Big-Omega)

Definizione 1.5 (Notazione Omega-grande)**.** *Date due funzioni $f(n)$ e $g(n)$ da $\mathbb{N}$ a $\mathbb{R}^+$, diciamo che $f(n) = \Omega(g(n))$ se esistono costanti positive c e n_0 tali che:*

$$0 \leq c \cdot g(n) \leq f(n) \quad \forall n \geq n_0$$

La notazione Omega-grande fornisce un **limite inferiore asintotico**.

In altre parole, $f(n) = \Omega(g(n))$ significa che $f(n)$ cresce almeno come $g(n)$ a meno di una costante moltiplicativa.

Esempi:

- $5n^2 = \Omega(n^2)$
- $n^3 = \Omega(n^2)$ perché n^3 cresce più velocemente di n^2
- $n = \Omega(\log n)$

1.4.3 Notazione Theta (Big-Theta)

Definizione 1.6 (Notazione Theta)**.** *Date due funzioni $f(n)$ e $g(n)$ da $\mathbb{N}$ a $\mathbb{R}^+$, diciamo che $f(n) = \Theta(g(n))$ se esistono costanti positive c_1, c_2 e n_0 tali che:*

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0$$

Equivalentemente, $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

La notazione Theta fornisce un **limite asintotico stretto**: $f(n)$ e $g(n)$ crescono allo stesso ritmo.

Esempi:

- $3n^2 + 5n + 2 = \Theta(n^2)$
- $\frac{n^2}{2} - 3n = \Theta(n^2)$
- $\log_2 n = \Theta(\log_{10} n)$ (i logaritmi in basi diverse differiscono solo per una costante)

$$n^2 + 3 * x; [red, dashed]0.5 * x^2; [green, dashed]2 * x^2; ^2 + 3nc_1g$$

1.4.4 Notazioni o-piccolo e omega-piccolo

Per completezza, introduciamo anche le notazioni asintotiche *strette*:

Definizione 1.7 (Notazione o-piccolo)**.** *$f(n) = o(g(n))$ se per ogni costante $c > 0$ esiste n_0 tale che:*

$$0 \leq f(n) < c \cdot g(n) \quad \forall n \geq n_0$$

Equivalentemente, $f(n) = o(g(n))$ se:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Esempio: $n = o(n^2)$ ma $n^2 \neq o(n^2)$

Definizione 1.8 (Notazione omega-piccolo)**.** *$f(n) = \omega(g(n))$ se per ogni costante $c > 0$ esiste n_0 tale che:*

$$0 \leq c \cdot g(n) < f(n) \quad \forall n \geq n_0$$

1.5 Proprietà delle notazioni asintotiche

Teorema 1.1 (Transitività)**.** *Per tutte le funzioni f, g, h:*

- *Se $f(n) = O(g(n))$ e $g(n) = O(h(n))$ allora $f(n) = O(h(n))$*
- *Se $f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n))$ allora $f(n) = \Omega(h(n))$*
- *Se $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$ allora $f(n) = \Theta(h(n))$*

Dimostrazione. Dimostriamo solo il primo caso (gli altri sono analoghi).

Supponiamo $f(n) = O(g(n))$ e $g(n) = O(h(n))$. Allora esistono costanti c_1, c_2, n_1, n_2 tali che:

$$f(n) \leq c_1 \cdot g(n) \quad \forall n \geq n_1$$
$$g(n) \leq c_2 \cdot h(n) \quad \forall n \geq n_2$$

Per $n \geq \max(n_1, n_2)$:

$$f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n)$$

Ponendo $c = c_1 \cdot c_2$ e $n_0 = \max(n_1, n_2)$, abbiamo $f(n) = O(h(n))$. □

Teorema 1.2 (Riflessività)**.** *Per ogni funzione $f(n)$: $f(n) = \Theta(f(n))$*

Teorema 1.3 (Simmetria di Theta)**.** *$f(n) = \Theta(g(n))$ se e solo se $g(n) = \Theta(f(n))$*

Teorema 1.4 (Simmetria trasposta)**.** *$f(n) = O(g(n))$ se e solo se $g(n) = \Omega(f(n))$*

1.6 Classi di complessità comuni

Ordinate dalla più efficiente alla meno efficiente:

Notazione	Nome	Esempio
$O(1)$	Costante	Accesso a array, operazioni aritmetiche
$O(\log n)$	Logaritmica	Ricerca binaria
$O(\sqrt{n})$	Radice quadrata	Controllo primalità (ingenuo)
$O(n)$	Lineare	Ricerca lineare, scansione array
$O(n \log n)$	Linearitmica	Merge sort, heap sort
$O(n^2)$	Quadratica	Bubble sort, insertion sort
$O(n^3)$	Cubica	Moltiplicazione matrici (ingenua)
$O(2^n)$	Esponenziale	Insieme delle parti, Torre di Hanoi
$O(n!)$	Fattoriale	Permutazioni, problema del commesso viaggiatore (brute force)

Confronto grafico delle crescite:

$$n^2; ,\log n) O(n) O(n \log n) O(n^2)$$

1.7 Tecniche di analisi

1.7.1 Conteggio delle operazioni

Il metodo più diretto consiste nel contare le operazioni elementari.

Esempio: Somma di array

```
def SommaArray(A, n):
    somma = 0              # 1 operazione
    for i = 1 to n:        # n iterazioni
        somma = somma + A[i]   # 2 operazioni per iterazione
    return somma           # 1 operazione
```

Analisi:

- Inizializzazione: 1 operazione
- Ciclo: n iterazioni $\times$ 2 operazioni $= 2n$ operazioni
- Return: 1 operazione
- Totale: $T(n) = 1 + 2n + 1 = 2n + 2$

Quindi: $T(n) = 2n + 2 = \Theta(n)$

1.7.2 Cicli annidati

Esempio: Somma di una matrice

```
def SommaMatrice(M, n):
    somma = 0
    for i = 1 to n:
        for j = 1 to n:
            somma = somma + M[i][j]
    return somma
```

Analisi:

- Il ciclo esterno esegue n iterazioni
- Il ciclo interno esegue n iterazioni per ogni iterazione del ciclo esterno
- Operazioni interne: costanti
- Totale: $T(n) = c \cdot n \cdot n = cn^2 = \Theta(n^2)$

1.7.3 Algoritmi ricorsivi

Per gli algoritmi ricorsivi usiamo le **relazioni di ricorrenza**.

Esempio: Fattoriale

```
def Fattoriale(n):
    if n == 0:
        return 1
    else:
        return n * Fattoriale(n - 1)
```

Relazione di ricorrenza:

$$T(n) = \begin{cases} O(1) & \text{se } n = 0 \\ T(n-1) + O(1) & \text{se } n > 0 \end{cases}$$

Risoluzione:

$$\begin{aligned} T(n) &= T(n-1) + c \\ &= T(n-2) + c + c = T(n-2) + 2c \\ &= T(n-3) + 3c \\ &\vdots \\ &= T(0) + nc \\ &= O(1) + nc = \Theta(n) \end{aligned}$$

1.8 Master Theorem

Per molti algoritmi ricorsivi di tipo divide-et-impera, il Master Theorem fornisce una soluzione immediata.

Teorema 1.5 (Master Theorem - Forma semplificata)**.** *Sia $T(n)$ definita dalla ricorrenza:*

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

dove $a \geq 1$, $b > 1$ sono costanti, e $f(n)$ è asintoticamente positiva. Allora:

1. *Se $f(n) = O(n^{\log_b a - \epsilon})$ per qualche $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$*
2. *Se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \log n)$*
3. *Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche $\epsilon > 0$, e se $a \cdot f(n/b) \leq c \cdot f(n)$ per qualche $c < 1$ e n sufficientemente grande, allora $T(n) = \Theta(f(n))$*

Esempi di applicazione:

1. Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$

- $a = 2, b = 2, f(n) = \Theta(n)$

- $n^{\log_b a} = n^{\log_2 2} = n^1 = n$
- $f(n) = \Theta(n) = \Theta(n^{\log_b a}) \rightarrow$ Caso 2
- Risultato: $T(n) = \Theta(n \log n)$

2. Ricerca binaria: $T(n) = T(n/2) + \Theta(1)$

- $a = 1, b = 2, f(n) = \Theta(1)$
- $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$
- $f(n) = \Theta(1) = \Theta(n^{\log_b a}) \rightarrow$ Caso 2
- Risultato: $T(n) = \Theta(\log n)$

3. Moltiplicazione matrici (Strassen): $T(n) = 7T(n/2) + \Theta(n^2)$

- $a = 7, b = 2, f(n) = \Theta(n^2)$
- $n^{\log_b a} = n^{\log_2 7} \approx n^{2.807}$
- $f(n) = O(n^{2.807-\epsilon}) \rightarrow$ Caso 1
- Risultato: $T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$

1.9 Casi di analisi: Best, Average, Worst

Un algoritmo può avere comportamenti diversi su input diversi della stessa dimensione. Distinguiamo tre scenari:

Definizione 1.9 (Caso migliore (Best Case))**.** *La complessità nel* ***caso migliore*** $T_{best}(n)$ *è il tempo minimo richiesto su un input di dimensione* n*.*

Definizione 1.10 (Caso peggiore (Worst Case))**.** *La complessità nel* ***caso peggiore*** $T_{worst}(n)$ *è il tempo massimo richiesto su un input di dimensione* n*.*

Definizione 1.11 (Caso medio (Average Case))**.** *La complessità nel* ***caso medio*** $T_{avg}(n)$ *è il tempo medio su tutti i possibili input di dimensione* n*, assumendo una distribuzione di probabilità sugli input.*

1.9.1 Esempio: Ricerca lineare

```
1 def RicercaLineare(A, n, chiave):
2     for i = 1 to n:
3         if A[i] == chiave:
4             return i
5     return -1  // elemento non trovato
```

Analisi dei casi:

- **Caso migliore:** La chiave è il primo elemento. $T_{\text{best}}(n) = \Theta(1)$
- **Caso peggiore:** La chiave non è presente o è l'ultimo elemento. $T_{\text{worst}}(n) = \Theta(n)$
- **Caso medio:** Assumendo che la chiave sia presente con uguale probabilità in ogni posizione:

$$T_{\text{avg}}(n) = \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2} = \Theta(n)$$

1.9.2 Esempio: Quick Sort

Caso peggiore: Il pivot è sempre il minimo o il massimo.

$$T_{\text{worst}}(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

Risoluzione:

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-2) + c(n-1) + cn \\ &= T(n-3) + c(n-2) + c(n-1) + cn \\ &= c\sum_{i=1}^{n} i = c \cdot \frac{n(n+1)}{2} = \Theta(n^2) \end{aligned}$$

Caso migliore: Il pivot divide sempre a metà.

$$T_{\text{best}}(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n) \quad \text{(Master Theorem)}$$

Caso medio: Con un'analisi più complessa si dimostra:

$$T_{\text{avg}}(n) = \Theta(n \log n)$$

1.10 Limiti inferiori

Oltre ad analizzare algoritmi specifici, possiamo studiare i **limiti inferiori** per classi di problemi.

Teorema 1.6 (Limite inferiore per l'ordinamento basato su confronti)**.** *Qualsiasi algoritmo di ordinamento basato su confronti richiede $\Omega(n \log n)$ confronti nel caso peggiore per ordinare n elementi.*

Idea della dimostrazione. Un algoritmo basato su confronti può essere rappresentato come un albero di decisione binario. Ogni foglia corrisponde a una possibile permutazione dell'input. Ci sono $n!$ permutazioni possibili, quindi l'albero deve avere almeno $n!$ foglie.

L'altezza h di un albero binario con ℓ foglie soddisfa:

$$\ell \leq 2^h \implies h \geq \log_2 \ell$$

Nel nostro caso:

$$h \geq \log_2(n!) = \sum_{i=1}^{n} \log i \geq \sum_{i=n/2}^{n} \log i \geq \frac{n}{2} \log \frac{n}{2} = \Omega(n \log n)$$

Quindi qualsiasi algoritmo di ordinamento basato su confronti richiede almeno $\Omega(n \log n)$ confronti. □

Questo teorema implica che algoritmi come Merge Sort e Heap Sort sono **asintoticamente ottimi** per l'ordinamento basato su confronti.

1.11 Complessità spaziale

Finora ci siamo concentrati sul tempo. Ma la memoria è anch'essa una risorsa importante.

Definizione 1.12 (Complessità spaziale)**.** *La **complessità spaziale** $S(n)$ di un algoritmo è la quantità massima di memoria utilizzata durante l'esecuzione su un input di dimensione n.*

Si distingue tra:

- **Spazio ausiliario**: Memoria extra usata oltre all'input
- **Spazio totale**: Spazio ausiliario + spazio per l'input

Esempio: Merge Sort

- Spazio ausiliario: $\Theta(n)$ (per l'array temporaneo nella fusione)
- Spazio totale: $\Theta(n)$

Esempio: Quick Sort

- Spazio ausiliario: $\Theta(\log n)$ (per lo stack di ricorsione nel caso medio)
- Spazio ausiliario peggiore: $\Theta(n)$ (quando l'albero di ricorsione è sbilanciato)

1.12 Esercizi

1.12.1 Esercizio 1

Dimostrare che $2n^2 + 3n + 1 = \Theta(n^2)$.

1.12.2 Esercizio 2

Determinare la complessità asintotica di:

```
def Funzione(n):
    somma = 0
    for i = 1 to n:
        for j = 1 to i:
            somma = somma + 1
    return somma
```

1.12.3 Esercizio 3

Risolvere la ricorrenza $T(n) = 3T(n/4) + n^2$ usando il Master Theorem.

1.12.4 Esercizio 4

Analizzare il caso migliore, medio e peggiore della ricerca binaria.

1.12.5 Esercizio 5

Dimostrare che se $f(n) = O(g(n))$ e $g(n) = O(h(n))$, allora $f(n) + g(n) = O(h(n))$.

1.13 Conclusioni

L'analisi di complessità è lo strumento fondamentale per valutare e confrontare algoritmi. Le notazioni asintotiche ci permettono di astrarre dai dettagli implementativi e concentrarci sul comportamento scalabile degli algoritmi.

I concetti chiave di questo capitolo:

- Le notazioni O, Omega, Theta caratterizzano limiti superiori, inferiori e stretti
- Il Master Theorem risolve molte ricorrenze di tipo divide-et-impera

- Gli algoritmi hanno comportamenti diversi nei casi best, average, worst
- Esistono limiti inferiori teorici per classi di problemi
- La complessità spaziale è importante quanto quella temporale

Nei prossimi capitoli applicheremo questi strumenti all'analisi di strutture dati e algoritmi concreti.

Capitolo 2

Array e Liste

2.1 Introduzione

Le strutture dati lineari sono il fondamento dell'organizzazione dei dati in memoria. In questo capitolo studieremo le due strutture lineari fondamentali: gli **array** e le **liste concatenate**. Entrambe memorizzano sequenze di elementi, ma con caratteristiche di accesso e modifica molto diverse.

La scelta tra array e liste dipende dalle operazioni che dobbiamo eseguire più frequentemente: accesso casuale, inserimento, cancellazione. Comprendere i trade-off tra queste strutture è essenziale per progettare algoritmi efficienti.

2.2 Array

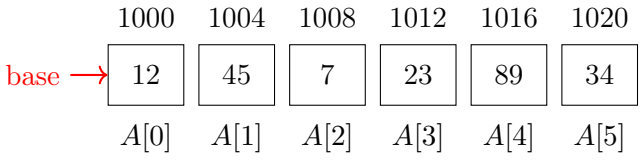
2.2.1 Definizione e proprietà

Definizione 2.1 (Array)**.** *Un **array** è una struttura dati che memorizza n elementi dello stesso tipo in posizioni di memoria contigue. Ogni elemento è identificato da un **indice** intero compreso tra 0 e $n-1$ (o tra 1 e n a seconda della convenzione).*

Proprietà fondamentali:

- **Dimensione fissa**: La dimensione dell'array è determinata alla creazione
- **Accesso diretto**: L'elemento in posizione i è accessibile in tempo $O(1)$
- **Memoria contiguità**: Gli elementi occupano celle consecutive in memoria
- **Tipo omogeneo**: Tutti gli elementi hanno lo stesso tipo

Rappresentazione in memoria:



Indirizzi di memoria assumendo 4 byte per elemento

2.2.2 Calcolo dell'indirizzo

Se base è l'indirizzo del primo elemento e size è la dimensione in byte di ogni elemento, l'indirizzo dell'elemento in posizione i è:

$$\text{address}(A[i]) = \text{base} + i \times \text{size}$$

Questa formula spiega perché l'accesso è $O(1)$: è una semplice operazione aritmetica.

2.2.3 Operazioni su array

Accesso

```
def Accesso(A, i):
    """
    Accede all'elemento in posizione i
    Input: array A, indice i
    Output: A[i]
    Complessità: O(1)
    """
    return A[i]
```

Complessità temporale: $\Theta(1)$
Complessità spaziale: $\Theta(1)$

Modifica

```
def Modifica(A, i, valore):
    """
    Modifica l'elemento in posizione i
    Complessità: O(1)
    """
    A[i] = valore
```

Complessità temporale: $\Theta(1)$
Complessità spaziale: $\Theta(1)$

Ricerca

Ricerca lineare (array non ordinato):

```
def RicercaLineare(A, n, chiave):
    """
    Cerca un elemento nell'array
    Input: array A di n elementi, chiave da cercare
    Output: indice se trovato, -1 altrimenti
    """
    for i = 0 to n-1:
        if A[i] == chiave:
            return i
    return -1
```

Analisi:

- Caso migliore: $\Theta(1)$ (elemento in prima posizione)
- Caso peggiore: $\Theta(n)$ (elemento assente o in ultima posizione)
- Caso medio: $\Theta(n)$

Ricerca binaria (array ordinato):

```python
def RicercaBinaria(A, n, chiave):
    """
    Cerca un elemento in un array ordinato
    Input: array ordinato A di n elementi, chiave
    Output: indice se trovato, -1 altrimenti
    Complessità: O(log n)
    """
    sinistra = 0
    destra = n - 1

    while sinistra <= destra:
        medio = (sinistra + destra) // 2

        if A[medio] == chiave:
            return medio
        elif A[medio] < chiave:
            sinistra = medio + 1
        else:
            destra = medio - 1

    return -1
```

Analisi della ricerca binaria:

Sia $T(n)$ il numero di confronti nel caso peggiore. Ad ogni iterazione, la dimensione dell'intervallo di ricerca si dimezza:

$$T(n) = T(n/2) + O(1)$$

Per il Master Theorem (caso 2 con $a = 1, b = 2, f(n) = O(1)$):

$$T(n) = \Theta(\log n)$$

Teorema 2.1 (Correttezza della ricerca binaria)**.** *Se l'array A è ordinato, l'algoritmo di ricerca binaria termina e restituisce l'indice della chiave se presente, -1 altrimenti.*

Dimostrazione. Dimostrazione per invariante di ciclo. L'invariante è: "Se la chiave è presente nell'array, allora si trova nell'intervallo $[sinistra, destra]$".

Inizializzazione: Prima del primo ciclo, $sinistra = 0$ e $destra = n - 1$, quindi l'intervallo copre tutto l'array.

Mantenimento: Ad ogni iterazione:

- Se $A[medio] = chiave$, l'algoritmo termina correttamente.
- Se $A[medio] < chiave$, per l'ordinamento, la chiave può essere solo a destra di $medio$, quindi restringiamo a $[medio + 1, destra]$.
- Se $A[medio] > chiave$, per l'ordinamento, la chiave può essere solo a sinistra di $medio$, quindi restringiamo a $[sinistra, medio - 1]$.

Terminazione: Il ciclo termina quando $sinistra > destra$ (intervallo vuoto) o quando troviamo la chiave. Se l'intervallo diventa vuoto, la chiave non è presente. □

Inserimento

Inserimento in coda (se c'è spazio):

```
def InserimentoCoda(A, n, valore):
    """
    Inserisce un elemento in coda
    Precondizione: n < capacità dell'array
    Complessità: O(1)
    """
    A[n] = valore
    return n + 1  // nuova dimensione
```

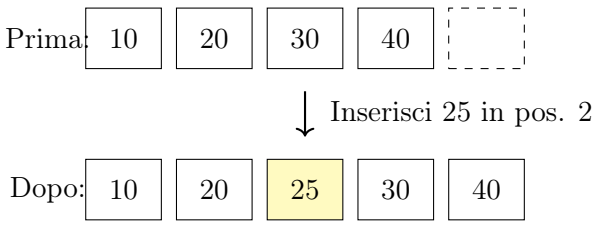
Complessità: $\Theta(1)$

Inserimento in posizione arbitraria:

```
def Inserimento(A, n, i, valore):
    """
    Inserisce valore in posizione i
    Richiede lo shift degli elementi successivi
    Complessità: O(n)
    """
    // Shift elementi verso destra
    for j = n-1 down to i:
        A[j+1] = A[j]

    A[i] = valore
    return n + 1
```

Analisi:

- Caso migliore: $\Theta(1)$ (inserimento in coda)
- Caso peggiore: $\Theta(n)$ (inserimento in testa, richiede shift di tutti gli elementi)
- Caso medio: $\Theta(n)$



Cancellazione

```
def Cancellazione(A, n, i):
    """
    Cancella l'elemento in posizione i
    Richiede lo shift degli elementi successivi
    Complessità: O(n)
    """
    // Shift elementi verso sinistra
    for j = i to n-2:
        A[j] = A[j+1]

    return n - 1  // nuova dimensione
```

Analisi: Analoga all'inserimento.

- Caso migliore: $\Theta(1)$ (cancellazione dell'ultimo elemento)

- Caso peggiore: $\Theta(n)$ (cancellazione del primo elemento)
- Caso medio: $\Theta(n)$

2.2.4 Array dinamici

Gli array statici hanno dimensione fissa, il che è limitante. Gli **array dinamici** (dynamic arrays, in C++ `std::vector`, in Java `ArrayList`, in Python `list`) risolvono questo problema.

Strategia di raddoppio:

- Quando l'array è pieno, allochiamo un nuovo array di dimensione doppia
- Copiamo tutti gli elementi nel nuovo array
- Deallochiamo il vecchio array

```
def InserimentoDinamico(A, n, capacità, valore):
    """
    Inserisce in un array dinamico
    """
    if n == capacità:
        // Array pieno, raddoppia
        nuova_capacità = 2 * capacità
        B = nuovo array di dimensione nuova_capacità

        for i = 0 to n-1:
            B[i] = A[i]

        A = B
        capacità = nuova_capacità

    A[n] = valore
    return n + 1, capacità
```

Analisi ammortizzata:

Un singolo inserimento può costare $\Theta(n)$ se richiede raddoppio, ma non tutti gli inserimenti richiedono raddoppio.

Teorema 2.2 (Costo ammortizzato dell'inserimento in array dinamico)**.** *Il costo ammortizzato di un inserimento in un array dinamico con strategia di raddoppio è* $\Theta(1)$.

Dimostrazione. Consideriamo una sequenza di n inserimenti partendo da un array vuoto.

Gli inserimenti che richiedono raddoppio avvengono quando $n = 1, 2, 4, 8, 16, \ldots, 2^k$ dove $2^k \leq n < 2^{k+1}$.

Il costo totale è:

$$\begin{aligned} C(n) &= n + \sum_{i=0}^{\lfloor \log n \rfloor} 2^i \\ &= n + (2^{\lfloor \log n \rfloor + 1} - 1) \\ &< n + 2n = 3n \end{aligned}$$

Il costo ammortizzato per operazione è:

$$\frac{C(n)}{n} < \frac{3n}{n} = 3 = O(1)$$

□

2.2.5 Tabella riassuntiva delle complessità

Operazione	Array statico	Array dinamico
Accesso	$O(1)$	$O(1)$
Modifica	$O(1)$	$O(1)$
Ricerca (non ordinato)	$O(n)$	$O(n)$
Ricerca (ordinato)	$O(\log n)$	$O(\log n)$
Inserimento in coda	$O(1)^*$	$O(1)$ ammortizzato
Inserimento in posizione i	$O(n)$	$O(n)$
Cancellazione	$O(n)$	$O(n)$

* se c'è spazio disponibile

2.3 Liste concatenate

2.3.1 Definizione e struttura

Definizione 2.2 (Lista concatenata semplice)**.** *Una **lista concatenata semplice** è una sequenza di nodi, dove ogni nodo contiene:*

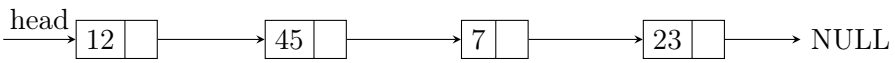
- *Un **dato** (valore memorizzato)*
- *Un **puntatore** al nodo successivo*

*Il primo nodo è chiamato **testa** (head), l'ultimo puntatore è `NULL`.*

Struttura del nodo:

```
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.next = None
```

Rappresentazione grafica:



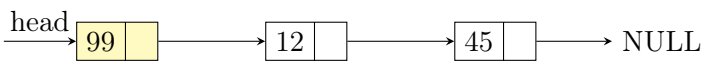
2.3.2 Operazioni su liste concatenate

Inserimento in testa

```
def InserimentoTesta(head, valore):
    """
    Inserisce un nuovo nodo in testa alla lista
    Input: testa della lista, valore da inserire
    Output: nuova testa
    Complessità: O(1)
    """
    nuovo = Nodo(valore)
    nuovo.next = head
    return nuovo
```

Complessità: $\Theta(1)$

Visualizzazione:



Inserimento in coda

```
def InserimentoCoda(head, valore):
    """
    Inserisce un nuovo nodo in coda alla lista
    Complessità: O(n)
    """
    nuovo = Nodo(valore)

    if head == None:
        return nuovo

    corrente = head
    while corrente.next != None:
        corrente = corrente.next

    corrente.next = nuovo
    return head
```

Complessità: $\Theta(n)$ (dobbiamo scorrere tutta la lista)
Ottimizzazione: Mantenere un puntatore alla coda riduce la complessità a $\Theta(1)$.

Inserimento in posizione

```
def InserimentoPosizione(head, valore, posizione):
    """
    Inserisce un nodo nella posizione specificata (0-based)
    Complessità: O(n)
    """
    if posizione == 0:
        return InserimentoTesta(head, valore)

    nuovo = Nodo(valore)
    corrente = head

    for i = 0 to posizione-2:
        if corrente == None:
            return head  // posizione non valida
        corrente = corrente.next

    if corrente != None:
        nuovo.next = corrente.next
        corrente.next = nuovo

    return head
```

Complessità: $O(n)$

Cancellazione in testa

```
def CancellazioneTesta(head):
    """
    Cancella il primo nodo
    Complessità: O(1)
    """
    if head == None:
        return None
```

```
8
9       nuova_head = head.next
10      // In linguaggi con garbage collection, head viene deallocato
            automaticamente
11      return nuova_head
```

Complessità: $\Theta(1)$

Cancellazione di un valore

```
1  def CancellaValore(head, valore):
2      """
3      Cancella il primo nodo con il valore specificato
4      Complessità: O(n)
5      """
6      if head == None:
7          return None
8
9      // Caso speciale: testa da cancellare
10     if head.dato == valore:
11         return head.next
12
13     corrente = head
14     while corrente.next != None:
15         if corrente.next.dato == valore:
16             corrente.next = corrente.next.next
17             return head
18         corrente = corrente.next
19
20     return head  // valore non trovato
```

Complessità: $O(n)$

Ricerca

```
1  def Ricerca(head, valore):
2      """
3      Cerca un valore nella lista
4      Output: il nodo se trovato, None altrimenti
5      Complessità: O(n)
6      """
7      corrente = head
8      while corrente != None:
9          if corrente.dato == valore:
10             return corrente
11         corrente = corrente.next
12     return None
```

Complessità: $O(n)$ in tutti i casi

Attraversamento

```
1  def Stampa(head):
2      """
3      Stampa tutti gli elementi della lista
4      Complessità: O(n)
5      """
```

```
    corrente = head
    while corrente != None:
        print(corrente.dato)
        corrente = corrente.next
```

Complessità: $\Theta(n)$

2.3.3 Liste concatenate doppie

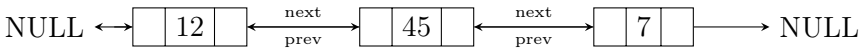
Definizione 2.3 (Lista concatenata doppia)**.** *Una **lista concatenata doppia** è una sequenza di nodi dove ogni nodo contiene:*

- *Un dato*
- *Un puntatore al nodo **successivo** (`next`)*
- *Un puntatore al nodo **precedente** (`prev`)*

Struttura del nodo:

```
class NodoDoppio:
    def __init__(self, dato):
        self.dato = dato
        self.next = None
        self.prev = None
```

Rappresentazione grafica:



Vantaggi:

- Scorrimento bidirezionale
- Cancellazione di un nodo dato il puntatore al nodo in $O(1)$
- Inserimento prima di un nodo in $O(1)$

Svantaggi:

- Maggiore uso di memoria (un puntatore in più per nodo)
- Maggiore complessità nel mantenere l'invariante dei puntatori

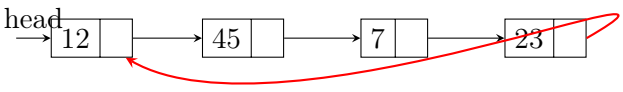
Cancellazione dato il nodo (lista doppia)

```
def CancellaNodo(nodo):
    """
    Cancella un nodo data la sua posizione
    SOLO per liste doppie
    Complessità: O(1)
    """
    if nodo.prev != None:
        nodo.prev.next = nodo.next

    if nodo.next != None:
        nodo.next.prev = nodo.prev

    // nodo viene deallocato
```

Complessità: $\Theta(1)$ — questo è un grande vantaggio rispetto alla lista semplice!

2.3.4 Liste circolari

Definizione 2.4 (Lista circolare)**.** *Una **lista circolare** è una lista concatenata in cui l'ultimo nodo punta al primo (invece di `NULL`).*



Applicazioni: Round-robin scheduling, buffer circolari.

2.3.5 Confronto array vs liste

Operazione	Array	Lista concatenata
Accesso al k-esimo elemento	$O(1)$	$O(k)$
Ricerca elemento	$O(n)$	$O(n)$
Inserimento in testa	$O(n)$	$O(1)$
Inserimento in coda	$O(1)^*$	$O(n)$ o $O(1)^{**}$
Inserimento in mezzo	$O(n)$	$O(n)$
Cancellazione in testa	$O(n)$	$O(1)$
Cancellazione in coda	$O(1)$	$O(n)$ o $O(1)^{**}$
Uso memoria	Compatto	Extra per puntatori
Località di cache	Ottima	Scarsa
Dimensione	Fissa o dinamica	Dinamica

* se c'è spazio; ** se si mantiene puntatore alla coda

2.4 Applicazioni pratiche

2.4.1 Implementazione di buffer

Gli array circolari sono usati per implementare buffer FIFO efficienti in sistemi embedded e streaming.

2.4.2 Undo/Redo in editor

Le liste doppie permettono di navigare avanti e indietro nella storia delle modifiche.

2.4.3 Rappresentazione di polinomi

Un polinomio sparso $P(x) = 3x^{100} + 5x^{50} + 2$ può essere rappresentato efficientemente con una lista di coppie (coefficiente, esponente).

2.4.4 Gestione memoria dinamica

I sistemi operativi usano liste concatenate per gestire blocchi di memoria liberi (free lists).

2.5 Algoritmi avanzati su liste

2.5.1 Inversione di una lista

```python
def Inverti(head):
    """
    Inverte una lista concatenata
    Complessità: O(n) tempo, O(1) spazio
    """
    prev = None
    corrente = head

    while corrente != None:
        prossimo = corrente.next
        corrente.next = prev
        prev = corrente
        corrente = prossimo

    return prev
```

Complessità: $\Theta(n)$ tempo, $\Theta(1)$ spazio

2.5.2 Rilevazione di cicli (Floyd's Algorithm)

```python
def HaCiclo(head):
    """
    Rileva se una lista ha un ciclo
    Algoritmo della tartaruga e della lepre
    Complessità: O(n) tempo, O(1) spazio
    """
    if head == None:
        return False

    lento = head
    veloce = head

    while veloce != None and veloce.next != None:
        lento = lento.next
        veloce = veloce.next.next

        if lento == veloce:
            return True

    return False
```

Teorema 2.3 (Correttezza dell'algoritmo di Floyd)**.** *Se una lista ha un ciclo, i puntatori lento e veloce si incontreranno.*

Idea. Quando il puntatore lento entra nel ciclo, il veloce è già nel ciclo. La distanza tra loro diminuisce di 1 ad ogni iterazione (il veloce avanza di 2, il lento di 1). Eventualmente la distanza diventa 0. □

2.5.3 Trovare il punto medio

```python
def TrovaMedio(head):
    """
    Trova il nodo medio della lista
    Complessità: O(n) tempo, O(1) spazio
    """
    if head == None:
```

```
        return None

    lento = head
    veloce = head

    while veloce.next != None and veloce.next.next != None:
        lento = lento.next
        veloce = veloce.next.next

    return lento
```

2.6 Conclusioni

Array e liste concatenate sono le strutture dati lineari fondamentali. La scelta tra le due dipende dalle operazioni predominanti:

- **Usa array** quando: accesso casuale frequente, dimensione nota, località di cache importante
- **Usa liste** quando: inserimenti/cancellazioni frequenti, dimensione variabile, no accesso casuale

Molte strutture dati complesse (stack, code, hash table con chaining) sono costruite su questi fondamenti.

Punti chiave:

- Array: $O(1)$ accesso, $O(n)$ inserimento/cancellazione
- Liste: $O(1)$ inserimento/cancellazione in testa, $O(n)$ accesso
- Liste doppie: $O(1)$ cancellazione dato il nodo
- Array dinamici: $O(1)$ inserimento ammortizzato

Capitolo 3

Stack e Code

3.1 Introduzione

Stack e code sono strutture dati lineari astratte che impongono regole specifiche sull'ordine di accesso agli elementi. A differenza di array e liste, dove possiamo accedere a qualsiasi posizione, stack e code permettono accesso solo agli estremi della sequenza.

Queste restrizioni, lungi dall'essere limitazioni, rendono stack e code strumenti potentissimi per modellare situazioni reali e risolvere problemi algoritmici fondamentali.

3.2 Stack (Pila)

3.2.1 Definizione e proprietà

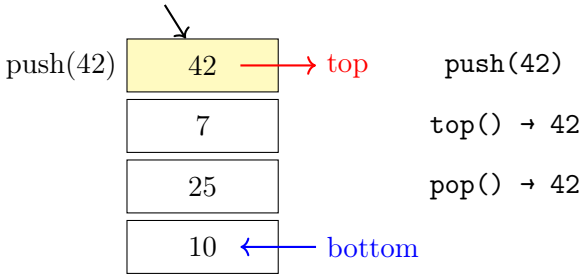
Definizione 3.1 (Stack)**.** *Uno **stack** (o pila) è una struttura dati lineare che segue il principio **LIFO** (Last In, First Out): l'ultimo elemento inserito è il primo ad essere rimosso.*

Metafora: Una pila di piatti. Puoi aggiungere un piatto in cima, e puoi rimuovere solo il piatto in cima.

Operazioni fondamentali:

- `push(x)`: Inserisce l'elemento x in cima allo stack
- `pop()`: Rimuove e restituisce l'elemento in cima
- `top()`/`peek()`: Restituisce l'elemento in cima senza rimuoverlo
- `isEmpty()`: Verifica se lo stack è vuoto
- `size()`: Restituisce il numero di elementi

Visualizzazione:



3.2.2 Implementazione con array

```
class StackArray:
    def __init__(self, capacità):
        self.array = nuovo array di dimensione capacità
        self.top = -1  // indice dell'elemento in cima
        self.capacità = capacità

    def isEmpty(self):
        return self.top == -1

    def isFull(self):
        return self.top == self.capacità - 1

    def push(self, x):
        """
        Inserisce x in cima
        Complessità: O(1)
        """
        if self.isFull():
            errore "Stack overflow"

        self.top = self.top + 1
        self.array[self.top] = x

    def pop(self):
        """
        Rimuove e restituisce l'elemento in cima
        Complessità: O(1)
        """
        if self.isEmpty():
            errore "Stack underflow"

        x = self.array[self.top]
        self.top = self.top - 1
        return x

    def peek(self):
        """
        Restituisce l'elemento in cima senza rimuoverlo
        Complessità: O(1)
        """
        if self.isEmpty():
            errore "Stack vuoto"

        return self.array[self.top]

    def size(self):
        return self.top + 1
```

Analisi:

- Tutte le operazioni sono $O(1)$
- Spazio: $O(n)$ dove n è la capacità
- Svantaggio: capacità fissa

3.2.3 Implementazione con lista concatenata

```
class StackLista:
    def __init__(self):
        self.top = None
        self.dimensione = 0

    def isEmpty(self):
        return self.top == None

    def push(self, x):
        """
        Inserisce x in cima
        Complessità: O(1)
        """
        nuovo = Nodo(x)
        nuovo.next = self.top
        self.top = nuovo
        self.dimensione = self.dimensione + 1

    def pop(self):
        """
        Rimuove e restituisce l'elemento in cima
        Complessità: O(1)
        """
        if self.isEmpty():
            errore "Stack underflow"

        x = self.top.dato
        self.top = self.top.next
        self.dimensione = self.dimensione - 1
        return x

    def peek(self):
        if self.isEmpty():
            errore "Stack vuoto"
        return self.top.dato

    def size(self):
        return self.dimensione
```

Vantaggi:

- Nessuna capacità massima
- Tutte le operazioni sono $O(1)$

Svantaggi:

- Overhead dei puntatori
- Peggiore località di cache

3.2.4 Applicazioni degli stack

Valutazione di espressioni

Gli stack sono fondamentali per valutare espressioni aritmetiche.

Esempio: Espressioni postfisse (notazione polacca inversa)

Nell'espressione postfissa, gli operatori seguono gli operandi:

- Infissa: $(3 + 4) \times 5$
- Postfissa: $3\ 4\ +\ 5\ \times$

```
def ValutaPostfissa(espressione):
    """
    Valuta un'espressione postfissa
    Input: array di token (numeri e operatori)
    Output: risultato
    Complessità: O(n)
    """
    stack = Stack()

    for token in espressione:
        if token è un numero:
            stack.push(token)
        else:  // token è un operatore
            b = stack.pop()
            a = stack.pop()

            if token == '+':
                risultato = a + b
            elif token == '-':
                risultato = a - b
            elif token == '*':
                risultato = a * b
            elif token == '/':
                risultato = a / b

            stack.push(risultato)

    return stack.pop()
```

Esempio di esecuzione: Espressione $3\ 4\ +\ 5\ \times$

Token	Stack	Azione
3	[3]	push 3
4	[3, 4]	push 4
+	[7]	pop 4, pop 3, push 3+4
5	[7, 5]	push 5
×	[35]	pop 5, pop 7, push 7×5

Risultato: 35

Conversione da infissa a postfissa (Shunting Yard Algorithm):

```
def InfissaAPostfissa(espressione):
    """
    Converte espressione infissa in postfissa
    Algoritmo di Dijkstra (Shunting Yard)
    Complessità: O(n)
    """
    output = []
    stack = Stack()

    precedenza = {'+': 1, '-': 1, '*': 2, '/': 2}

    for token in espressione:
        if token è un numero:
```

```
            output.append(token)

        elif token == '(':
            stack.push(token)

        elif token == ')':
            while not stack.isEmpty() and stack.peek() != '(':
                output.append(stack.pop())
            stack.pop()  // rimuove '('

        elif token è un operatore:
            while (not stack.isEmpty() and
                   stack.peek() != '(' and
                   precedenza[stack.peek()] >= precedenza[token]):
                output.append(stack.pop())
            stack.push(token)

    while not stack.isEmpty():
        output.append(stack.pop())

    return output
```

Bilanciamento di parentesi

Problema: Verificare se le parentesi in un'espressione sono bilanciate.

```
def ParentesiBilanciate(espressione):
    """
    Verifica bilanciamento parentesi/bracket/braces
    Input: stringa con caratteri (), [], {}
    Output: True se bilanciate, False altrimenti
    Complessità: O(n)
    """
    stack = Stack()
    aperture = {'(', '[', '{'}
    chiusure = {')', ']', '}'}
    match = {'(': ')', '[': ']', '{': '}'}

    for char in espressione:
        if char in aperture:
            stack.push(char)

        elif char in chiusure:
            if stack.isEmpty():
                return False

            top = stack.pop()
            if match[top] != char:
                return False

    return stack.isEmpty()
```

Esempi:

- `"(())"` → True
- `"([)]"` → False (ordine sbagliato)
- `"(("` → False (non chiuse)

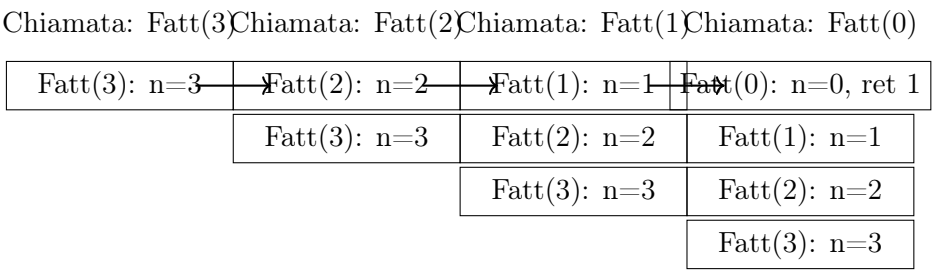
Backtracking e ricorsione

Lo stack di sistema (call stack) gestisce le chiamate ricorsive. Ogni chiamata di funzione aggiunge un *frame* allo stack contenente variabili locali e indirizzo di ritorno.

Esempio: Fattoriale

```
def Fattoriale(n):
    if n == 0:
        return 1
    else:
        return n * Fattoriale(n-1)
```

Per `Fattoriale(3)`, lo stack evolve così:



Poi lo stack si svuota man mano che le funzioni ritornano: $1 \to 1 \to 2 \to 6$.

Percorsi in profondità (DFS)

Il Depth-First Search usa uno stack (esplicito o tramite ricorsione) per esplorare grafi.

Undo/Redo

Editor di testo e software di grafica usano due stack: uno per undo, uno per redo.

```
class Editor:
    def __init__(self):
        self.undo_stack = Stack()
        self.redo_stack = Stack()

    def eseguiAzione(self, azione):
        azione.esegui()
        self.undo_stack.push(azione)
        self.redo_stack.clear()  // invalida redo

    def undo(self):
        if not self.undo_stack.isEmpty():
            azione = self.undo_stack.pop()
            azione.annulla()
            self.redo_stack.push(azione)

    def redo(self):
        if not self.redo_stack.isEmpty():
            azione = self.redo_stack.pop()
            azione.esegui()
            self.undo_stack.push(azione)
```

3.3 Code (Queue)

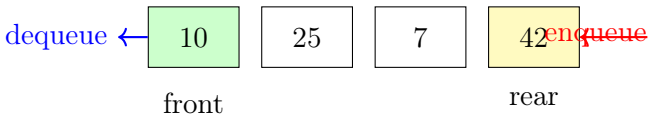
3.3.1 Definizione e proprietà

Definizione 3.2 (Coda)**.** *Una **coda** (queue) è una struttura dati lineare che segue il principio **FIFO** (First In, First Out): il primo elemento inserito è il primo ad essere rimosso.*

Metafora: Una coda di persone in fila. Chi arriva prima viene servito prima.
Operazioni fondamentali:

- `enqueue(x)`: Inserisce l'elemento x in coda
- `dequeue()`: Rimuove e restituisce l'elemento in testa
- `front()`: Restituisce l'elemento in testa senza rimuoverlo
- `isEmpty()`: Verifica se la coda è vuota
- `size()`: Restituisce il numero di elementi

Visualizzazione:



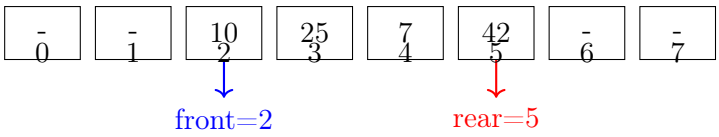
3.3.2 Implementazione con array circolare

Per evitare di sprecare spazio quando facciamo dequeue, usiamo un **array circolare**: quando raggiungiamo la fine dell'array, "avvolgiamo" all'inizio.

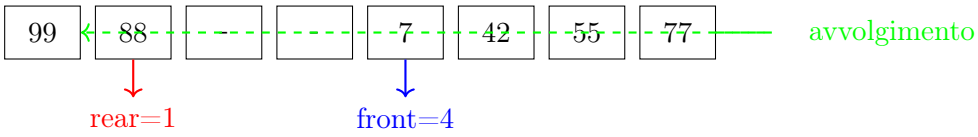
```
class QueueArray:
    def __init__(self, capacità):
        self.array = nuovo array di dimensione capacità
        self.front = 0
        self.rear = -1
        self.dimensione = 0
        self.capacità = capacità

    def isEmpty(self):
        return self.dimensione == 0

    def isFull(self):
        return self.dimensione == self.capacità

    def enqueue(self, x):
        """
        Inserisce x in coda
        Complessità: O(1)
        """
        if self.isFull():
            errore "Queue overflow"

        self.rear = (self.rear + 1) % self.capacità
        self.array[self.rear] = x
        self.dimensione = self.dimensione + 1

    def dequeue(self):
        """
```

```
        Rimuove e restituisce l'elemento in testa
        Complessità: O(1)
        """
        if self.isEmpty():
            errore "Queue underflow"

        x = self.array[self.front]
        self.front = (self.front + 1) % self.capacità
        self.dimensione = self.dimensione - 1
        return x

    def getFront(self):
        if self.isEmpty():
            errore "Queue vuota"
        return self.array[self.front]

    def size(self):
        return self.dimensione
```

Visualizzazione array circolare:



Dopo dequeue, enqueue:



3.3.3 Implementazione con lista concatenata

```
class QueueLista:
    def __init__(self):
        self.front = None
        self.rear = None
        self.dimensione = 0

    def isEmpty(self):
        return self.front == None

    def enqueue(self, x):
        """
        Inserisce x in coda
        Complessità: O(1)
        """
        nuovo = Nodo(x)

        if self.isEmpty():
            self.front = nuovo
            self.rear = nuovo
        else:
```

```
            self.rear.next = nuovo
            self.rear = nuovo

        self.dimensione = self.dimensione + 1

    def dequeue(self):
        """
        Rimuove e restituisce l'elemento in testa
        Complessità: O(1)
        """
        if self.isEmpty():
            errore "Queue underflow"

        x = self.front.dato
        self.front = self.front.next

        if self.front == None:  // coda ora vuota
            self.rear = None

        self.dimensione = self.dimensione - 1
        return x

    def getFront(self):
        if self.isEmpty():
            errore "Queue vuota"
        return self.front.dato

    def size(self):
        return self.dimensione
```

3.3.4 Applicazioni delle code

Sistemi operativi

Scheduling dei processi: I processi pronti per l'esecuzione sono mantenuti in una coda. Lo scheduler seleziona il primo processo (FIFO).

Buffer di stampa: I documenti da stampare sono accodati e stampati nell'ordine di arrivo.

Breadth-First Search (BFS)

L'algoritmo BFS per grafi usa una coda per esplorare i nodi livello per livello.

```
def BFS(grafo, sorgente):
    """
    Visita in ampiezza di un grafo
    Complessità: O(V + E)
    """
    visitato = insieme vuoto
    coda = Queue()

    coda.enqueue(sorgente)
    visitato.add(sorgente)

    while not coda.isEmpty():
        u = coda.dequeue()
        print(u)  // processa u

        for v in grafo.adiacenti(u):
```

```
17              if v not in visitato:
18                  visitato.add(v)
19                  coda.enqueue(v)
```

Gestione delle richieste

Server web: Le richieste HTTP sono accodate e processate in ordine FIFO.

Call center: Le chiamate sono accodate e gli operatori rispondono nell'ordine di arrivo.

3.3.5 Code con priorità (Priority Queue)

Definizione 3.3 (Coda con priorità)**.** *Una **coda con priorità** è una struttura dati in cui ogni elemento ha una priorità associata. L'elemento con priorità più alta viene rimosso per primo.*

Operazioni:

- `insert(x, priorità)`: Inserisce x con la priorità data
- `extractMax()`: Rimuove e restituisce l'elemento con priorità massima
- `getMax()`: Restituisce (senza rimuovere) l'elemento con priorità massima

Le code con priorità sono tipicamente implementate con **heap** (capitolo successivo).

Applicazioni:

- Algoritmo di Dijkstra (cammini minimi)
- Algoritmo di Prim (minimum spanning tree)
- Scheduling con priorità
- Simulazione di eventi discreti

3.3.6 Deque (Double-Ended Queue)

Definizione 3.4 (Deque)**.** *Una **deque** (pronuncia "deck") è una coda doppia: si possono inserire e rimuovere elementi da entrambe le estremità.*

Operazioni:

- `insertFront(x)`, `insertRear(x)`
- `deleteFront()`, `deleteRear()`
- `getFront()`, `getRear()`

Implementazione efficiente: Lista doppiamente concatenata o array circolare.

Proprietà interessante: Una deque può simulare sia uno stack che una coda!

3.4 Confronto delle strutture

Operazione	**Stack**	**Queue**
Inserimento	$O(1)$ (push)	$O(1)$ (enqueue)
Rimozione	$O(1)$ (pop)	$O(1)$ (dequeue)
Accesso in cima/testa	$O(1)$	$O(1)$
Accesso arbitrario	$O(n)$	$O(n)$
Ricerca	$O(n)$	$O(n)$
Spazio	$O(n)$	$O(n)$
Principio	LIFO	FIFO

3.5 Teoremi e proprietà

Teorema 3.1 (Complessità delle operazioni)**.** *In uno stack o coda implementati correttamente (con array o lista), tutte le operazioni fondamentali (push/pop/enqueue/dequeue) hanno complessità* $\Theta(1)$ *nel caso peggiore.*

Teorema 3.2 (Simulazione)**.** *1. Due stack possono simulare una coda (con operazioni ammortizzate* $O(1)$*)*

2. Una coda NON può simulare efficacemente uno stack

3. Una deque può simulare sia stack che code con tutte le operazioni $O(1)$

Dimostrazione del punto 1. Usiamo due stack: `inbox` e `outbox`.
Enqueue: Push su `inbox` – $O(1)$
Dequeue:

- Se `outbox` non è vuoto: pop da `outbox` – $O(1)$
- Altrimenti: sposta tutti gli elementi da `inbox` a `outbox` (invertendone l'ordine), poi pop da `outbox`

Il costo ammortizzato è $O(1)$ perché ogni elemento viene spostato al più una volta. □

3.6 Esercizi

3.6.1 Esercizio 1

Implementare una funzione che inverte uno stack usando un solo stack ausiliario.

3.6.2 Esercizio 2

Implementare uno stack che supporta anche l'operazione `getMin()` in $O(1)$.

3.6.3 Esercizio 3

Data una sequenza di operazioni push/pop, verificare se produce un output valido. Es: push(1), push(2), pop(), push(3), pop(), pop() → [2, 3, 1] è valido?

3.6.4 Esercizio 4

Implementare una coda usando due stack e dimostrare che il costo ammortizzato di dequeue è $O(1)$.

3.6.5 Esercizio 5

Scrivere un algoritmo che, data un'espressione infissa, la valuti direttamente usando due stack (uno per operandi, uno per operatori).

3.7 Conclusioni

Stack e code sono strutture dati fondamentali che, nonostante la loro semplicità concettuale, hanno applicazioni vastissime:

- **Stack**: Ricorsione, backtracking, parsing, valutazione espressioni, undo/redo

- **Code**: Scheduling, BFS, gestione eventi, buffer, simulazioni

Punti chiave:

- Stack = LIFO, Code = FIFO
- Tutte le operazioni sono $O(1)$
- Implementabili con array o liste
- Array circolari per code efficienti
- Code con priorità richiedono heap
- Deque generalizza stack e code

La scelta tra implementazione con array o lista dipende da:

- Array: migliore località di cache, dimensione massima nota
- Lista: dimensione dinamica illimitata, ma overhead di puntatori

Queste strutture sono i building block per algoritmi più complessi che vedremo nei prossimi capitoli.

Capitolo 4

Alberi

4.1 Introduzione

Gli alberi sono tra le strutture dati più importanti e versatili in informatica. A differenza delle strutture lineari (array, liste, stack, code), gli alberi organizzano i dati in modo **gerarchico**, riflettendo naturalmente relazioni di tipo "padre-figlio" che occorrono in innumerevoli contesti: filesystem, organizzazioni aziendali, documenti HTML, alberi sintattici, e molto altro.

In questo capitolo studieremo alberi binari, alberi binari di ricerca (BST), alberi AVL auto-bilancianti, e heap.

4.2 Alberi: definizioni fondamentali

4.2.1 Definizione matematica

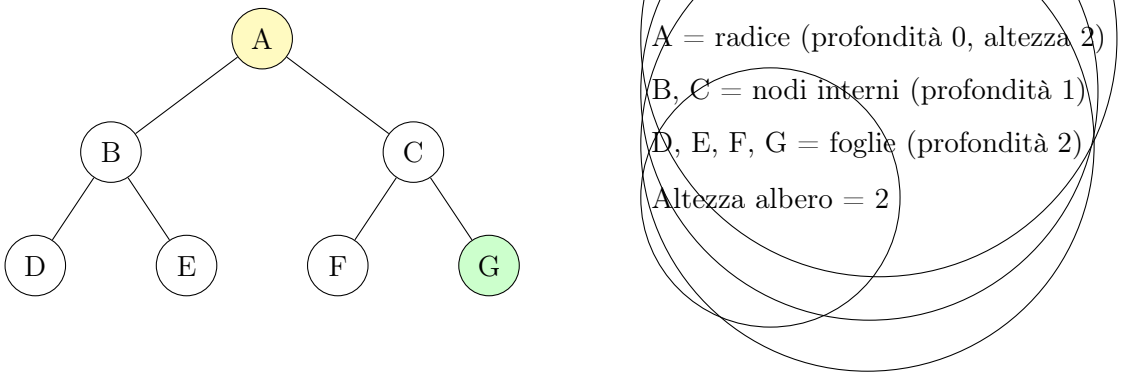
Definizione 4.1 (Albero)**.** *Un **albero** è un grafo connesso aciclico. Equivalentemente, è una collezione di nodi con le seguenti proprietà:*

- *Esiste un nodo speciale chiamato **radice** (root)*
- *Ogni nodo diverso dalla radice ha esattamente un **padre***
- *Non ci sono cicli*

Definizione 4.2 (Terminologia degli alberi)**.**

- ***Radice**: Nodo senza padre*
- ***Foglia**: Nodo senza figli*
- ***Nodo interno**: Nodo con almeno un figlio*
- ***Sottoalbero**: Albero formato da un nodo e tutti i suoi discendenti*
- ***Profondità** di un nodo: Numero di archi dalla radice al nodo*
- ***Altezza** di un nodo: Numero massimo di archi dal nodo a una foglia*
- ***Altezza dell'albero**: Altezza della radice*
- ***Livello** k: Insieme di nodi a profondità k*

Visualizzazione:



Teorema 4.1 (Proprietà degli alberi)**.** *Sia T un albero con n nodi. Allora:*

1. *T ha esattamente $n-1$ archi*
2. *Esiste un unico cammino tra qualsiasi coppia di nodi*
3. *Rimuovendo un qualsiasi arco, T diventa disconnesso*
4. *Aggiungendo un arco tra due nodi, si crea un ciclo*

Dimostrazione del punto 1. Per induzione su n.

Caso base: $n = 1$. Un albero con un solo nodo (la radice) ha $1 - 1 = 0$ archi.

Passo induttivo: Assumiamo la proprietà vera per alberi con k nodi. Consideriamo un albero T con $k+1$ nodi. Rimuoviamo una foglia v (che esiste sempre se $k+1 > 1$). L'albero rimanente T' ha k nodi e quindi, per ipotesi induttiva, ha $k-1$ archi. Aggiungendo la foglia v con il suo arco, otteniamo $k - 1 + 1 = k = (k+1) - 1$ archi. □

4.3 Alberi binari

Definizione 4.3 (Albero binario)**.** *Un **albero binario** è un albero in cui ogni nodo ha al massimo due figli, distinti come **figlio sinistro** e **figlio destro**.*

Struttura del nodo:

```
class NodoBinario:
    def __init__(self, valore):
        self.valore = valore
        self.sinistro = None
        self.destro = None
```

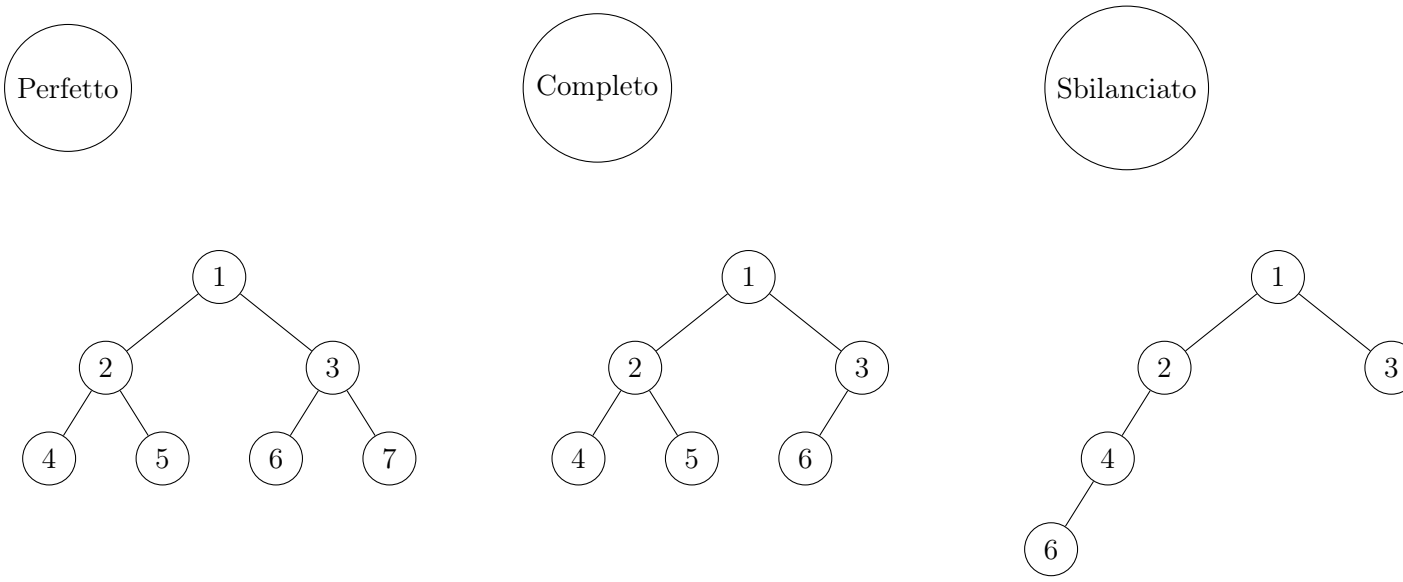
4.3.1 Tipi di alberi binari

Definizione 4.4 (Albero binario completo)**.** *Un albero binario è **completo** se tutti i livelli sono completamente riempiti, eccetto eventualmente l'ultimo, che è riempito da sinistra a destra.*

Definizione 4.5 (Albero binario perfetto)**.** *Un albero binario è **perfetto** se tutti i nodi interni hanno esattamente due figli e tutte le foglie sono allo stesso livello.*

Definizione 4.6 (Albero binario bilanciato)**.** *Un albero binario è* ***bilanciato*** *se per ogni nodo, le altezze dei suoi sottoalberi sinistro e destro differiscono al massimo di 1.*

Visualizzazione:



Teorema 4.2 (Proprietà degli alberi binari perfetti)**.** *Un albero binario perfetto di altezza h ha:*

- $2^{h+1} - 1$ *nodi totali*
- 2^h *foglie*
- $2^h - 1$ *nodi interni*

Dimostrazione. Il numero di nodi a livello k è 2^k (per $k = 0, 1, \ldots, h$).

Numero totale di nodi:

$$n = \sum_{k=0}^{h} 2^k = 2^{h+1} - 1$$

Le foglie sono tutte al livello h: 2^h foglie.

I nodi interni sono ai livelli $0, \ldots, h-1$:

$$\sum_{k=0}^{h-1} 2^k = 2^h - 1$$

□

4.3.2 Visite di alberi binari

Le visite (o attraversamenti) sono algoritmi fondamentali per processare tutti i nodi di un albero.

Visita in pre-ordine (Pre-order)

Ordine: **Radice → Sinistro → Destro**

```
def PreOrdine(nodo):
    """
    Visita in pre-ordine
    Complessità: O(n)
```

```
    """
    if nodo == None:
        return

    print(nodo.valore)        // Processa radice
    PreOrdine(nodo.sinistro) // Ricorsione a sinistra
    PreOrdine(nodo.destro)   // Ricorsione a destra
```

Visita in-ordine (In-order)

Ordine: **Sinistro → Radice → Destro**

```
def InOrdine(nodo):
    """
    Visita in-ordine
    Complessità: O(n)
    """
    if nodo == None:
        return

    InOrdine(nodo.sinistro)  // Ricorsione a sinistra
    print(nodo.valore)        // Processa radice
    InOrdine(nodo.destro)    // Ricorsione a destra
```

Visita post-ordine (Post-order)

Ordine: **Sinistro → Destro → Radice**

```
def PostOrdine(nodo):
    """
    Visita post-ordine
    Complessità: O(n)
    """
    if nodo == None:
        return

    PostOrdine(nodo.sinistro) // Ricorsione a sinistra
    PostOrdine(nodo.destro)   // Ricorsione a destra
    print(nodo.valore)        // Processa radice
```

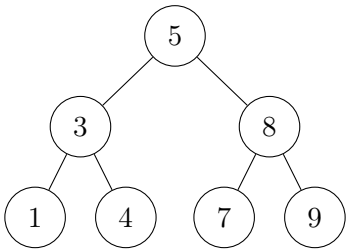
Visita per livelli (Level-order / BFS)

Usa una coda per visitare livello per livello.

```
def VisitaLivelli(radice):
    """
    Visita per livelli (BFS)
    Complessità: O(n)
    """
    if radice == None:
        return

    coda = Queue()
    coda.enqueue(radice)

    while not coda.isEmpty():
        nodo = coda.dequeue()
```

```
14          print(nodo.valore)
15
16          if nodo.sinistro != None:
17              coda.enqueue(nodo.sinistro)
18
19          if nodo.destro != None:
20              coda.enqueue(nodo.destro)
```

Esempio di visite:



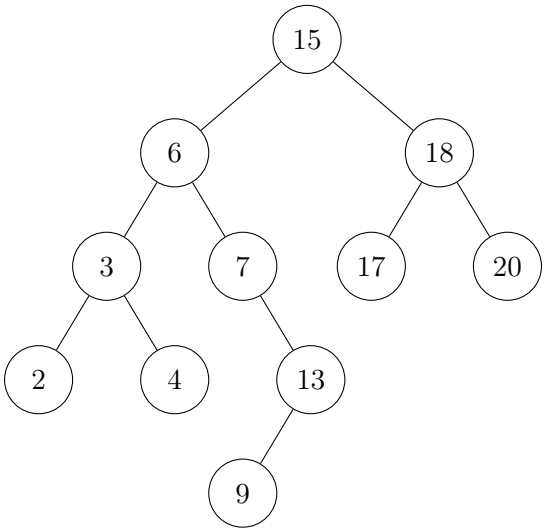
- Pre-ordine: 5, 3, 1, 4, 8, 7, 9
- In-ordine: 1, 3, 4, 5, 7, 8, 9
- Post-ordine: 1, 4, 3, 7, 9, 8, 5
- Per livelli: 5, 3, 8, 1, 4, 7, 9

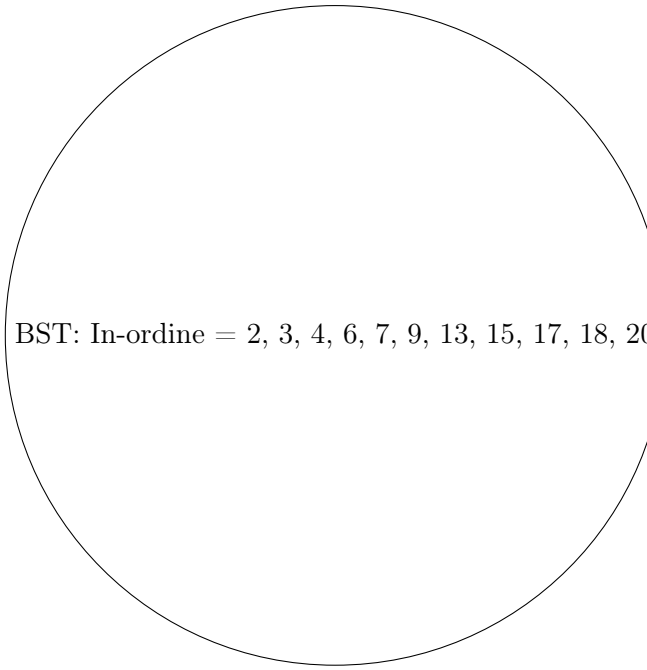
4.4 Alberi binari di ricerca (BST)

Definizione 4.7 (Albero binario di ricerca)**.** *Un **albero binario di ricerca** (BST) è un albero binario in cui, per ogni nodo x:*

- *Tutti i nodi nel sottoalbero sinistro di x hanno valori $\leq x.valore$*
- *Tutti i nodi nel sottoalbero destro di x hanno valori $> x.valore$*

Proprietà fondamentale: La visita in-ordine di un BST produce i valori in ordine crescente.





4.4.1 Operazioni su BST

Ricerca

```
def Ricerca(nodo, chiave):
    """
    Cerca una chiave nel BST
    Input: radice, chiave da cercare
    Output: nodo se trovato, None altrimenti
    Complessità: O(h) dove h = altezza
    """
    if nodo == None or nodo.valore == chiave:
        return nodo

    if chiave < nodo.valore:
        return Ricerca(nodo.sinistro, chiave)
    else:
        return Ricerca(nodo.destro, chiave)
```

Complessità:

- Caso migliore: $O(1)$ (chiave alla radice)
- Caso peggiore: $O(h)$ dove h è l'altezza
- Se bilanciato: $O(\log n)$
- Se sbilanciato (lista): $O(n)$

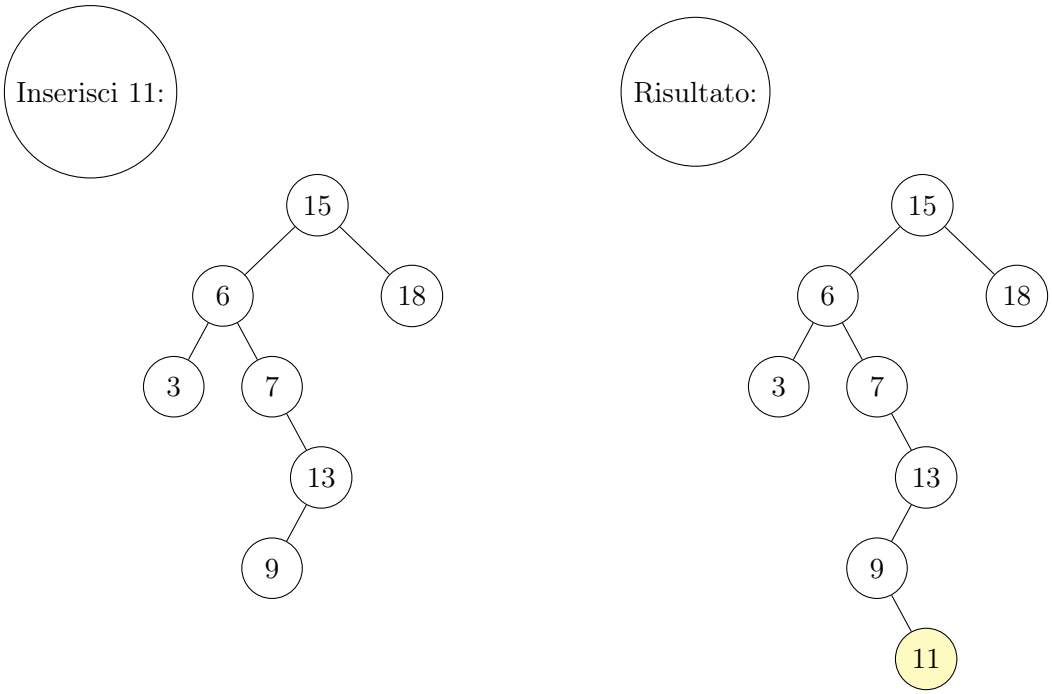
Minimo e massimo

```
def Minimo(nodo):
    """
    Trova il valore minimo (nodo più a sinistra)
    Complessità: O(h)
    """
    while nodo.sinistro != None:
        nodo = nodo.sinistro
    return nodo

def Massimo(nodo):
    """
    Trova il valore massimo (nodo più a destra)
    Complessità: O(h)
    """
    while nodo.destro != None:
        nodo = nodo.destro
    return nodo
```

Inserimento

```
def Inserisci(nodo, chiave):
    """
    Inserisce una chiave nel BST
    Complessità: O(h)
    """
    if nodo == None:
        return NodoBinario(chiave)

    if chiave < nodo.valore:
        nodo.sinistro = Inserisci(nodo.sinistro, chiave)
    elif chiave > nodo.valore:
        nodo.destro = Inserisci(nodo.destro, chiave)
    // Se chiave == nodo.valore, ignoriamo (no duplicati)

    return nodo
```

Esempio di inserimento:



Cancellazione

La cancellazione è l'operazione più complessa. Tre casi:

1. **Nodo foglia**: Rimuoviamo semplicemente il nodo
2. **Nodo con un figlio**: Sostituiamo il nodo con il suo unico figlio
3. **Nodo con due figli**: Sostituiamo il nodo con il suo *successore* (il minimo del sottoalbero destro) o *predecessore* (il massimo del sottoalbero sinistro)

```
def Cancella(nodo, chiave):
    """
    Cancella una chiave dal BST
    Complessità: O(h)
    """
    if nodo == None:
        return None

    if chiave < nodo.valore:
        nodo.sinistro = Cancella(nodo.sinistro, chiave)
    elif chiave > nodo.valore:
        nodo.destro = Cancella(nodo.destro, chiave)
    else:
        // Nodo trovato, cancelliamolo
        // Caso 1: Foglia o un solo figlio
        if nodo.sinistro == None:
            return nodo.destro
        elif nodo.destro == None:
            return nodo.sinistro

        // Caso 2: Due figli
        // Trova il successore (minimo del sottoalbero destro)
        successore = Minimo(nodo.destro)
        nodo.valore = successore.valore
```

```
25          nodo.destro = Cancella(nodo.destro, successore.valore)
26
27      return nodo
```

4.4.2 Analisi delle prestazioni dei BST

Operazione	Caso medio	Caso peggiore
Ricerca	$O(\log n)$	$O(n)$
Inserimento	$O(\log n)$	$O(n)$
Cancellazione	$O(\log n)$	$O(n)$
Minimo/Massimo	$O(\log n)$	$O(n)$

Il caso peggiore si verifica quando l'albero è completamente sbilanciato (degenera in lista).

4.5 Alberi AVL (Auto-bilancianti)

Il problema dei BST sbilanciati è risolto dagli **alberi AVL**, che mantengono l'albero bilanciato automaticamente.

Definizione 4.8 (Albero AVL)**.** *Un albero AVL è un BST in cui, per ogni nodo, le altezze dei sottoalberi sinistro e destro differiscono al massimo di 1.*

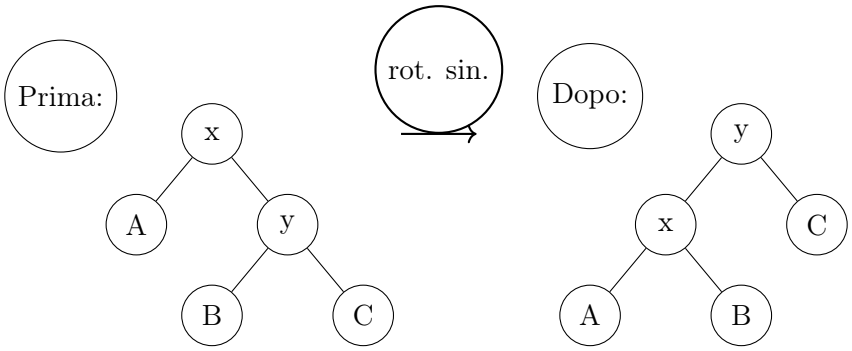
Fattore di bilanciamento:

$$BF(nodo) = \text{altezza(sinistro)} - \text{altezza(destro)} \in \{-1, 0, 1\}$$

4.5.1 Rotazioni

Per mantenere il bilanciamento, gli alberi AVL usano le **rotazioni**.

Rotazione sinistra (Left Rotation)




```
1   def RotazioneSinistra(x):
2       """
3       Rotazione sinistra attorno a x
4       Complessità: O(1)
5       """
6       y = x.destro
7       B = y.sinistro
8
9       // Effettua rotazione
10      y.sinistro = x
11      x.destro = B
```

```
12
13      // Aggiorna altezze
14      x.altezza = 1 + max(altezza(x.sinistro), altezza(x.destro))
15      y.altezza = 1 + max(altezza(y.sinistro), altezza(y.destro))
16
17      return y  // nuova radice
```

Rotazione destra (Right Rotation)

Simmetrica alla rotazione sinistra.

```
1  def RotazioneDestra(y):
2      """
3      Rotazione destra attorno a y
4      Complessità: O(1)
5      """
6      x = y.sinistro
7      B = x.destro
8
9      // Effettua rotazione
10     x.destro = y
11     y.sinistro = B
12
13     // Aggiorna altezze
14     y.altezza = 1 + max(altezza(y.sinistro), altezza(y.destro))
15     x.altezza = 1 + max(altezza(x.sinistro), altezza(x.destro))
16
17     return x  // nuova radice
```

4.5.2 Quattro casi di sbilanciamento

1. **Left-Left (LL)**: Risolto con rotazione destra
2. **Right-Right (RR)**: Risolto con rotazione sinistra
3. **Left-Right (LR)**: Risolto con rotazione sinistra sul figlio sinistro, poi rotazione destra
4. **Right-Left (RL)**: Risolto con rotazione destra sul figlio destro, poi rotazione sinistra

4.5.3 Inserimento in AVL

```
1  def InserisciAVL(nodo, chiave):
2      """
3      Inserimento in albero AVL con bilanciamento
4      Complessità: O(log n)
5      """
6      // 1. Inserimento BST normale
7      if nodo == None:
8          return NodoBinario(chiave)
9
10     if chiave < nodo.valore:
11         nodo.sinistro = InserisciAVL(nodo.sinistro, chiave)
12     else:
13         nodo.destro = InserisciAVL(nodo.destro, chiave)
14
15     // 2. Aggiorna altezza
16     nodo.altezza = 1 + max(altezza(nodo.sinistro), altezza(nodo.destro))
```

```
17
18      // 3. Calcola fattore di bilanciamento
19      bf = altezza(nodo.sinistro) - altezza(nodo.destro)
20
21      // 4. Bilancia se necessario
22      // Caso Left-Left
23      if bf > 1 and chiave < nodo.sinistro.valore:
24          return RotazioneDestra(nodo)
25
26      // Caso Right-Right
27      if bf < -1 and chiave > nodo.destro.valore:
28          return RotazioneSinistra(nodo)
29
30      // Caso Left-Right
31      if bf > 1 and chiave > nodo.sinistro.valore:
32          nodo.sinistro = RotazioneSinistra(nodo.sinistro)
33          return RotazioneDestra(nodo)
34
35      // Caso Right-Left
36      if bf < -1 and chiave < nodo.destro.valore:
37          nodo.destro = RotazioneDestra(nodo.destro)
38          return RotazioneSinistra(nodo)
39
40      return nodo
```

Teorema 4.3 (Complessità AVL)**.** *In un albero AVL con n nodi:*

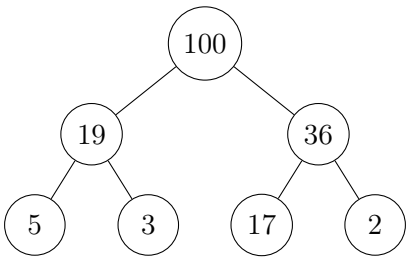
- *L'altezza è $O(\log n)$*
- *Ricerca, inserimento, cancellazione richiedono $O(\log n)$ nel caso peggiore*

4.6 Heap

Definizione 4.9 (Heap)**.** *Un **heap** è un albero binario quasi completo che soddisfa la **proprietà di heap**:*

- ***Max-heap**: Ogni nodo ha valore $\geq$ dei suoi figli*
- ***Min-heap**: Ogni nodo ha valore $\leq$ dei suoi figli*

Visualizzazione di un max-heap:



4.6.1 Rappresentazione con array

Gli heap sono implementati efficientemente con array, sfruttando la proprietà di essere quasi completi.

Per un nodo in posizione i (partendo da 1):

- Padre: $\lfloor i/2 \rfloor$
- Figlio sinistro: $2i$
- Figlio destro: $2i + 1$

A:	100	19	36	5	3	17	2
	1	2	3	4	5	6	7

4.6.2 Operazioni su heap

Heapify (Ripristina proprietà heap)

```
def MaxHeapify(A, i, heap_size):
    """
    Ripristina la proprietà di max-heap
    Assumendo che i sottoalberi siano già heap validi
    Complessità: O(log n)
    """
    sinistro = 2 * i
    destro = 2 * i + 1
    massimo = i

    if sinistro <= heap_size and A[sinistro] > A[massimo]:
        massimo = sinistro

    if destro <= heap_size and A[destro] > A[massimo]:
        massimo = destro

    if massimo != i:
        scambia A[i] con A[massimo]
        MaxHeapify(A, massimo, heap_size)
```

Costruzione heap

```
def CostruisciMaxHeap(A, n):
    """
    Costruisce un max-heap da un array non ordinato
    Complessità: O(n)  // Non O(n log n)!
    """
    heap_size = n
    // Partiamo dall'ultimo nodo non-foglia e andiamo indietro
    for i = n/2 down to 1:
        MaxHeapify(A, i, heap_size)
```

Teorema 4.4 (Complessità di CostruisciMaxHeap)**.** *La costruzione di un heap da un array di n elementi richiede $O(n)$ tempo.*

Idea. Il numero di nodi a livello h è al massimo $\lceil n/2^{h+1} \rceil$, e l'altezza di questi nodi è h.

Il costo totale è:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil \cdot O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

dove abbiamo usato $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$. □

Inserimento

```
def InserisciHeap(A, heap_size, chiave):
    """
    Inserisce una chiave nel max-heap
    Complessità: O(log n)
    """
    heap_size = heap_size + 1
    i = heap_size
    A[i] = -

    // Risali verso la radice finché la proprietà heap è violata
    while i > 1 and A[i/2] < chiave:
        A[i] = A[i/2]
        i = i / 2

    A[i] = chiave
    return heap_size
```

Estrazione del massimo

```
def EstraiMassimo(A, heap_size):
    """
    Estrae e restituisce il massimo (radice)
    Complessità: O(log n)
    """
    if heap_size < 1:
        errore "Heap underflow"

    max = A[1]
    A[1] = A[heap_size]
    heap_size = heap_size - 1
    MaxHeapify(A, 1, heap_size)

    return max, heap_size
```

4.6.3 Applicazione: Heap Sort

```
def HeapSort(A, n):
    """
    Ordinamento con heap
    Complessità: O(n log n) tempo, O(1) spazio
    """
    CostruisciMaxHeap(A, n)
    heap_size = n

    for i = n down to 2:
        scambia A[1] con A[i]
        heap_size = heap_size - 1
        MaxHeapify(A, 1, heap_size)
```

Complessità: $O(n \log n)$ nel caso peggiore, $O(1)$ spazio ausiliario.

4.7 Tabella riassuntiva

Operazione	**BST**	**AVL**	**Heap**
Ricerca	$O(n)$ worst	$O(\log n)$	$O(n)$
Inserimento	$O(n)$ worst	$O(\log n)$	$O(\log n)$
Cancellazione	$O(n)$ worst	$O(\log n)$	$O(\log n)$
Trova min/max	$O(n)$ worst	$O(\log n)$	$O(1)$
Costruzione	$O(n \log n)$	$O(n \log n)$	$O(n)$
In-order traversal	Ordinato	Ordinato	Non ordinato

4.8 Conclusioni

Gli alberi sono strutture dati potentissime con applicazioni vastissime:

- **BST**: Dizionari, insiemi ordinati, database indicizzati
- **AVL**: Quando servono garanzie worst-case
- **Heap**: Code con priorità, HeapSort, algoritmi su grafi (Dijkstra, Prim)

Punti chiave:

- BST: In-order produce sequenza ordinata
- AVL: Bilanciamento automatico con rotazioni
- Heap: Proprietà di heap, rappresentazione con array
- Complessità dipende dall'altezza dell'albero
- Alberi bilanciati garantiscono $O(\log n)$

Capitolo 5

Grafi

5.1 Introduzione

I grafi sono tra le strutture dati più generali e potenti in informatica. Modellano relazioni tra oggetti e sono alla base di innumerevoli applicazioni: reti sociali, mappe stradali, reti di computer, dipendenze tra task, circuiti elettronici, molecole chimiche, e molto altro.

In questo capitolo studieremo le definizioni formali, le rappresentazioni in memoria, e gli algoritmi fondamentali di visita e ricerca di cammini.

5.2 Definizioni fondamentali

Definizione 5.1 (Grafo)**.** *Un* ***grafo*** $G = (V, E)$ *è una coppia ordinata dove:*

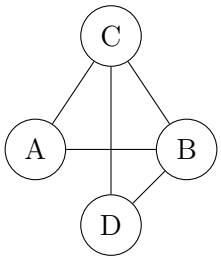
- V *è un insieme finito di* ***vertici*** *(o nodi)*
- $E \subseteq V \times V$ *è un insieme di* ***archi*** *(o spigoli)*

Definizione 5.2 (Grafo orientato vs non orientato)**.**

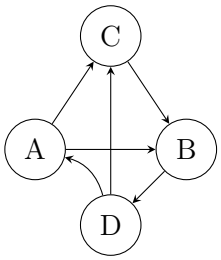
- *Un grafo è* ***orientato*** *(diretto) se gli archi hanno una direzione:* $(u, v) \neq (v, u)$
- *Un grafo è* ***non orientato*** *se gli archi non hanno direzione:* $(u, v) = (v, u)$

Visualizzazione:

Non orientato



Orientato



Definizione 5.3 (Terminologia dei grafi)**.**

- ***Adiacenza****: Due vertici* u, v *sono adiacenti se esiste un arco* $(u, v) \in E$
- ***Grado di un vertice****: Numero di archi incidenti*
  - *Grafo orientato: grado entrante (in-degree), grado uscente (out-degree)*
- ***Cammino****: Sequenza di vertici* $v_0, v_1, \ldots, v_k$ *tale che* $(v_i, v_{i+1}) \in E$ *per* $i = 0, \ldots, k-1$
- ***Lunghezza di un cammino****: Numero di archi nel cammino*

- ***Cammino semplice**: Cammino senza vertici ripetuti*
- ***Ciclo**: Cammino dove $v_0 = v_k$*
- ***Grafo connesso**: Esiste un cammino tra ogni coppia di vertici*
- ***Componente connessa**: Sottografo massimale connesso*
- ***Grafo pesato**: Ogni arco ha un peso $w(u, v)$*
- ***Albero**: Grafo connesso aciclico*
- ***Foresta**: Collezione di alberi disgiunti*

Teorema 5.1 (Somma dei gradi)**.** *Per un grafo $G = (V, E)$:*

$$\sum_{v \in V} \deg(v) = 2|E|$$

Dimostrazione. Ogni arco (u, v) contribuisce 1 al grado di u e 1 al grado di v, quindi contribuisce 2 alla somma totale. Sommando su tutti gli archi otteniamo $2|E|$. □

Teorema 5.2 (Numero di archi)**.** *In un grafo con n vertici:*

- *Grafo non orientato: al massimo $\binom{n}{2} = \frac{n(n-1)}{2}$ archi*
- *Grafo orientato: al massimo $n(n-1)$ archi*

5.3 Rappresentazioni dei grafi

Esistono due rappresentazioni principali: matrice di adiacenza e lista di adiacenza.

5.3.1 Matrice di adiacenza

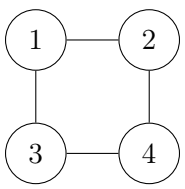
Definizione 5.4 (Matrice di adiacenza)**.** *Per un grafo $G = (V, E)$ con $n = |V|$ vertici, la **matrice di adiacenza** A è una matrice $n \times n$ dove:*

$$A[i][j] = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{altrimenti} \end{cases}$$

Per grafi pesati:

$$A[i][j] = \begin{cases} w(i, j) & \text{se } (i, j) \in E \\ \infty & \text{altrimenti} \end{cases}$$

Esempio:



Matrice di adiacenza:

	1	2	3	4
1	0	1	1	0
2	1	0	0	1
3	1	0	0	1
4	0	1	1	0

Proprietà:

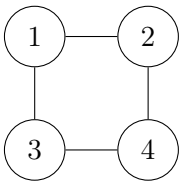
- Spazio: $\Theta(n^2)$

- Verifica se $(u, v) \in E$: $O(1)$
- Trovare tutti i vicini di v: $O(n)$
- Adatta per grafi densi ($|E| \approx n^2$)
- Per grafi non orientati, la matrice è simmetrica

5.3.2 Lista di adiacenza

Definizione 5.5 (Lista di adiacenza)**.** *Per un grafo $G = (V, E)$, la **lista di adiacenza** è un array di $|V|$ liste, dove la lista $Adj[v]$ contiene tutti i vertici adiacenti a v.*

Esempio (stesso grafo):



Liste di adiacenza:

1	2	3
2	1	4
3	1	4
4	2	3

Proprietà:

- Spazio: $\Theta(n + m)$ dove $m = |E|$
- Verifica se $(u, v) \in E$: $O(\deg(u))$
- Trovare tutti i vicini di v: $O(\deg(v))$
- Adatta per grafi sparsi ($|E| \ll n^2$)

5.3.3 Confronto

Operazione	**Matrice**	**Lista**
Spazio	$O(n^2)$	$O(n + m)$
Verifica arco	$O(1)$	$O(\deg(v))$
Trova vicini	$O(n)$	$O(\deg(v))$
Aggiungi vertice	$O(n^2)$	$O(1)$
Aggiungi arco	$O(1)$	$O(1)$
Rimuovi arco	$O(1)$	$O(\deg(v))$
Migliore per	Grafi densi	Grafi sparsi

5.4 Visita in ampiezza (BFS)

Definizione 5.6 (Breadth-First Search)**.** *La **visita in ampiezza** (BFS) esplora il grafo livello per livello, partendo da un vertice sorgente s: prima visita tutti i vicini di s, poi i vicini dei vicini, e così via.*

Struttura dati: Coda FIFO

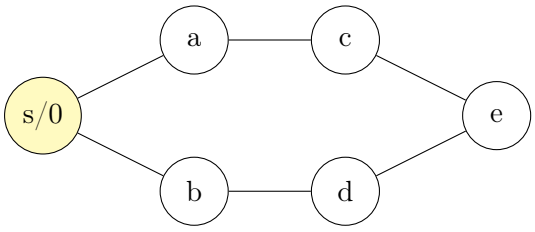
```
1 def BFS(G, s):
2     """
3     Visita in ampiezza da s
4     Input: grafo G, vertice sorgente s
5     Output: distanze e predecessori
```

```
    Complessità: O(V + E) con liste di adiacenza
    """
    // Inizializzazione
    for ogni vertice u in G.V:
        u.colore = BIANCO
        u.distanza =
        u.padre = None

    s.colore = GRIGIO
    s.distanza = 0
    s.padre = None

    Q = Queue()
    Q.enqueue(s)

    while not Q.isEmpty():
        u = Q.dequeue()

        for ogni v in G.Adj[u]:
            if v.colore == BIANCO:
                v.colore = GRIGIO
                v.distanza = u.distanza + 1
                v.padre = u
                Q.enqueue(v)

        u.colore = NERO

    return distanze, padri
```

Colori:

- BIANCO: Non ancora scoperto
- GRIGIO: Scoperto ma non completamente esplorato
- NERO: Completamente esplorato

Esempio di esecuzione:



Distanze da s:
s: 0 (livello 0)
a, b: 1 (livello 1)
c, d: 2 (livello 2)
e: 3 (livello 3)

Teorema 5.3 (Correttezza di BFS)**.** *Sia $G = (V, E)$ un grafo e $s \in V$ il vertice sorgente. Allora:*

1. *BFS visita tutti i vertici raggiungibili da s*
2. *Per ogni v raggiungibile da s, v.distanza è la distanza minima da s a v (numero minimo di archi)*
3. *L'albero dei padri BFS rappresenta i cammini minimi da s*

Idea. Per induzione sulla distanza da s. BFS procede per livelli, quindi scopre prima tutti i vertici a distanza k prima di scoprire vertici a distanza $k + 1$. □

Applicazioni di BFS:

- Trovare il cammino più breve (non pesato)
- Testare se un grafo è connesso
- Trovare componenti connesse
- Testare se un grafo è bipartito
- Web crawling
- Sistemi di raccomandazione (amici di amici)

5.5 Visita in profondità (DFS)

Definizione 5.7 (Depth-First Search)**.** *La **visita in profondità** (DFS) esplora il grafo andando "in profondità" il più possibile prima di backtrackare.*

Struttura dati: Stack (esplicito o tramite ricorsione)

```
def DFS(G):
    """
    Visita in profondità di tutto il grafo
    Complessità: O(V + E)
    """
    for ogni vertice u in G.V:
        u.colore = BIANCO
        u.padre = None

    tempo = 0

    for ogni vertice u in G.V:
        if u.colore == BIANCO:
            DFS_Visit(G, u)

def DFS_Visit(G, u):
    """
    Visita ricorsiva da u
    """
    tempo = tempo + 1
    u.tempo_scoperta = tempo
    u.colore = GRIGIO

    for ogni v in G.Adj[u]:
        if v.colore == BIANCO:
            v.padre = u
            DFS_Visit(G, v)

    u.colore = NERO
    tempo = tempo + 1
    u.tempo_fine = tempo
```

Timestamp:

- $u.tempo\_scoperta$: Quando u viene scoperto
- $u.tempo\_fine$: Quando l'esplorazione di u termina

Visualizzazione:

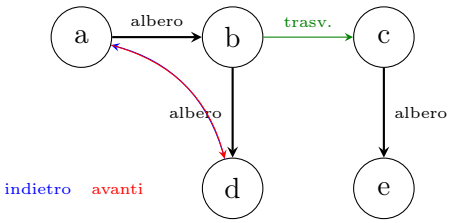
Teorema 5.4 (Proprietà della parentesizzazione)**.** *Per ogni coppia di vertici u, v, vale esattamente una delle seguenti:*

1. *Gli intervalli $[u.d, u.f]$ e $[v.d, v.f]$ sono disgiunti (u e v non sono antenati l'uno dell'altro)*
2. *$[u.d, u.f] \subset [v.d, v.f]$ (u è discendente di v)*
3. *$[v.d, v.f] \subset [u.d, u.f]$ (v è discendente di u)*

5.5.1 Classificazione degli archi

Durante DFS, ogni arco (u, v) può essere classificato:

1. **Arco dell'albero**: v è scoperto da u (parte della foresta DFS)
2. **Arco all'indietro**: v è un antenato di u (crea un ciclo)
3. **Arco in avanti**: v è un discendente di u (ma non nell'albero DFS)
4. **Arco trasversale**: Tutti gli altri casi



Teorema 5.5 (Rilevazione di cicli)**.** *Un grafo orientato ha un ciclo se e solo se DFS trova un arco all'indietro.*

Applicazioni di DFS:

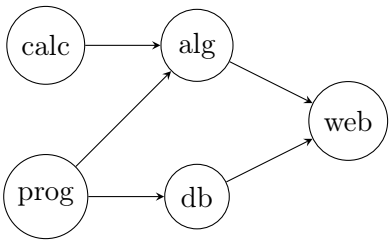
- Rilevazione di cicli
- Ordinamento topologico
- Componenti fortemente connesse
- Risoluzione di labirinti
- Analisi di dipendenze

5.6 Ordinamento topologico

Definizione 5.8 (Ordinamento topologico)**.** *Un **ordinamento topologico** di un grafo orientato aciclico (DAG) è un ordinamento lineare dei vertici tale che per ogni arco (u, v), u appare prima di v nell'ordinamento.*

```
def OrdinamentoTopologico(G):
    """
    Ordinamento topologico usando DFS
    Precondizione: G è un DAG
    Complessità: O(V + E)
    """
    lista = []

    def DFS_Visit_Topo(u):
        u.colore = GRIGIO

        for ogni v in G.Adj[u]:
            if v.colore == BIANCO:
                DFS_Visit_Topo(v)

        u.colore = NERO
        lista.prepend(u)  // Aggiungi in testa

    for ogni vertice u in G.V:
        u.colore = BIANCO

    for ogni vertice u in G.V:
        if u.colore == BIANCO:
            DFS_Visit_Topo(u)

    return lista
```

Esempio:



Ordinamento possibile: prog, calc, db, alg, web

5.7 Cammini minimi

5.7.1 Algoritmo di Dijkstra

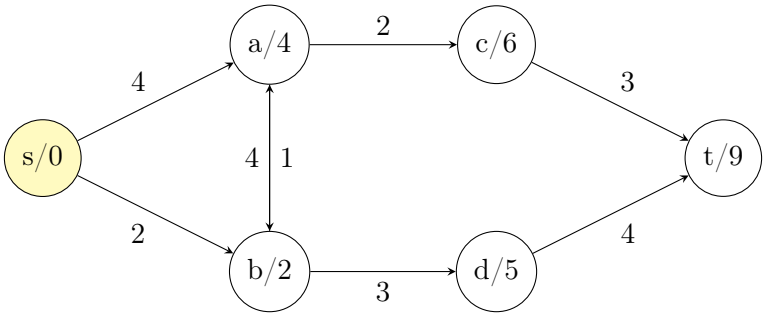
L'algoritmo di Dijkstra trova i cammini minimi da una sorgente s a tutti gli altri vertici in un grafo con pesi **non negativi**.

Idea: Espansione greedy. Manteniamo un insieme S di vertici per cui conosciamo già la distanza minima. Ad ogni iterazione, aggiungiamo il vertice $u \notin S$ con distanza minima.

```
def Dijkstra(G, w, s):
```

```
    """
    Cammini minimi da s con pesi non negativi
    Input: grafo G, funzione peso w, sorgente s
    Output: distanze minime da s
    Complessità: O((V + E) log V) con min-heap
    """
    // Inizializzazione
    for ogni vertice v in G.V:
        v.distanza =
        v.padre = None

    s.distanza = 0

    // Coda con priorità (min-heap)
    Q = MinPriorityQueue(G.V)  // priorità = distanza

    while not Q.isEmpty():
        u = Q.extractMin()

        for ogni v in G.Adj[u]:
            // Rilassamento
            if v.distanza > u.distanza + w(u, v):
                v.distanza = u.distanza + w(u, v)
                v.padre = u
                Q.decreaseKey(v, v.distanza)

    return distanze
```

Esempio:



Etichette: vertice/distanza_minima_da_s

Teorema 5.6 (Correttezza di Dijkstra)**.** *Se tutti i pesi sono non negativi, l'algoritmo di Dijkstra calcola correttamente le distanze minime dalla sorgente.*

Idea. Per induzione sull'insieme S dei vertici processati. Quando aggiungiamo u a S, $u.distanza$ è già il valore minimo perché tutti i cammini alternativi passano attraverso vertici non ancora in S, che hanno distanza $\geq u.distanza$, e tutti i pesi sono non negativi. □

5.7.2 Algoritmo di Bellman-Ford

Bellman-Ford gestisce anche pesi negativi e rileva cicli di peso negativo.

```
def BellmanFord(G, w, s):
    """
    Cammini minimi anche con pesi negativi
```

```
    Rileva cicli di peso negativo
    Complessità: O(VE)
    """
    // Inizializzazione
    for ogni vertice v in G.V:
        v.distanza =
        v.padre = None

    s.distanza = 0

    // Rilassamento ripetuto
    for i = 1 to |G.V| - 1:
        for ogni arco (u, v) in G.E:
            if v.distanza > u.distanza + w(u, v):
                v.distanza = u.distanza + w(u, v)
                v.padre = u

    // Controllo cicli negativi
    for ogni arco (u, v) in G.E:
        if v.distanza > u.distanza + w(u, v):
            return "Ciclo di peso negativo rilevato"

    return distanze
```

Complessità: $O(VE)$ — meno efficiente di Dijkstra, ma più generale.

5.8 Tabella riassuntiva

Algoritmo	Complessità	Uso
BFS	$O(V+E)$	Cammini minimi non pesati
DFS	$O(V+E)$	Cicli, ordinamento topologico
Dijkstra	$O((V+E)\log V)$	Cammini minimi (pesi ≥ 0)
Bellman-Ford	$O(VE)$	Cammini minimi (pesi qualsiasi)

5.9 Esercizi

5.9.1 Esercizio 1

Dimostrare che un albero con n vertici ha esattamente $n-1$ archi.

5.9.2 Esercizio 2

Scrivere un algoritmo per verificare se un grafo non orientato è bipartito usando BFS.

5.9.3 Esercizio 3

Implementare l'algoritmo di Dijkstra usando una coda con priorità basata su heap.

5.9.4 Esercizio 4

Dato un grafo orientato, trovare le componenti fortemente connesse usando DFS.

5.9.5 Esercizio 5

Dimostrare che l'algoritmo di Bellman-Ford è corretto.

5.10 Conclusioni

I grafi sono strutture dati fondamentali con applicazioni vastissime:

- **BFS**: Cammini minimi, livelli, bipartitismo
- **DFS**: Cicli, ordinamento topologico, componenti connesse
- **Dijkstra**: Navigatori GPS, routing di rete
- **Bellman-Ford**: Protocolli di routing, arbitraggio valutario

Punti chiave:

- Liste di adiacenza per grafi sparsi, matrici per grafi densi
- BFS usa coda (FIFO), DFS usa stack (ricorsione)
- BFS trova cammini minimi non pesati
- Dijkstra richiede pesi non negativi
- Bellman-Ford gestisce pesi negativi ma è più lento
- Ordinamento topologico esiste solo per DAG

I grafi sono alla base di molti algoritmi avanzati: minimum spanning trees (Kruskal, Prim), flussi massimi (Ford-Fulkerson), matching, colorazione, e molto altro.

Capitolo 6

Tabelle Hash

6.1 Introduzione

Le tabelle hash sono tra le strutture dati più utilizzate in informatica pratica. Offrono tempi di ricerca, inserimento e cancellazione **medi** $O(1)$, prestazioni inarrivabili da altre strutture dati per operazioni di dizionario.

Un dizionario (o mappa) è una collezione di coppie (chiave, valore) che supporta le operazioni:

- `insert(chiave, valore)`: Inserisce una coppia
- `search(chiave)`: Cerca e restituisce il valore associato
- `delete(chiave)`: Rimuove la coppia

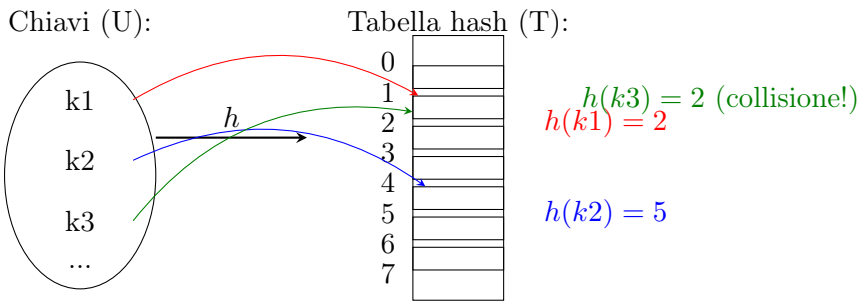
Le tabelle hash implementano efficientemente queste operazioni tramite una **funzione hash** che mappa le chiavi in posizioni di un array.

6.2 Concetti fondamentali

Definizione 6.1 (Funzione hash)**.** *Una* ***funzione hash*** *è una funzione* $h : U \to \{0, 1, \ldots, m-1\}$ *che mappa elementi di un universo* U *(potenzialmente molto grande) in un insieme di dimensione* m *(relativamente piccola) di posizioni in un array chiamato* ***tabella hash****.*

Definizione 6.2 (Collisione)**.** *Una* ***collisione*** *si verifica quando due chiavi diverse* $k_1 \neq k_2$ *hanno lo stesso valore hash:* $h(k_1) = h(k_2)$*.*

Visualizzazione:



Problema delle collisioni: Dato che tipicamente $|U| \gg m$, per il principio del pigeonhole, le collisioni sono inevitabili. Dobbiamo gestirle!

6.3 Funzioni hash

Una buona funzione hash deve:

1. Essere **deterministica**: Stessa chiave → stesso hash
2. Essere **veloce** da calcolare: $O(1)$
3. Distribuire uniformemente le chiavi (minimizzare collisioni)
4. Minimizzare il clustering (raggruppamento di chiavi)

6.3.1 Metodo della divisione

Definizione 6.3 (Metodo della divisione)**.**

$$h(k) = k \bmod m$$

dove m è la dimensione della tabella.

Scelta di m:

- Evitare potenze di 2 (usa solo i bit meno significativi)
- Preferire numeri primi lontani da potenze di 2
- Esempio: se si prevedono ≈ 2000 elementi, $m = 2003$ (primo)

Esempio:

$$h(123) = 123 \bmod 11 = 2$$
$$h(456) = 456 \bmod 11 = 5$$
$$h(789) = 789 \bmod 11 = 8$$

6.3.2 Metodo della moltiplicazione

Definizione 6.4 (Metodo della moltiplicazione)**.**

$$h(k) = \lfloor m \cdot ((k \cdot A) \bmod 1) \rfloor$$

dove A è una costante in $(0, 1)$, e $(k \cdot A) \bmod 1$ estrae la parte frazionaria.

Scelta comune: $A = \frac{\sqrt{5}-1}{2} \approx 0.618$ (rapporto aureo coniugato).
Vantaggio: La scelta di m non è critica (spesso $m = 2^p$ per efficienza).

6.3.3 Hash per stringhe

Per stringhe, una funzione hash comune è:

$$h(s) = \left(\sum_{i=0}^{|s|-1} s[i] \cdot a^i \right) \bmod m$$

dove a è una costante (tipicamente un numero primo, es. 31 o 37).
Implementazione efficiente (Horner's rule):

```python
def hashStringa(s, m):
    """
    Hash per stringhe usando Horner's rule
    Complessità: O(|s|)
    """
    hash_val = 0
    a = 31  // costante moltiplicativa

    for i = 0 to len(s) - 1:
        hash_val = (hash_val * a + ord(s[i])) % m

    return hash_val
```

6.3.4 Hash universali

Definizione 6.5 (Hashing universale)**.** *Una famiglia $\mathcal{H}$ di funzioni hash è **universale** se per ogni coppia di chiavi distinte k, ℓ:*

$$P_{h \in \mathcal{H}}[h(k) = h(\ell)] \leq \frac{1}{m}$$

dove la probabilità è presa scegliendo uniformemente h da $\mathcal{H}$.

Esempio di famiglia universale:
Per chiavi intere e m primo:

$$\mathcal{H} = \{h_{a,b}(k) = ((ak + b) \bmod p) \bmod m : 1 \leq a < p, 0 \leq b < p\}$$

Teorema 6.1 (Prestazioni con hashing universale)**.** *Se usiamo una funzione hash scelta casualmente da una famiglia universale, il tempo medio per cercare una chiave è $O(1+\alpha)$ dove $\alpha = n/m$ è il **fattore di carico**.*

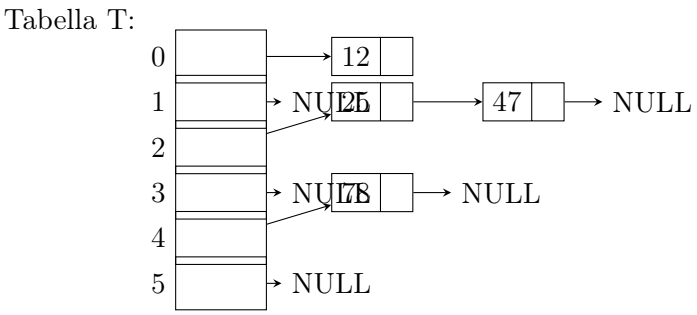
6.4 Gestione delle collisioni

Esistono due approcci principali: concatenamento (chaining) e indirizzamento aperto (open addressing).

6.4.1 Concatenamento (Chaining)

Definizione 6.6 (Concatenamento)**.** *Nel **concatenamento**, ogni posizione della tabella contiene una lista concatenata di tutte le chiavi che hanno lo stesso hash.*

Visualizzazione:



Operazioni:

```python
class HashTableChaining:
    def __init__(self, m):
        self.m = m
        self.table = array di m liste vuote

    def hash(self, k):
        return k % self.m

    def insert(self, k, v):
        """
        Inserisce (k, v)
        Complessità: O(1)
        """
        i = self.hash(k)
        self.table[i].insertFront(k, v)

    def search(self, k):
        """
        Cerca chiave k
        Complessità: O(1 +   ) in media
        """
        i = self.hash(k)
        return self.table[i].search(k)

    def delete(self, k):
        """
        Cancella chiave k
        Complessità: O(1 +   ) in media
        """
        i = self.hash(k)
        self.table[i].delete(k)
```

Analisi:
Sia $\alpha = n/m$ il **fattore di carico** (numero medio di elementi per lista).

Teorema 6.2 (Complessità del concatenamento)**.** *Con una buona funzione hash:*

- *Ricerca **senza successo**:* $\Theta(1+\alpha)$ *in media*
- *Ricerca **con successo**:* $\Theta(1+\alpha)$ *in media*
- *Inserimento:* $\Theta(1)$ *(in testa alla lista)*
- *Cancellazione:* $\Theta(1+\alpha)$ *in media*

Ricerca senza successo. Assumiamo hashing uniforme semplice: ogni chiave ha probabilità $1/m$ di finire in ogni slot.

Il numero atteso di elementi esaminati è il numero atteso di elementi nella lista, che è $\alpha = n/m$. Aggiungendo il tempo $O(1)$ per calcolare l'hash, otteniamo $O(1+\alpha)$. □

Implicazione: Se manteniamo $\alpha = O(1)$ (es. $\alpha \leq 1$), tutte le operazioni sono $O(1)$ in media!

Questo si ottiene con il **rehashing dinamico**: quando α supera una soglia (es. 0.75), raddoppiamo m e re-inseriamo tutti gli elementi.

6.4.2 Indirizzamento aperto (Open Addressing)

Definizione 6.7 (Indirizzamento aperto)**.** *Nell'**indirizzamento aperto**, tutti gli elementi sono memorizzati nell'array stesso (no liste). Quando c'è una collisione, cerchiamo la prossima posizione libera secondo una sequenza di probe.*

La funzione hash diventa:

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

dove il secondo argomento è il **numero di probe** i.
Per inserire la chiave k, proviamo le posizioni:

$$h(k, 0), h(k, 1), h(k, 2), \ldots$$

finché troviamo uno slot libero.

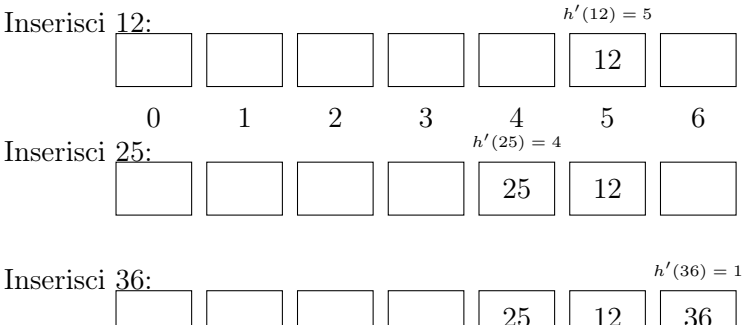
Linear Probing

Definizione 6.8 (Linear probing)**.**

$$h(k, i) = (h'(k) + i) \bmod m$$

dove h' è una funzione hash ausiliaria.

Se la posizione $h'(k)$ è occupata, proviamo $h'(k) + 1, h'(k) + 2, \ldots$ (modulo m).
Esempio: Inserimento di 12, 25, 36 con $m = 7$ e $h'(k) = k \bmod 7$.



$h'(36) = 1$ occupato, prova 2,3,4,5 occupati

Problema: *Primary clustering* – si formano lunghe sequenze contigue di celle occupate, peggiorando le prestazioni.

Quadratic Probing

Definizione 6.9 (Quadratic probing)**.**

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

dove c_1, c_2 sono costanti.

Esempio comune: $c_1 = c_2 = 1$, quindi $h(k, i) = (h'(k) + i + i^2) \bmod m$.
Vantaggio: Riduce il primary clustering.
Problema: *Secondary clustering* – chiavi con lo stesso hash iniziale seguono la stessa sequenza di probe.

Double Hashing

Definizione 6.10 (Double hashing)**.**

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

dove h_1, h_2 sono funzioni hash ausiliarie.

Requisiti:

- $h_2(k)$ deve essere relativamente primo con m (se m è primo, basta $h_2(k) \neq 0$)
- Comune: $m = 2^p$ e $h_2(k)$ dispari, oppure m primo

Esempio:

$$h_1(k) = k \bmod m$$
$$h_2(k) = 1 + (k \bmod (m-1))$$

Vantaggio: Minimizza clustering, approssima hashing uniforme.
Operazioni con open addressing:

```
def insert(T, k, v):
    """
    Inserimento con open addressing
    Complessità: O(1/(1-  )) in media
    """
    i = 0
    while i < m:
        j = h(k, i)
        if T[j] == None or T[j] == DELETED:
            T[j] = (k, v)
            return j
        i = i + 1

    errore "Tabella piena"

def search(T, k):
    """
    Ricerca con open addressing
    """
    i = 0
    while i < m:
        j = h(k, i)
        if T[j] == None:
            return None  // chiave non presente
        if T[j].key == k:
            return T[j].value
        i = i + 1

    return None
```

Cancellazione: Non possiamo semplicemente mettere `None`, altrimenti spezziamo le catene di probe. Soluzione: marcare la cella come `DELETED` (tombstone).

Teorema 6.3 (Complessità con open addressing)**.** *Con hashing uniforme, il numero atteso di probe è:*

- *Ricerca senza successo:* $\frac{1}{1-\alpha}$

- *Ricerca con successo:* $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

dove $\alpha = n/m < 1$ *è il fattore di carico.*

Osservazione: Con $\alpha = 0.5$ (tabella metà piena), ricerca senza successo richiede in media 2 probe. Con $\alpha = 0.9$, servono 10 probe!

Pratica: Mantenere $\alpha < 0.7$ per buone prestazioni.

6.5 Confronto delle tecniche

Caratteristica	**Chaining**	**Open Addressing**
Memoria extra	Puntatori	No
α può superare 1	Sì	No
Cancellazione	Semplice	Complessa (tombstones)
Località cache	Scarsa	Buona
Prestazioni con α alto	Degrada linearmente	Degrada rapidamente
Implementazione	Più semplice	Più complessa
Migliore per	α variabile	α basso, memoria limitata

6.6 Analisi formale

Definizione 6.11 (Hashing uniforme semplice)**.** *Assumiamo che ogni chiave abbia uguale probabilità* $1/m$ *di finire in ogni slot, indipendentemente da dove finiscono le altre chiavi.*

Teorema 6.4 (Lunghezza attesa delle liste nel chaining)**.** *Sotto hashing uniforme semplice, la lunghezza attesa di una lista è* $\alpha = n/m$*.*

Dimostrazione. Sia X_i il numero di elementi nella lista i. Per linearità del valore atteso:

$$E[X_i] = E\left[\sum_{j=1}^{n} \mathbb{1}[\text{chiave } j \text{ va in slot } i]\right] = \sum_{j=1}^{n} P[\text{chiave } j \text{ in slot } i] = \sum_{j=1}^{n} \frac{1}{m} = \frac{n}{m} = \alpha$$

□

Teorema 6.5 (Costo di ricerca con successo nel chaining)**.** *Il costo atteso di una ricerca con successo è* $1 + \alpha/2$*.*

Idea. In media, dobbiamo scansionare metà della lista contenente la chiave cercata. La lunghezza media è α, quindi $1 + \alpha/2$ (il $+1$ per calcolare l'hash). □

6.7 Applicazioni pratiche

Le tabelle hash sono onnipresenti:

- **Database**: Indici hash per accesso rapido
- **Compilatori**: Tabelle dei simboli
- **Caching**: Memorizzazione di risultati precedenti
- **Set e dizionari**: Python `dict`, Java `HashMap`, C++ `unordered_map`
- **Deduplicazione**: Rilevare elementi duplicati in $O(n)$
- **Crittografia**: Hash crittografici (SHA-256, MD5 – non per tabelle hash!)
- **Blockchain**: Hash dei blocchi
- **Password storage**: Hash di password (con salt)

6.8 Hash crittografici vs hash per tabelle

Differenze fondamentali:

	Hash per tabelle	**Hash crittografici**
Scopo	Distribuzione uniforme	Sicurezza
Velocità	Più veloce possibile	Può essere lento
Collisioni	Accettabili, gestite	Devono essere computazionalmente intrattabili
Reversibilità	Irrilevante	Deve essere one-way
Dimensione output	Piccola ($\log m$ bit)	Grande (256+ bit)
Esempi	$k \bmod m$, MurmurHash	SHA-256, SHA-3, BLAKE2

Attenzione: NON usare hash crittografici per tabelle hash (troppo lenti), e NON usare hash per tabelle in contesti di sicurezza!

6.9 Tabelle hash perfette

Definizione 6.12 (Perfect hashing)**.** *Un **perfect hash** è una funzione hash senza collisioni per un insieme statico di chiavi.*

Se l'insieme di chiavi è noto a priori e non cambia, possiamo costruire una tabella hash perfetta che garantisce $O(1)$ nel **caso peggiore** (non solo in media).

Schema a due livelli:

1. Prima funzione hash $h : U \to \{0, \ldots, m-1\}$
2. In ogni slot i, seconda tabella hash con $m_i = n_i^2$ slot (dove n_i è il numero di chiavi in slot i)

Teorema 6.6 (Perfect hashing)**.** *Con hashing universale a due livelli, si può costruire una tabella hash perfetta con spazio $O(n)$ e tempo di ricerca $O(1)$ worst-case.*

Applicazioni: Compilatori, riserve di parole chiave, dizionari statici.

6.10 Bloom Filters

Un'applicazione avanzata dell'hashing sono i **Bloom filters**, strutture dati probabilistiche per test di appartenenza.

Proprietà:

- Spazio: $O(n)$ bit (molto compatto)
- Inserimento: $O(k)$ dove k è il numero di funzioni hash
- Ricerca: $O(k)$
- **Falsi positivi possibili, falsi negativi NO**

Applicazioni: Database distribuiti, web caching, bioinformatica.

6.11 Esercizi

6.11.1 Esercizio 1

Progettare una funzione hash per numeri di telefono italiani (10 cifre).

6.11.2 Esercizio 2

Dimostrare che con chaining e $\alpha = 1$, il numero atteso di probe per ricerca senza successo è 1.

6.11.3 Esercizio 3

Implementare una tabella hash con double hashing.

6.11.4 Esercizio 4

Calcolare il numero atteso di collisioni quando si inseriscono n chiavi in una tabella di dimensione m.

6.11.5 Esercizio 5

Progettare una tabella hash che supporta `deleteRandom()` in $O(1)$ ammortizzato.

6.12 Conclusioni

Le tabelle hash sono strutture dati fondamentali che offrono prestazioni eccezionali:

Punti chiave:

- Operazioni in tempo medio $O(1)$ – inarrivabile per BST
- Funzione hash cruciale: deve distribuire uniformemente
- Collisioni inevitabili: gestirle con chaining o open addressing
- Fattore di carico α determina prestazioni
- Chaining: semplice, robusto a α alto
- Open addressing: compatto, buona località cache
- Rehashing dinamico per mantenere α basso
- Perfect hashing per insiemi statici: $O(1)$ worst-case

Quando usare hash table:

- Serve accesso rapido per chiave
- Non serve ordinamento
- Non serve ricerca per range
- Spazio non è critico

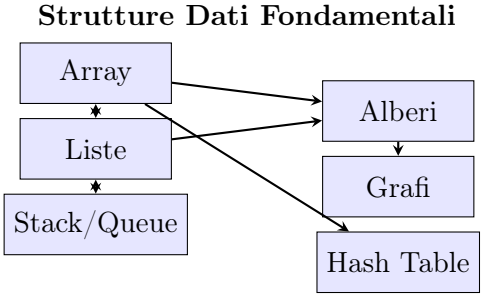
Quando usare BST/AVL:

- Serve ordinamento
- Serve ricerca per range
- Serve navigazione ordinata (min, max, successore)

Le tabelle hash, insieme ad array, liste, alberi e grafi, formano il toolkit essenziale delle strutture dati. La scelta appropriata tra queste strutture, basata sulle operazioni richieste e sui vincoli di prestazioni, è una competenza fondamentale per ogni informatico.

Prospettive future:

Esistono molte varianti avanzate di tabelle hash: cuckoo hashing, hopscotch hashing, Robin Hood hashing, ognuna con diversi trade-off tra prestazioni, semplicità e garanzie worst-case. L'esplorazione di queste varianti è un'area di ricerca attiva, particolarmente importante per sistemi concorrenti e distribuiti.



Ogni struttura ha il suo posto: la chiave è scegliere quella giusta!

Fine del capitolo sulle tabelle hash.

Capitolo 7

Algoritmi di Ordinamento

7.1 Introduzione

L'ordinamento è uno dei problemi fondamentali dell'informatica. Dato un insieme di elementi con una relazione d'ordine totale, l'obiettivo è riorganizzare gli elementi in ordine crescente (o decrescente).

7.1.1 Definizione Formale

Dato un array $A[1..n]$ di elementi confrontabili, vogliamo permutare A in modo che:

$$A[1] \leq A[2] \leq \ldots \leq A[n]$$

7.1.2 Classificazione degli Algoritmi

Gli algoritmi di ordinamento si classificano secondo:

- **Complessità temporale**: numero di confronti e scambi
- **Complessità spaziale**: memoria aggiuntiva richiesta
- **Stabilità**: preservazione dell'ordine relativo di elementi uguali
- **In-place**: utilizzo di memoria costante $O(1)$
- **Adattività**: prestazioni migliori su dati parzialmente ordinati

7.2 Bubble Sort

7.2.1 Descrizione

Bubble Sort confronta ripetutamente coppie di elementi adiacenti, scambiandoli se non sono nell'ordine corretto. L'elemento più grande "bolle" verso la fine dell'array ad ogni iterazione.

7.2.2 Pseudocodice

Bubble Sort [1] BubbleSortA, n $i \leftarrow 1$ $n-1$ $j \leftarrow 1$ $n-i$ $A[j] > A[j+1]$ Swap$A[j], A[j+1]$

7.2.3 Versione Ottimizzata

Bubble Sort Ottimizzato [1] BubbleSortOptA, n *swapped* $\leftarrow$ *swapped* *swapped* $\leftarrow$ $j \leftarrow 1$ $n-1$ $A[j] > A[j+1]$ Swap$A[j], A[j+1]$ *swapped* $\leftarrow$ $n \leftarrow n-1$

7.2.4 Analisi di Complessità

Complessità Temporale:

- **Caso peggiore**: $T(n) = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = O(n^2)$
- **Caso medio**: $\Theta(n^2)$ confronti e $\Theta(n^2)$ scambi
- **Caso migliore**: $O(n)$ con ottimizzazione (array già ordinato)

Complessità Spaziale: $O(1)$ - algoritmo in-place

Proprietà:

- Stabile: mantiene l'ordine relativo
- In-place: non richiede memoria aggiuntiva
- Adattivo: termina presto se l'array è già ordinato (versione ottimizzata)

7.2.5 Prova di Correttezza

Invariante del ciclo esterno: Dopo i iterazioni, gli ultimi i elementi sono nella loro posizione finale e sono ordinati.

Base: $i = 0$, nessun elemento in posizione finale (vero banalmente).

Passo induttivo: Assumiamo che dopo i iterazioni gli ultimi i elementi siano ordinati. Durante l'iterazione $i+1$, il ciclo interno porta il massimo tra i primi $n-i$ elementi alla posizione $n-i$, quindi dopo $i+1$ iterazioni gli ultimi $i+1$ elementi sono ordinati.

Terminazione: Dopo $n-1$ iterazioni, gli ultimi $n-1$ elementi sono ordinati, quindi anche il primo è in posizione corretta.

7.2.6 Implementazione Python

```python
def bubble_sort(arr):
    """
    Ordina un array usando Bubble Sort.

    Args:
        arr: lista di elementi confrontabili

    Returns:
        lista ordinata

    Complessita: O(n^2) tempo, O(1) spazio
    """
    n = len(arr)
    arr = arr.copy()  # Non modifichiamo l'originale

    for i in range(n - 1):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

    return arr


def bubble_sort_optimized(arr):
```

```
    """
    Bubble Sort ottimizzato con early stopping.

    Complessita: O(n) best case, O(n^2) worst case
    """
    n = len(arr)
    arr = arr.copy()

    for i in range(n - 1):
        swapped = False
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        if not swapped:
            break  # Array gia ordinato

    return arr
```

7.3 Selection Sort

7.3.1 Descrizione

Selection Sort divide l'array in due parti: ordinata e non ordinata. Ad ogni iterazione, seleziona il minimo dalla parte non ordinata e lo sposta alla fine della parte ordinata.

7.3.2 Pseudocodice

Selection Sort [1] SelectionSortA, n $i \leftarrow 1$ $n-1$ $min\_idx \leftarrow i$ $j \leftarrow i+1$ n $A[j] < A[min\_idx]$ $min\_idx \leftarrow j$ $min\_idx \neq i$ Swap$A[i], A[min_idx]$

7.3.3 Analisi di Complessità

Complessità Temporale:

- **Tutti i casi**: $T(n) = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \Theta(n^2)$
- Numero di confronti: sempre $\frac{n(n-1)}{2}$
- Numero di scambi: al massimo $n-1$ (meglio di Bubble Sort)

Complessità Spaziale: $O(1)$

Proprietà:

- Non stabile (può cambiare ordine relativo)
- In-place
- Non adattivo (sempre $O(n^2)$ confronti)
- Minimo numero di scambi: utile quando gli scambi sono costosi

7.3.4 Prova di Correttezza

Invariante: Dopo i iterazioni, i primi i elementi sono ordinati e sono i più piccoli dell'array.

Base: $i = 0$, nessun elemento ordinato (vero).

Passo: Se i primi i elementi sono i più piccoli e ordinati, l'iterazione $i+1$ trova il minimo tra gli elementi rimanenti e lo posiziona in $i+1$.

Terminazione: Dopo $n-1$ iterazioni, i primi $n-1$ elementi sono ordinati, quindi anche l'ultimo è corretto.

7.3.5 Implementazione Python

```python
def selection_sort(arr):
    """
    Ordina un array usando Selection Sort.

    Complessita: O(n^2) tempo, O(1) spazio
    Caratteristiche: minimo numero di scambi
    """
    n = len(arr)
    arr = arr.copy()

    for i in range(n - 1):
        # Trova il minimo nella parte non ordinata
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j

        # Scambia se necessario
        if min_idx != i:
            arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr
```

7.4 Insertion Sort

7.4.1 Descrizione

Insertion Sort costruisce l'array ordinato un elemento alla volta, inserendo ogni nuovo elemento nella posizione corretta rispetto agli elementi già ordinati.

7.4.2 Pseudocodice

Insertion Sort [1] InsertionSortA, n $i \leftarrow 2$ n $key \leftarrow A[i]$ $j \leftarrow i-1$ $j > 0$ & A[j] > keyA[j+1] $\leftarrow A[j]$ $j \leftarrow j-1$ $A[j+1] \leftarrow key$

7.4.3 Analisi di Complessità

Complessità Temporale:

- **Caso peggiore**: $T(n) = \sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2} = O(n^2)$ (array ordinato al contrario)
- **Caso medio**: $\Theta(n^2)$
- **Caso migliore**: $O(n)$ (array già ordinato)

Complessità Spaziale: $O(1)$

Proprietà:

- Stabile
- In-place
- Adattivo: efficiente su array quasi ordinati
- Online: può ordinare mentre riceve dati
- Efficiente per piccoli array ($n < 50$)

7.4.4 Prova di Correttezza

Invariante: All'inizio dell'iterazione i, il sotto-array $A[1..i-1]$ è ordinato.

Base: $i = 2$, $A[1..1]$ è ordinato (un solo elemento).

Passo: Se $A[1..i-1]$ è ordinato, l'iterazione i inserisce $A[i]$ nella posizione corretta in $A[1..i]$, mantenendo l'ordine.

Terminazione: Dopo n iterazioni, $A[1..n]$ è ordinato.

7.4.5 Implementazione Python

```python
def insertion_sort(arr):
    """
    Ordina un array usando Insertion Sort.

    Complessita: O(n) best case, O(n^2) worst case
    Caratteristiche: stabile, adattivo, efficiente su piccoli array
    """
    n = len(arr)
    arr = arr.copy()

    for i in range(1, n):
        key = arr[i]
        j = i - 1

        # Sposta elementi maggiori di key verso destra
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        # Inserisci key nella posizione corretta
        arr[j + 1] = key

    return arr


def insertion_sort_binary(arr):
    """
    Insertion Sort con ricerca binaria per trovare la posizione.
    Riduce i confronti ma non migliora la complessita asintotica.
    """
    import bisect
    arr = arr.copy()

```

```
    for i in range(1, len(arr)):
        key = arr[i]
        # Trova posizione con ricerca binaria
        pos = bisect.bisect_left(arr, key, 0, i)
        # Sposta e inserisci
        arr = arr[:pos] + [key] + arr[pos:i] + arr[i+1:]

    return arr
```

7.5 Merge Sort

7.5.1 Descrizione

Merge Sort è un algoritmo divide-et-impera che divide ricorsivamente l'array in due metà, le ordina separatamente e poi le fonde.

7.5.2 Pseudocodice

Merge Sort [1] MergeSortA, p, r $p < r$ $q \leftarrow \lfloor (p+r)/2 \rfloor$ MergeSortA, p, q MergeSort$A, q+1, r$ MergeA, p, q, r MergeA, p, q, r $n_1 \leftarrow q - p + 1$ $n_2 \leftarrow r - q$ Crea array $L[1..n_1]$ e $R[1..n_2]$ $i \leftarrow 1$ n_1 $L[i] \leftarrow A[p + i - 1]$ $j \leftarrow 1$ n_2 $R[j] \leftarrow A[q + j]$ $i \leftarrow 1$, $j \leftarrow 1$, $k \leftarrow p$ $i \leq n_1$ & j $\leq n_2$ $L[i] \leq R[j]$ $A[k] \leftarrow L[i]$ $i \leftarrow i + 1$ $A[k] \leftarrow R[j]$ $j \leftarrow j + 1$ $k \leftarrow k + 1$ $i \leq n_1$ $A[k] \leftarrow L[i]$ $i \leftarrow i + 1$, $k \leftarrow k + 1$ $j \leq n_2$ $A[k] \leftarrow R[j]$ $j \leftarrow j + 1$, $k \leftarrow k + 1$

7.5.3 Analisi di Complessità

Ricorrenza:

$$T(n) = 2T(n/2) + \Theta(n)$$

Applicando il Master Theorem (caso 2):

$$T(n) = \Theta(n \log n)$$

Complessità Temporale:

- **Tutti i casi**: $\Theta(n \log n)$ - sempre lo stesso
- Numero di confronti: $\approx n \log n$

Complessità Spaziale: $O(n)$ per gli array temporanei

Proprietà:

- Stabile (se implementato correttamente con $\leq$ nel merge)
- Non in-place (richiede $O(n)$ spazio aggiuntivo)
- Non adattivo (sempre $\Theta(n \log n)$)
- Ottimo per grandi dataset
- Parallelizzabile

7.5.4 Prova di Correttezza

Per induzione sulla dimensione $n = r - p + 1$:

Base: $n = 1$ (p=r), l'array è già ordinato.

Passo: Se MergeSort ordina correttamente array di dimensione $< n$, allora:

- Le due chiamate ricorsive ordinano le due metà
- Merge fonde correttamente due array ordinati in uno ordinato
- Quindi $A[p..r]$ è ordinato

7.5.5 Implementazione Python

```python
def merge_sort(arr):
    """
    Ordina un array usando Merge Sort.

    Complessita: O(n log n) tempo, O(n) spazio
    Caratteristiche: stabile, garantisce O(n log n) anche nel caso
       peggiore
    """
    if len(arr) <= 1:
        return arr

    # Dividi
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    # Fondi
    return merge(left, right)


def merge(left, right):
    """
    Fonde due array ordinati in uno ordinato.
    """
    result = []
    i = j = 0

    # Confronta e fondi
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:  # <= per stabilita
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Aggiungi elementi rimanenti
    result.extend(left[i:])
    result.extend(right[j:])

    return result


def merge_sort_inplace(arr, start=0, end=None):
```

```
    """
    Versione in-place di Merge Sort (usa ancora O(n) spazio nella
       ricorsione).
    """
    if end is None:
        end = len(arr)
        arr = arr.copy()

    if end - start <= 1:
        return arr if end is None else None

    mid = (start + end) // 2
    merge_sort_inplace(arr, start, mid)
    merge_sort_inplace(arr, mid, end)

    # Merge in-place
    temp = []
    i, j = start, mid

    while i < mid and j < end:
        if arr[i] <= arr[j]:
            temp.append(arr[i])
            i += 1
        else:
            temp.append(arr[j])
            j += 1

    temp.extend(arr[i:mid])
    temp.extend(arr[j:end])

    for i, val in enumerate(temp):
        arr[start + i] = val

    return arr if end == len(arr) else None
```

7.6 Quick Sort

7.6.1 Descrizione

Quick Sort sceglie un elemento pivot e partiziona l'array in elementi minori e maggiori del pivot, poi ordina ricorsivamente le due partizioni.

7.6.2 Pseudocodice

Quick Sort [1] QuickSortA, p, r $p < r$ $q \leftarrow$ PartitionA, p, r QuickSort$A, p, q-1$ QuickSort$A, q+1, r$ PartitionA, p, r $x \leftarrow A[r]$ Pivot $i \leftarrow p-1$ $j \leftarrow p$ $r-1$ $A[j] \leq x$ $i \leftarrow i+1$ Swap$A[i], A[j]$ Swap$A[i+1], A[r]$ $i+1$

7.6.3 Analisi di Complessità

Complessità Temporale:

- **Caso peggiore**: $T(n) = T(n-1) + \Theta(n) = O(n^2)$ (pivot sempre min/max)
- **Caso medio**: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$
- **Caso migliore**: $\Theta(n \log n)$ (partizioni bilanciate)

Complessità Spaziale:

- $O(\log n)$ stack ricorsivo nel caso medio
- $O(n)$ nel caso peggiore

Proprietà:

- Non stabile
- In-place (ignoring stack)
- Molto efficiente in pratica (costanti piccole)
- Cache-friendly

7.6.4 Ottimizzazioni

Randomized Quick Sort

Randomized Partition [1] RandomizedPartitionA, p, r $i \leftarrow$ Randomp, r Swap$A[i], A[r]$ PartitionA, p, r

La randomizzazione garantisce complessità attesa $O(n \log n)$ indipendentemente dall'input.

Three-Way Partitioning

Per gestire duplicati efficientemente (Dijkstra's 3-way partition):

Three-Way Partition [1] Partition3WayA, p, r $pivot \leftarrow A[p]$ $lt \leftarrow p$, $gt \leftarrow r$, $i \leftarrow p$ $i \leq gt$ $A[i] < pivot$ Swap$A[lt], A[i]$ $lt \leftarrow lt + 1$, $i \leftarrow i + 1$ $A[i] > pivot$ Swap$A[i], A[gt]$ $gt \leftarrow gt - 1$ $i \leftarrow i + 1$ (lt, gt)

7.6.5 Implementazione Python

```python
def quick_sort(arr):
    """
    Ordina un array usando Quick Sort.

    Complessita: O(n log n) medio, O(n^2) peggiore
    Caratteristiche: in-place, molto efficiente in pratica
    """
    arr = arr.copy()
    _quick_sort_helper(arr, 0, len(arr) - 1)
    return arr


def _quick_sort_helper(arr, low, high):
    """Helper ricorsivo per Quick Sort."""
    if low < high:
        # Partiziona e ottieni indice pivot
        pi = partition(arr, low, high)

        # Ordina ricorsivamente le due partizioni
        _quick_sort_helper(arr, low, pi - 1)
        _quick_sort_helper(arr, pi + 1, high)


def partition(arr, low, high):
```

```
    """
    Partiziona l'array usando l'ultimo elemento come pivot.
    Ritorna l'indice finale del pivot.
    """
    pivot = arr[high]
    i = low - 1  # Indice dell'elemento piu piccolo

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    # Metti il pivot nella posizione corretta
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


def quick_sort_randomized(arr):
    """Quick Sort con pivot randomizzato."""
    import random
    arr = arr.copy()

    def randomized_partition(arr, low, high):
        # Scegli pivot casuale
        pivot_idx = random.randint(low, high)
        arr[pivot_idx], arr[high] = arr[high], arr[pivot_idx]
        return partition(arr, low, high)

    def helper(arr, low, high):
        if low < high:
            pi = randomized_partition(arr, low, high)
            helper(arr, low, pi - 1)
            helper(arr, pi + 1, high)

    helper(arr, 0, len(arr) - 1)
    return arr


def quick_sort_3way(arr):
    """Quick Sort con partizione a 3 vie (per duplicati)."""
    arr = arr.copy()

    def partition_3way(arr, low, high):
        if high <= low:
            return

        pivot = arr[low]
        lt = low      # arr[low..lt-1] < pivot
        gt = high     # arr[gt+1..high] > pivot
        i = low + 1   # arr[lt..i-1] == pivot

        while i <= gt:
            if arr[i] < pivot:
                arr[lt], arr[i] = arr[i], arr[lt]
                lt += 1
                i += 1
            elif arr[i] > pivot:
                arr[i], arr[gt] = arr[gt], arr[i]
```

```
                gt -= 1
            else:
                i += 1

        partition_3way(arr, low, lt - 1)
        partition_3way(arr, gt + 1, high)

    partition_3way(arr, 0, len(arr) - 1)
    return arr
```

7.7 Heap Sort

7.7.1 Descrizione

Heap Sort usa una struttura dati heap (coda di priorità) per ordinare. Costruisce un max-heap e poi estrae ripetutamente il massimo.

7.7.2 Richiami sugli Heap

Un **max-heap** è un albero binario completo dove ogni nodo è $\geq$ dei suoi figli:

$$A[\text{PARENT}(i)] \geq A[i]$$

Operazioni fondamentali:

- $\text{PARENT}(i) = \lfloor i/2 \rfloor$
- $\text{LEFT}(i) = 2i$
- $\text{RIGHT}(i) = 2i + 1$

7.7.3 Pseudocodice

Heap Sort [1] HeapSortA, n BuildMaxHeapA, n $i \leftarrow n$ 2 Swap$A[1], A[i]$ $heap\_size \leftarrow heap\_size - 1$ MaxHeapify$A, 1$ BuildMaxHeapA, n $heap\_size \leftarrow n$ $i \leftarrow \lfloor n/2 \rfloor$ 1 MaxHeapifyA, i MaxHeapifyA, i $l \leftarrow$ LEFT(i) $r \leftarrow$ RIGHT(i) $largest \leftarrow i$ $l \leq heap\_size$ & A[l] $>$ A[largest]largest $\leftarrow l$ $r \leq heap\_size$ & A[r] $>$ A[largest]largest $\leftarrow r$ $largest \neq i$ Swap$A[i], A[largest]$ MaxHeapify$A, largest$

7.7.4 Analisi di Complessità

MaxHeapify: $O(\log n)$ - altezza dell'heap

BuildMaxHeap: Analisi tight: $O(n)$ (non $O(n \log n)$!)

Dimostrazione: In un heap di n elementi, ci sono al più $\lceil n/2^{h+1} \rceil$ nodi ad altezza h:

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil O(h) \\ &= O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} h/2^h \right) \\ &= O(n) \end{aligned}$$

HeapSort: $O(n \log n)$

- BuildMaxHeap: $O(n)$
- $n-1$ estrazioni: $(n-1) \times O(\log n) = O(n \log n)$

Complessità Spaziale: $O(1)$ - in-place

Proprietà:

- Non stabile
- In-place
- $O(n \log n)$ garantito (come Merge Sort)
- Non cache-friendly (accessi sparsi)

7.7.5 Implementazione Python

```python
def heap_sort(arr):
    """
    Ordina un array usando Heap Sort.

    Complessita: O(n log n) tempo, O(1) spazio
    Caratteristiche: in-place, garantisce O(n log n)
    """
    arr = arr.copy()
    n = len(arr)

    # Costruisci max-heap
    build_max_heap(arr, n)

    # Estrai elementi uno alla volta
    for i in range(n - 1, 0, -1):
        # Sposta la radice (max) alla fine
        arr[0], arr[i] = arr[i], arr[0]
        # Ripristina heap sulla parte ridotta
        max_heapify(arr, 0, i)

    return arr


def build_max_heap(arr, n):
    """Costruisce un max-heap dall'array."""
    # Inizia dall'ultimo nodo non-foglia
    for i in range(n // 2 - 1, -1, -1):
        max_heapify(arr, i, n)


def max_heapify(arr, i, heap_size):
    """
    Mantiene la proprieta di max-heap al nodo i.
    Assume che i sottoalberi siano gia max-heap.
    """
    left = 2 * i + 1
    right = 2 * i + 2
    largest = i
```

```
39
40      # Trova il piu grande tra nodo, figlio sx e dx
41      if left < heap_size and arr[left] > arr[largest]:
42          largest = left
43
44      if right < heap_size and arr[right] > arr[largest]:
45          largest = right
46
47      # Se necessario, scambia e ricorsivamente heapify
48      if largest != i:
49          arr[i], arr[largest] = arr[largest], arr[i]
50          max_heapify(arr, largest, heap_size)
51
52
53  # Versione iterativa di max_heapify (evita ricorsione)
54  def max_heapify_iterative(arr, i, heap_size):
55      """Versione iterativa di max_heapify."""
56      while True:
57          left = 2 * i + 1
58          right = 2 * i + 2
59          largest = i
60
61          if left < heap_size and arr[left] > arr[largest]:
62              largest = left
63          if right < heap_size and arr[right] > arr[largest]:
64              largest = right
65
66          if largest == i:
67              break
68
69          arr[i], arr[largest] = arr[largest], arr[i]
70          i = largest
```

7.8 Confronto degli Algoritmi

Tabella 7.1: Confronto algoritmi di ordinamento

Algoritmo	Best	Average	Worst	Spazio	Stabile	In-place
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sì	Sì
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Sì
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sì	Sì
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Sì	No
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No	Sì
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Sì

7.8.1 Quando Usare Quale Algoritmo

- **Insertion Sort**: piccoli array ($n < 50$), array quasi ordinati
- **Merge Sort**: quando serve stabilità e $O(n \log n)$ garantito, linked list
- **Quick Sort**: caso generale, migliori prestazioni medie
- **Heap Sort**: quando serve $O(n \log n)$ garantito con $O(1)$ spazio

- **Bubble/Selection Sort**: solo per scopi didattici o casi molto particolari

7.9 Algoritmi Ibridi

7.9.1 Introsort

Introsort (usato in C++ STL) combina Quick Sort, Heap Sort e Insertion Sort:

- Inizia con Quick Sort
- Se la ricorsione supera $2 \log n$, passa a Heap Sort
- Per array piccoli (< 16 elementi), usa Insertion Sort

Garantisce $O(n \log n)$ nel caso peggiore con ottime prestazioni medie.

7.9.2 Timsort

Timsort (usato in Python e Java) combina Merge Sort e Insertion Sort:

- Identifica "run" naturali ordinati
- Usa Insertion Sort per run piccoli
- Fonde i run con Merge Sort ottimizzato

Eccellente per dati reali (spesso parzialmente ordinati).

7.10 Esercizi

1. Implementare una versione di Quick Sort che usa Insertion Sort per sottoarray $\leq k$ elementi. Trovare sperimentalmente il valore ottimale di k.
2. Dimostrare che ogni algoritmo di ordinamento basato su confronti richiede $\Omega(n \log n)$ confronti nel caso peggiore.
3. Implementare Merge Sort per liste concatenate con complessità spaziale $O(1)$.
4. Analizzare la complessità di Quick Sort quando l'array contiene molti duplicati. Implementare e testare la versione 3-way.
5. Modificare Heap Sort per trovare i k elementi più grandi in $O(n \log k)$.

Capitolo 8

Algoritmi di Ricerca

8.1 Introduzione

La ricerca è un'operazione fondamentale che consiste nel determinare se un elemento appartiene a un insieme e, eventualmente, trovare la sua posizione.

8.1.1 Definizione Formale

Dato un insieme S di n elementi e un elemento chiave k, il problema di ricerca consiste nel:

- Determinare se $k \in S$
- Se $k \in S$, trovare la posizione o un riferimento a k
- Se $k \notin S$, ritornare un valore speciale (es. -1, `None`)

8.1.2 Classificazione

Gli algoritmi di ricerca si classificano in base a:

- **Struttura dei dati**: array ordinati vs non ordinati, liste, alberi
- **Complessità temporale**: lineare, logaritmica, sub-lineare
- **Complessità spaziale**: iterativi vs ricorsivi
- **Tipo di ricerca**: esatta, approssimata, pattern matching

8.2 Ricerca Lineare (Linear Search)

8.2.1 Descrizione

La ricerca lineare (o sequenziale) esamina sequenzialmente ogni elemento dell'array fino a trovare quello cercato o raggiungere la fine.

8.2.2 Pseudocodice

Linear Search [1] LinearSearchA, n, key $i \leftarrow 1$ n $A[i] = key$ i NOT_FOUND

8.2.3 Variante con Sentinella

Per eliminare il controllo del limite ad ogni iterazione:

Linear Search con Sentinella [1] LinearSearchSentinelA, n, key $last \leftarrow A[n]$ $A[n] \leftarrow key$ Sentinella $i \leftarrow 1$ $A[i] \neq key$ $i \leftarrow i + 1$ $A[n] \leftarrow last$ Ripristina $i < n$ $last = key$ i NOT_FOUND

8.2.4 Analisi di Complessità

Complessità Temporale:

- **Caso migliore**: $O(1)$ - elemento in prima posizione
- **Caso peggiore**: $O(n)$ - elemento in ultima posizione o assente
- **Caso medio**: $O(n)$ - assumendo distribuzione uniforme:

$$T_{avg} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2} = \Theta(n)$$

Complessità Spaziale: $O(1)$

Proprietà:

- Funziona su array non ordinati
- Semplice da implementare
- Ottimo per piccoli array o ricerche occasionali
- Non richiede preprocessing

8.2.5 Prova di Correttezza

Invariante: All'inizio dell'iterazione i, $key \notin A[1..i-1]$.
Base: $i = 1$, l'insieme vuoto non contiene key.
Passo: Se $key \notin A[1..i-1]$ e $A[i] \neq key$, allora $key \notin A[1..i]$.
Terminazione:

- Se il ciclo termina con $A[i] = key$, abbiamo trovato la chiave in posizione i
- Se il ciclo termina con $i > n$, allora $key \notin A[1..n]$

8.2.6 Implementazione Python

```python
def linear_search(arr, key):
    """
    Ricerca lineare di un elemento in un array.

    Args:
        arr: array di elementi confrontabili
        key: elemento da cercare

    Returns:
        indice dell'elemento se trovato, -1 altrimenti

    Complessita: O(n) tempo, O(1) spazio
    """
    for i in range(len(arr)):
        if arr[i] == key:
            return i
    return -1


def linear_search_all(arr, key):
```

```
    """
    Trova tutte le occorrenze di key nell'array.

    Returns:
        lista di indici dove key appare
    """
    indices = []
    for i in range(len(arr)):
        if arr[i] == key:
            indices.append(i)
    return indices


def linear_search_sentinel(arr, key):
    """
    Ricerca lineare con sentinella (ottimizzazione).
    Elimina un confronto per iterazione.
    """
    if not arr:
        return -1

    n = len(arr)
    last = arr[-1]

    # Metti sentinella
    arr[-1] = key

    i = 0
    while arr[i] != key:
        i += 1

    # Ripristina ultimo elemento
    arr[-1] = last

    # Verifica se trovato
    if i < n - 1 or last == key:
        return i
    return -1


def linear_search_predicate(arr, predicate):
    """
    Ricerca lineare con predicato personalizzato.

    Args:
        arr: array di elementi
        predicate: funzione che ritorna True per l'elemento cercato

    Returns:
        indice del primo elemento che soddisfa il predicato
    """
    for i, elem in enumerate(arr):
        if predicate(elem):
            return i
    return -1

# Esempio d'uso:
# idx = linear_search_predicate(arr, lambda x: x > 10 and x % 2 == 0)
```

8.3 Ricerca Binaria (Binary Search)

8.3.1 Descrizione

La ricerca binaria sfrutta l'ordinamento dell'array per dimezzare ripetutamente lo spazio di ricerca, confrontando la chiave con l'elemento centrale.

8.3.2 Pseudocodice

Versione Iterativa

Binary Search Iterativo [1] BinarySearch$A, n, key\ left \leftarrow 1\ right \leftarrow n\ left \leq right\ mid \leftarrow \lfloor (left + right)/2 \rfloor\ A[mid] = key\ mid\ A[mid] < key\ left \leftarrow mid + 1\ right \leftarrow mid - 1$ NOT_FOUND

Versione Ricorsiva

Binary Search Ricorsivo [1] BinarySearchRec$A, left, right, key\ left > right$ NOT_FOUND $mid \leftarrow \lfloor (left+right)/2 \rfloor\ A[mid] = key\ mid\ A[mid] < key$ BinarySearchRec$A, mid+1, right, key$ BinarySearchRec$A, left, mid-1, key$

8.3.3 Analisi di Complessità

Ricorrenza:

$$T(n) = T(n/2) + O(1)$$

Applicando il Master Theorem (caso 2):

$$T(n) = \Theta(\log n)$$

Complessità Temporale:

- **Caso migliore**: $O(1)$ - elemento al centro
- **Caso peggiore**: $O(\log n)$ - massimo numero di divisioni
- **Caso medio**: $O(\log n)$

Complessità Spaziale:

- Iterativa: $O(1)$
- Ricorsiva: $O(\log n)$ stack ricorsivo

Numero di confronti: Nel caso peggiore: $\lceil \log_2(n+1) \rceil$

8.3.4 Prova di Correttezza

Invariante: All'inizio di ogni iterazione, se $key \in A$, allora $key \in A[left..right]$.
Base: Inizialmente $left = 1$, $right = n$, quindi $key \in A[1..n]$.
Passo: Se l'invariante è vero e:

- $A[mid] = key$: trovato
- $A[mid] < key$: per l'ordinamento, key può essere solo in $A[mid + 1..right]$
- $A[mid] > key$: per l'ordinamento, key può essere solo in $A[left..mid - 1]$

Terminazione: Il ciclo termina quando $left > right$, cioè l'intervallo è vuoto, quindi $key \notin A$.

8.3.5 Varianti della Ricerca Binaria

Lower Bound

Trova la prima posizione dove inserire key mantenendo l'ordinamento (primo elemento $\geq key$):

Binary Search Lower Bound [1] LowerBoundA, n, key $left \leftarrow 1$, $right \leftarrow n+1$ $left < right$ $mid \leftarrow \lfloor (left + right)/2 \rfloor$ $A[mid] < key$ $left \leftarrow mid + 1$ $right \leftarrow mid$ $left$

Upper Bound

Trova la prima posizione dove un elemento è $> key$:

Binary Search Upper Bound [1] UpperBoundA, n, key $left \leftarrow 1$, $right \leftarrow n+1$ $left < right$ $mid \leftarrow \lfloor (left + right)/2 \rfloor$ $A[mid] \leq key$ $left \leftarrow mid + 1$ $right \leftarrow mid$ $left$

8.3.6 Implementazione Python

```python
def binary_search(arr, key):
    """
    Ricerca binaria iterativa.

    Args:
        arr: array ordinato
        key: elemento da cercare

    Returns:
        indice dell'elemento se trovato, -1 altrimenti

    Complessita: O(log n) tempo, O(1) spazio
    Precondizione: arr deve essere ordinato
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        # Evita overflow: mid = (left + right) // 2
        mid = left + (right - left) // 2

        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            left = mid + 1
        else:
            right = mid - 1

    return -1


def binary_search_recursive(arr, key, left=0, right=None):
    """
    Ricerca binaria ricorsiva.

    Complessita: O(log n) tempo, O(log n) spazio (stack)
    """
    if right is None:
        right = len(arr) - 1

    if left > right:
        return -1

```

```
43      mid = left + (right - left) // 2
44
45      if arr[mid] == key:
46          return mid
47      elif arr[mid] < key:
48          return binary_search_recursive(arr, key, mid + 1, right)
49      else:
50          return binary_search_recursive(arr, key, left, mid - 1)
51
52
53  def binary_search_leftmost(arr, key):
54      """
55      Trova l'indice della prima occorrenza di key.
56      (Lower bound: primo elemento >= key)
57
58      Returns:
59          indice del primo elemento >= key,
60          len(arr) se tutti gli elementi sono < key
61      """
62      left, right = 0, len(arr)
63
64      while left < right:
65          mid = left + (right - left) // 2
66          if arr[mid] < key:
67              left = mid + 1
68          else:
69              right = mid
70
71      return left
72
73
74  def binary_search_rightmost(arr, key):
75      """
76      Trova l'indice dopo l'ultima occorrenza di key.
77      (Upper bound: primo elemento > key)
78
79      Returns:
80          indice del primo elemento > key,
81          len(arr) se tutti gli elementi sono <= key
82      """
83      left, right = 0, len(arr)
84
85      while left < right:
86          mid = left + (right - left) // 2
87          if arr[mid] <= key:
88              left = mid + 1
89          else:
90              right = mid
91
92      return left
93
94
95  def binary_search_range(arr, key):
96      """
97      Trova il range [start, end) di tutte le occorrenze di key.
98
99      Returns:
```

```
        tupla (start, end) dove arr[start:end] contiene tutte le occorrenze
    """
    start = binary_search_leftmost(arr, key)
    end = binary_search_rightmost(arr, key)
    return (start, end)


def binary_search_insert_position(arr, key):
    """
    Trova la posizione dove inserire key per mantenere l'ordinamento.
    Equivalente a lower_bound.
    """
    return binary_search_leftmost(arr, key)


# Uso del modulo bisect (built-in Python)
def binary_search_bisect(arr, key):
    """
    Ricerca binaria usando il modulo bisect di Python.
    """
    import bisect

    # bisect_left: lower bound
    pos = bisect.bisect_left(arr, key)

    if pos < len(arr) and arr[pos] == key:
        return pos
    return -1
```

8.3.7 Applicazioni della Ricerca Binaria

Ricerca in Floating Point

Trovare $\sqrt{x}$ con precisione ϵ:

```
def binary_search_sqrt(x, epsilon=1e-6):
    """
    Calcola la radice quadrata di x usando ricerca binaria.

    Complessita: O(log(x/epsilon))
    """
    if x < 0:
        raise ValueError("x deve essere non negativo")
    if x == 0:
        return 0

    left, right = 0.0, max(1.0, x)

    while right - left > epsilon:
        mid = (left + right) / 2
        square = mid * mid

        if abs(square - x) < epsilon:
            return mid
        elif square < x:
            left = mid
        else:
```

```
            right = mid

    return (left + right) / 2
```

Ricerca della Soluzione

Trovare il minimo k tale che $f(k) \geq target$ (assumendo f monotona):

```
def binary_search_monotonic(f, target, low, high):
    """
    Ricerca binaria su funzione monotona crescente.

    Args:
        f: funzione monotona crescente
        target: valore target
        low, high: range di ricerca

    Returns:
        minimo k in [low, high] tale che f(k) >= target
    """
    result = high + 1

    while low <= high:
        mid = low + (high - low) // 2

        if f(mid) >= target:
            result = mid
            high = mid - 1  # Cerca a sinistra
        else:
            low = mid + 1

    return result if result <= high else None
```

8.4 Ricerca per Interpolazione (Interpolation Search)

8.4.1 Descrizione

L'interpolation search migliora la ricerca binaria usando interpolazione lineare per stimare la posizione della chiave, invece di dividere sempre a metà.

8.4.2 Idea

Se i dati sono uniformemente distribuiti, possiamo stimare la posizione di key con:

$$pos = left + \frac{(key - A[left]) \cdot (right - left)}{A[right] - A[left]}$$

Analogia: cercare "Smith" in un dizionario - iniziamo vicino alla fine, non al centro.

8.4.3 Pseudocodice

Interpolation Search [1] InterpolationSearchA, n, key $left \leftarrow 1$, $right \leftarrow n$ $left \leq right$ & key $\geq A[left]$ & key $\leq A[right]$ $left = right$ $A[left] = key$ $left$ NOT_FOUND $pos \leftarrow left + \lfloor \frac{(key - A[left])(right - left)}{A[right] - A[left]} \rfloor$ $A[pos] = key$ pos $A[pos] < key$ $left \leftarrow pos + 1$ $right \leftarrow pos - 1$ NOT_FOUND

8.4.4 Analisi di Complessità

Complessità Temporale:

- **Caso medio** (dati uniformi): $O(\log \log n)$ - migliore di binary search!
- **Caso peggiore** (dati non uniformi): $O(n)$ - peggiore di binary search

Assunzioni: Per ottenere $O(\log \log n)$, i dati devono essere:

- Ordinati
- Distribuiti uniformemente
- Numerici (o convertibili in numeri)

Complessità Spaziale: $O(1)$

8.4.5 Analisi Dettagliata

Per dati uniformemente distribuiti, ad ogni passo riduciamo l'intervallo a $\sqrt{n}$:

$$T(n) = T(\sqrt{n}) + O(1)$$

Risolvendo:

$$T(n) = O(\log \log n)$$

Dimostrazione: Sia $n = 2^m$, allora $T(2^m) = T(2^{m/2}) + O(1)$. Ponendo $S(m) = T(2^m)$:

$$S(m) = S(m/2) + O(1) = O(\log m) = O(\log \log n)$$

8.4.6 Prova di Correttezza

Simile alla ricerca binaria, l'invariante è:

$$\text{Se } key \in A, \text{ allora } key \in A[left..right]$$

La differenza è nel calcolo di pos, ma la correttezza dell'interpolazione garantisce che:

- Se $A[pos] < key$, allora per monotonia $key \in A[pos + 1..right]$
- Se $A[pos] > key$, allora per monotonia $key \in A[left..pos - 1]$

8.4.7 Implementazione Python

```
def interpolation_search(arr, key):
    """
    Ricerca per interpolazione.

    Args:
        arr: array ordinato di numeri
        key: valore numerico da cercare

    Returns:
        indice dell'elemento se trovato, -1 altrimenti

    Complessita:
        - Caso medio (dati uniformi): O(log log n)
```

```
        - Caso peggiore: O(n)

    Precondizioni:
        - arr ordinato
        - elementi numerici
        - preferibilmente uniformemente distribuiti
    """
    left, right = 0, len(arr) - 1

    while left <= right and key >= arr[left] and key <= arr[right]:
        # Caso speciale: un solo elemento
        if left == right:
            if arr[left] == key:
                return left
            return -1

        # Evita divisione per zero
        if arr[right] == arr[left]:
            if arr[left] == key:
                return left
            return -1

        # Interpolazione lineare
        pos = left + int(
            ((key - arr[left]) * (right - left)) /
            (arr[right] - arr[left])
        )

        # Assicurati che pos sia nell'intervallo
        pos = max(left, min(right, pos))

        if arr[pos] == key:
            return pos
        elif arr[pos] < key:
            left = pos + 1
        else:
            right = pos - 1

    return -1


def interpolation_search_safe(arr, key):
    """
    Versione robusta che gestisce meglio dati non uniformi.
    """
    left, right = 0, len(arr) - 1

    while left <= right and key >= arr[left] and key <= arr[right]:
        if left == right:
            return left if arr[left] == key else -1

        # Evita problemi con distribuzione non uniforme
        range_arr = arr[right] - arr[left]
        if range_arr == 0:
            return left if arr[left] == key else -1

        # Interpolazione
        ratio = (key - arr[left]) / range_arr
```

```
72          pos = left + int(ratio * (right - left))
73
74          # Clamp pos nell'intervallo valido
75          pos = max(left, min(right, pos))
76
77          if arr[pos] == key:
78              return pos
79          elif arr[pos] < key:
80              left = pos + 1
81          else:
82              right = pos - 1
83
84      return -1
```

8.5 Confronto degli Algoritmi di Ricerca

Tabella 8.1: Confronto algoritmi di ricerca

Algoritmo	Ordinamento	Tempo (avg)	Tempo (worst)	Spazio
Linear Search	No	$O(n)$	$O(n)$	$O(1)$
Binary Search	Sì	$O(\log n)$	$O(\log n)$	$O(1)$ iter
Interpolation	Sì + Uniforme	$O(\log \log n)$	$O(n)$	$O(1)$

8.5.1 Quando Usare Quale Algoritmo

- **Linear Search**:
  - Array piccoli ($n < 20$)
  - Array non ordinati
  - Ricerche rare (costo ordinamento non giustificato)
- **Binary Search**:
  - Array ordinati
  - Ricerche frequenti
  - Caso generale più affidabile
  - Garanzia di $O(\log n)$ anche nel caso peggiore
- **Interpolation Search**:
  - Dati numerici uniformemente distribuiti
  - Dataset molto grandi
  - Quando $O(\log \log n)$ fa differenza
  - Non usare con distribuzioni skewed

8.6 Ricerca in Strutture Dati Speciali

8.6.1 Ricerca in Array Ruotato

Un array ordinato ruotato: $[4, 5, 6, 7, 0, 1, 2]$ (originale $[0, 1, 2, 4, 5, 6, 7]$ ruotato di 4 posizioni).

```python
def search_rotated_array(arr, key):
    """
    Ricerca in array ordinato ruotato.

    Complessita: O(log n)
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == key:
            return mid

        # Determina quale meta e ordinata
        if arr[left] <= arr[mid]:  # Meta sinistra ordinata
            if arr[left] <= key < arr[mid]:
                right = mid - 1  # Cerca a sinistra
            else:
                left = mid + 1   # Cerca a destra
        else:  # Meta destra ordinata
            if arr[mid] < key <= arr[right]:
                left = mid + 1   # Cerca a destra
            else:
                right = mid - 1  # Cerca a sinistra

    return -1
```

8.6.2 Ricerca del Picco

Trovare un elemento picco (maggiore dei vicini) in $O(\log n)$:

```python
def find_peak(arr):
    """
    Trova un elemento picco nell'array.
    Un picco e un elemento maggiore dei suoi vicini.

    Complessita: O(log n)
    """
    left, right = 0, len(arr) - 1

    while left < right:
        mid = left + (right - left) // 2

        # Confronta con il vicino destro
        if arr[mid] < arr[mid + 1]:
            # Picco nella meta destra
            left = mid + 1
        else:
            # Picco nella meta sinistra (o mid stesso)
            right = mid

    return left  # left == right, posizione del picco
```

8.6.3 Ricerca in Matrice Ordinata

Matrice con righe e colonne ordinate:

```python
def search_2d_matrix(matrix, key):
    """
    Ricerca in matrice ordinata (ogni riga e colonna ordinata).

    Algoritmo: parti dall'angolo in alto a destra.

    Complessita: O(m + n) dove m=righe, n=colonne
    """
    if not matrix or not matrix[0]:
        return False

    rows, cols = len(matrix), len(matrix[0])
    row, col = 0, cols - 1  # Angolo in alto a destra

    while row < rows and col >= 0:
        if matrix[row][col] == key:
            return True
        elif matrix[row][col] > key:
            col -= 1  # Vai a sinistra
        else:
            row += 1  # Vai in basso

    return False


def search_2d_matrix_binary(matrix, key):
    """
    Ricerca in matrice completamente ordinata
    (primo elemento riga i+1 > ultimo elemento riga i).

    Complessita: O(log(m*n))
    """
    if not matrix or not matrix[0]:
        return False

    rows, cols = len(matrix), len(matrix[0])
    left, right = 0, rows * cols - 1

    while left <= right:
        mid = left + (right - left) // 2
        # Converti indice 1D in 2D
        mid_val = matrix[mid // cols][mid % cols]

        if mid_val == key:
            return True
        elif mid_val < key:
            left = mid + 1
        else:
            right = mid - 1

    return False
```

8.7 Tecniche Avanzate

8.7.1 Exponential Search

Utile quando non conosciamo la dimensione dell'array o quando l'elemento è vicino all'inizio:

```python
def exponential_search(arr, key):
    """
    Exponential search: trova il range, poi usa binary search.

    Complessita: O(log i) dove i e la posizione del key
    Utile quando key e vicino all'inizio.
    """
    n = len(arr)

    # Caso speciale: primo elemento
    if arr[0] == key:
        return 0

    # Trova il range usando raddoppio esponenziale
    i = 1
    while i < n and arr[i] <= key:
        i *= 2

    # Binary search nell'intervallo [i/2, min(i, n-1)]
    return binary_search_range_helper(
        arr, key, i // 2, min(i, n - 1)
    )


def binary_search_range_helper(arr, key, left, right):
    """Helper per binary search in un range."""
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

8.7.2 Fibonacci Search

Divide l'array usando numeri di Fibonacci invece di divisione per 2:

```python
def fibonacci_search(arr, key):
    """
    Fibonacci search: usa numeri di Fibonacci per dividere.

    Vantaggi:
    - Evita divisione (usa solo addizione/sottrazione)
    - Utile su hardware senza divisore veloce

    Complessita: O(log n)
    """
    n = len(arr)

    # Trova i piu piccoli Fibonacci >= n
```

```
    fib_m2 = 0  # (m-2)-esimo Fibonacci
    fib_m1 = 1  # (m-1)-esimo Fibonacci
    fib_m = fib_m2 + fib_m1  # m-esimo Fibonacci

    while fib_m < n:
        fib_m2 = fib_m1
        fib_m1 = fib_m
        fib_m = fib_m2 + fib_m1

    offset = -1

    while fib_m > 1:
        # Controlla se fib_m2 e una posizione valida
        i = min(offset + fib_m2, n - 1)

        if arr[i] < key:
            # Cerca nel sotto-array dopo i
            fib_m = fib_m1
            fib_m1 = fib_m2
            fib_m2 = fib_m - fib_m1
            offset = i
        elif arr[i] > key:
            # Cerca nel sotto-array prima di i
            fib_m = fib_m2
            fib_m1 = fib_m1 - fib_m2
            fib_m2 = fib_m - fib_m1
        else:
            return i

    # Controlla l'ultimo elemento
    if fib_m1 and offset + 1 < n and arr[offset + 1] == key:
        return offset + 1

    return -1
```

8.8 Esercizi

1. Implementare una funzione che trova l'elemento più vicino a k in un array ordinato usando ricerca binaria.
2. Dato un array ordinato con duplicati, implementare funzioni per trovare la prima e l'ultima occorrenza di un elemento in $O(\log n)$.
3. Dimostrare che la ricerca binaria richiede al massimo $\lceil \log_2(n+1) \rceil$ confronti.
4. Implementare una ricerca binaria che trova l'elemento più piccolo in un array ordinato ruotato.
5. Analizzare empiricamente le prestazioni di interpolation search su dati con diverse distribuzioni (uniforme, normale, esponenziale).
6. Implementare ternary search (divide in 3 parti) e confrontare con binary search.
7. Trovare il punto fisso (elemento dove $A[i] = i$) in un array ordinato di interi distinti in $O(\log n)$.
8. Dato un array infinito (o molto grande), implementare una ricerca efficiente senza conoscere la dimensione.

8.9 Note Pratiche

8.9.1 Evitare Overflow

Il classico calcolo di `mid`:

```
mid = (left + right) // 2  # Puo causare overflow!
```

Dovrebbe essere:

```
mid = left + (right - left) // 2  # Sicuro
```

8.9.2 Gestione dei Bounds

Attenzione ai confini degli array:

- Usare $<$ vs $\leq$ correttamente
- Verificare array vuoti
- Controllare limiti prima di accedere

8.9.3 Testing

Casi di test importanti:

- Array vuoto
- Un solo elemento
- Due elementi
- Elemento all'inizio/fine/centro
- Elemento assente
- Tutti elementi uguali

Capitolo 9

Ricorsione

9.1 Introduzione

La ricorsione è una tecnica di programmazione dove una funzione richiama se stessa per risolvere istanze più piccole dello stesso problema.

9.1.1 Definizione

Una funzione f è **ricorsiva** se nella sua definizione compare una chiamata a f stessa:

```
def f(x):
    # ...
    result = f(y)  # Chiamata ricorsiva
    # ...
    return result
```

9.1.2 Componenti Fondamentali

Ogni funzione ricorsiva deve avere:

1. **Caso base** (o terminale): condizione che termina la ricorsione
2. **Caso ricorsivo**: scomposizione del problema in sottoproblemi più semplici
3. **Progresso**: ogni chiamata ricorsiva deve avvicinare al caso base

9.1.3 Principio di Induzione

La correttezza della ricorsione si basa sull'induzione matematica:

- **Base**: il caso base è corretto
- **Passo**: se l'algoritmo è corretto per istanze più piccole, è corretto per l'istanza corrente

9.2 Ricorsione Lineare

9.2.1 Fattoriale

Definizione Matematica

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

Pseudocodice

Fattoriale Ricorsivo [1] Factorial$n$ $n = 0$ 1 Caso base $n\cdot$ Factorial$n - 1$ Caso ricorsivo

Analisi

Ricorrenza:

$$T(n) = T(n-1) + O(1) = O(n)$$

Complessità Spaziale: $O(n)$ stack ricorsivo

Implementazione Python

```python
def factorial(n):
    """
    Calcola il fattoriale di n ricorsivamente.

    Args:
        n: intero non negativo

    Returns:
        n!

    Complessita: O(n) tempo, O(n) spazio (stack)
    """
    if n < 0:
        raise ValueError("n deve essere non negativo")

    # Caso base
    if n == 0 or n == 1:
        return 1

    # Caso ricorsivo
    return n * factorial(n - 1)


def factorial_iterative(n):
    """
    Versione iterativa (piu efficiente).

    Complessita: O(n) tempo, O(1) spazio
    """
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```

9.2.2 Successione di Fibonacci

Definizione

$$F(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F(n-1) + F(n-2) & \text{se } n > 1 \end{cases}$$

Pseudocodice

Fibonacci Ricorsivo [1] Fibonacci$n$ $n \leq 1$ n Fibonacci$n-1$ + Fibonacci$n-2$

Analisi - Versione Naive

Ricorrenza:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Risolvendo (simile a $F(n)$):

$$T(n) = O(\phi^n) \text{ dove } \phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

Problema: crescita esponenziale! Fibonacci(40) richiede $\sim 10^8$ chiamate.

Implementazioni Python

```python
def fibonacci_naive(n):
    """
    Fibonacci ricorsivo naive - INEFFICIENTE!

    Complessita: O(2^n) tempo, O(n) spazio
    """
    if n <= 1:
        return n
    return fibonacci_naive(n - 1) + fibonacci_naive(n - 2)


def fibonacci_memoized(n, memo=None):
    """
    Fibonacci con memoization (Dynamic Programming).

    Complessita: O(n) tempo, O(n) spazio
    """
    if memo is None:
        memo = {}

    if n in memo:
        return memo[n]

    if n <= 1:
        return n

    memo[n] = fibonacci_memoized(n - 1, memo) + \
              fibonacci_memoized(n - 2, memo)
    return memo[n]


def fibonacci_iterative(n):
    """
    Versione iterativa - piu efficiente.

    Complessita: O(n) tempo, O(1) spazio
    """
    if n <= 1:
        return n

```

```
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b

    return b


# Usando decoratore per memoization automatica
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci_cached(n):
    """
    Fibonacci con cache automatica (Python 3.9+).

    Complessita: O(n) tempo, O(n) spazio
    """
    if n <= 1:
        return n
    return fibonacci_cached(n - 1) + fibonacci_cached(n - 2)
```

9.3 Ricorsione con Alberi

9.3.1 Somma degli Elementi

Data una lista (possibilmente annidata), sommare tutti i numeri:

```
def sum_nested(lst):
    """
    Somma ricorsiva di lista annidata.

    Args:
        lst: lista che puo contenere numeri o altre liste

    Returns:
        somma di tutti i numeri

    Esempio:
        sum_nested([1, [2, 3], [[4], 5]]) -> 15
    """
    total = 0

    for element in lst:
        if isinstance(element, list):
            # Caso ricorsivo: lista annidata
            total += sum_nested(element)
        else:
            # Caso base: numero
            total += element

    return total
```

9.3.2 Ricorsione su Alberi Binari

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
```

```
        self.val = val
        self.left = left
        self.right = right


def tree_height(root):
    """
    Calcola l'altezza di un albero binario.

    Complessita: O(n) tempo, O(h) spazio (h = altezza)
    """
    # Caso base: albero vuoto
    if root is None:
        return 0

    # Caso ricorsivo: 1 + max delle altezze dei sottoalberi
    left_height = tree_height(root.left)
    right_height = tree_height(root.right)

    return 1 + max(left_height, right_height)


def tree_sum(root):
    """
    Somma di tutti i nodi dell'albero.
    """
    if root is None:
        return 0
    return root.val + tree_sum(root.left) + tree_sum(root.right)


def tree_count_nodes(root):
    """
    Conta i nodi dell'albero.
    """
    if root is None:
        return 0
    return 1 + tree_count_nodes(root.left) + \
           tree_count_nodes(root.right)


def tree_max(root):
    """
    Trova il valore massimo nell'albero.
    """
    if root is None:
        return float('-inf')

    left_max = tree_max(root.left)
    right_max = tree_max(root.right)

    return max(root.val, left_max, right_max)


def tree_contains(root, target):
    """
    Verifica se target e nell'albero.
    """
```

```
    if root is None:
        return False

    if root.val == target:
        return True

    return tree_contains(root.left, target) or \
           tree_contains(root.right, target)
```

9.4 Tail Recursion

9.4.1 Definizione

Una funzione è **tail-recursive** se la chiamata ricorsiva è l'ultima operazione eseguita (il risultato viene ritornato direttamente senza ulteriori computazioni).

9.4.2 Fattoriale Tail-Recursive

Fattoriale Tail-Recursive [1] FactorialTailn, acc $n = 0$ acc FactorialTail$n - 1, n \cdot acc$ Factorial$n$ FactorialTail$n, 1$

9.4.3 Ottimizzazione: Tail Call Elimination

Compilatori/interpreti che supportano **tail call optimization** (TCO) possono convertire tail recursion in loop, eliminando l'overhead dello stack:

- Complessità spaziale: da $O(n)$ a $O(1)$
- Nessun rischio di stack overflow
- Prestazioni simili a versione iterativa

Nota: Python non implementa TCO per scelta di design.

9.4.4 Implementazioni Python

```
def factorial_tail(n, acc=1):
    """
    Fattoriale tail-recursive.

    Nota: Python non ottimizza tail recursion,
    ma e utile per capire il concetto.
    """
    if n == 0:
        return acc
    return factorial_tail(n - 1, n * acc)


def fibonacci_tail(n, a=0, b=1):
    """
    Fibonacci tail-recursive.
    """
    if n == 0:
        return a
    if n == 1:
        return b
```

```
    return fibonacci_tail(n - 1, b, a + b)


def sum_list_tail(lst, acc=0):
    """
    Somma lista con tail recursion.
    """
    if not lst:
        return acc
    return sum_list_tail(lst[1:], acc + lst[0])


# Conversione tail recursion -> iterazione (manuale)
def factorial_from_tail(n):
    """
    Conversione manuale di tail recursion in loop.
    """
    acc = 1
    while n > 0:
        acc = n * acc
        n = n - 1
    return acc
```

9.4.5 Pattern: Accumulatore

Le funzioni tail-recursive usano spesso un **accumulatore** per mantenere lo stato:

```
def reverse_list_tail(lst, acc=None):
    """
    Inversione lista con tail recursion.
    """
    if acc is None:
        acc = []

    if not lst:
        return acc

    return reverse_list_tail(lst[1:], [lst[0]] + acc)


def gcd_tail(a, b):
    """
    MCD (algoritmo di Euclide) tail-recursive.
    """
    if b == 0:
        return a
    return gcd_tail(b, a % b)
```

9.5 Divide et Impera

9.5.1 Paradigma

Il paradigma **divide et impera** (divide and conquer) consiste in:

1. **Divide**: scomponi il problema in sottoproblemi più piccoli
2. **Impera**: risolvi ricorsivamente i sottoproblemi

3. **Combina**: combina le soluzioni dei sottoproblemi

9.5.2 Merge Sort (Richiamo)

Già visto nel capitolo 7:

```python
def merge_sort(arr):
    """Divide et impera classico."""
    # Caso base
    if len(arr) <= 1:
        return arr

    # Divide
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])    # Impera
    right = merge_sort(arr[mid:])   # Impera

    # Combina
    return merge(left, right)
```

9.5.3 Ricerca Binaria Ricorsiva

```python
def binary_search_recursive(arr, key, left=0, right=None):
    """
    Ricerca binaria con divide et impera.
    """
    if right is None:
        right = len(arr) - 1

    # Caso base: elemento non trovato
    if left > right:
        return -1

    # Divide
    mid = left + (right - left) // 2

    # Caso base: elemento trovato
    if arr[mid] == key:
        return mid

    # Impera su una meta
    if arr[mid] < key:
        return binary_search_recursive(arr, key, mid + 1, right)
    else:
        return binary_search_recursive(arr, key, left, mid - 1)
```

9.5.4 Maximum Subarray Problem

Trovare il sotto-array con somma massima usando divide et impera:

```python
def max_subarray_divide_conquer(arr, low=0, high=None):
    """
    Trova il sotto-array con somma massima.

    Complessita: O(n log n)
    """
    if high is None:
```

```
        high = len(arr) - 1

    # Caso base: un solo elemento
    if low == high:
        return arr[low]

    # Divide
    mid = (low + high) // 2

    # Impera: max nelle tre regioni
    left_max = max_subarray_divide_conquer(arr, low, mid)
    right_max = max_subarray_divide_conquer(arr, mid + 1, high)
    cross_max = max_crossing_subarray(arr, low, mid, high)

    # Combina
    return max(left_max, right_max, cross_max)


def max_crossing_subarray(arr, low, mid, high):
    """
    Trova il max subarray che attraversa mid.
    """
    # Max a sinistra di mid
    left_sum = float('-inf')
    total = 0
    for i in range(mid, low - 1, -1):
        total += arr[i]
        left_sum = max(left_sum, total)

    # Max a destra di mid
    right_sum = float('-inf')
    total = 0
    for i in range(mid + 1, high + 1):
        total += arr[i]
        right_sum = max(right_sum, total)

    return left_sum + right_sum
```

9.6 Backtracking - Introduzione

Il **backtracking** è una tecnica di ricerca esaustiva che costruisce incrementalmente soluzioni candidateabbandonando quelle che non possono portare a soluzioni valide.

9.6.1 Schema Generale

Backtracking Schema [1] Backtrack$solution, candidates$ IsSolution$solution$ ProcessSolution$solution$ $candidate$ $candidates$ IsValid$solution, candidate$ AddToSolution$solution, candidate$ Backtrack$solution, remaini$ RemoveFromSolution$solution, candidate$ Backtrack!

9.6.2 Generazione di Permutazioni

```
def permutations(arr):
    """
    Genera tutte le permutazioni di arr usando backtracking.

```

```
    Complessita: O(n! * n) tempo
    """
    result = []

    def backtrack(current, remaining):
        # Caso base: permutazione completa
        if not remaining:
            result.append(current[:])  # Copia
            return

        # Prova ogni elemento rimanente
        for i in range(len(remaining)):
            # Scegli
            current.append(remaining[i])
            # Esplora
            backtrack(current, remaining[:i] + remaining[i+1:])
            # Annulla (backtrack)
            current.pop()

    backtrack([], arr)
    return result


def permutations_efficient(arr):
    """
    Permutazioni con scambi in-place (piu efficiente).
    """
    result = []

    def backtrack(start):
        if start == len(arr):
            result.append(arr[:])
            return

        for i in range(start, len(arr)):
            # Scambia
            arr[start], arr[i] = arr[i], arr[start]
            # Ricorsione
            backtrack(start + 1)
            # Backtrack (ripristina)
            arr[start], arr[i] = arr[i], arr[start]

    backtrack(0)
    return result
```

9.6.3 Generazione di Sottoinsiemi

```
def subsets(arr):
    """
    Genera tutti i sottoinsiemi di arr.

    Complessita: O(2^n * n) tempo
    """
    result = []

    def backtrack(start, current):
        # Ogni stato e una soluzione valida
```

```
        result.append(current[:])

        # Prova ad aggiungere elementi successivi
        for i in range(start, len(arr)):
            current.append(arr[i])
            backtrack(i + 1, current)
            current.pop()

    backtrack(0, [])
    return result


def subsets_iterative(arr):
    """
    Versione iterativa usando bit manipulation.
    """
    n = len(arr)
    result = []

    # 2^n sottoinsiemi possibili
    for mask in range(1 << n):
        subset = []
        for i in range(n):
            # Controlla se l'i-esimo bit e settato
            if mask & (1 << i):
                subset.append(arr[i])
        result.append(subset)

    return result
```

9.6.4 Combinazioni

Generare tutte le combinazioni di k elementi da n:

```
def combinations(arr, k):
    """
    Genera tutte le combinazioni di k elementi.

    Complessita: O(C(n,k) * k) dove C(n,k) = n!/(k!(n-k)!)
    """
    result = []

    def backtrack(start, current):
        # Caso base: combinazione completa
        if len(current) == k:
            result.append(current[:])
            return

        # Quanti elementi servono ancora
        needed = k - len(current)

        # Prova ogni elemento da start in poi
        for i in range(start, len(arr)):
            # Pruning: controlla se ci sono abbastanza elementi
               rimanenti
            remaining = len(arr) - i
            if remaining < needed:
                break
```

```
            current.append(arr[i])
            backtrack(i + 1, current)
            current.pop()

    backtrack(0, [])
    return result
```

9.7 Ricorsione Multipla

9.7.1 Numero di Cammini in Griglia

Trovare il numero di cammini da $(0, 0)$ a (m, n) muovendosi solo a destra o in basso:

```
def count_paths(m, n):
    """
    Conta i cammini in griglia m x n.

    Soluzione ricorsiva naive: O(2^(m+n))
    """
    # Caso base: bordo della griglia
    if m == 0 or n == 0:
        return 1

    # Ricorsione multipla
    return count_paths(m - 1, n) + count_paths(m, n - 1)


def count_paths_memoized(m, n, memo=None):
    """
    Con memoization: O(m * n)
    """
    if memo is None:
        memo = {}

    if (m, n) in memo:
        return memo[(m, n)]

    if m == 0 or n == 0:
        return 1

    memo[(m, n)] = count_paths_memoized(m - 1, n, memo) + \
                   count_paths_memoized(m, n - 1, memo)

    return memo[(m, n)]


def count_paths_formula(m, n):
    """
    Soluzione matematica: C(m+n, m) = (m+n)! / (m! * n!)

    Complessita: O(m + n)
    """
    from math import comb
    return comb(m + n, m)
```

9.7.2 Torre di Hanoi

Spostare n dischi da palo A a palo C usando palo B come ausiliario:

```
def hanoi(n, source='A', target='C', auxiliary='B'):
    """
    Risolve le Torri di Hanoi.

    Complessita: O(2^n) mosse
    """
    if n == 1:
        print(f"Sposta disco 1 da {source} a {target}")
        return 1

    moves = 0

    # Sposta n-1 dischi da source ad auxiliary
    moves += hanoi(n - 1, source, auxiliary, target)

    # Sposta disco n da source a target
    print(f"Sposta disco {n} da {source} a {target}")
    moves += 1

    # Sposta n-1 dischi da auxiliary a target
    moves += hanoi(n - 1, auxiliary, target, source)

    return moves


def hanoi_iterative(n, source='A', target='C', auxiliary='B'):
    """
    Versione iterativa (meno intuitiva).
    """
    # Numero totale di mosse: 2^n - 1
    total_moves = (1 << n) - 1

    # Se n e pari, scambia target e auxiliary
    if n % 2 == 0:
        target, auxiliary = auxiliary, target

    poles = {source: list(range(n, 0, -1)),
             auxiliary: [],
             target: []}

    for move in range(1, total_moves + 1):
        if move % 3 == 1:
            move_disk(poles, source, target)
        elif move % 3 == 2:
            move_disk(poles, source, auxiliary)
        else:
            move_disk(poles, auxiliary, target)

    return total_moves


def move_disk(poles, from_pole, to_pole):
    """Helper per muovere un disco."""
    if not poles[from_pole]:
        from_pole, to_pole = to_pole, from_pole
```

```
56
57      if not poles[to_pole] or \
58         poles[from_pole][-1] < poles[to_pole][-1]:
59          disk = poles[from_pole].pop()
60          poles[to_pole].append(disk)
61          print(f"Sposta disco {disk} da {from_pole} a {to_pole}")
```

9.8 Tecniche di Ottimizzazione

9.8.1 Memoization

Salvare i risultati di chiamate già computate:

```
1  # Pattern generale per memoization
2  def memoized_function(n, memo=None):
3      if memo is None:
4          memo = {}
5
6      if n in memo:
7          return memo[n]
8
9      # Calcola risultato
10     result = compute(n)
11
12     memo[n] = result
13     return result
14
15
16 # Usando decoratore Python
17 from functools import lru_cache
18
19 @lru_cache(maxsize=None)
20 def cached_function(n):
21     # La cache e automatica
22     return compute(n)
```

9.8.2 Pruning

Tagliare rami dell'albero di ricorsione che non possono portare a soluzioni:

```
1  def subset_sum(arr, target):
2      """
3      Trova un sottoinsieme con somma = target.
4      Con pruning per efficienza.
5      """
6      def backtrack(index, current_sum):
7          # Soluzione trovata
8          if current_sum == target:
9              return True
10
11         # Pruning: somma troppo grande
12         if current_sum > target:
13             return False
14
15         # Pruning: nessun elemento rimanente
16         if index >= len(arr):
17             return False
```

```
18
19          # Include arr[index]
20          if backtrack(index + 1, current_sum + arr[index]):
21              return True
22
23          # Esclude arr[index]
24          if backtrack(index + 1, current_sum):
25              return True
26
27          return False
28
29      return backtrack(0, 0)
```

9.9 Limiti della Ricorsione

9.9.1 Stack Overflow

Python ha un limite di ricorsione (default ∼1000):

```
1  import sys
2
3  # Visualizza limite corrente
4  print(sys.getrecursionlimit())  # Tipicamente 1000
5
6  # Aumenta limite (con cautela!)
7  sys.setrecursionlimit(10000)
8
9  # Meglio: converti in iterazione
10 def safe_deep_recursion(n):
11     """Usa iterazione per evitare stack overflow."""
12     stack = [n]
13     result = 0
14
15     while stack:
16         current = stack.pop()
17         if current == 0:
18             result += 1
19         else:
20             stack.append(current - 1)
21
22     return result
```

9.9.2 Quando Evitare la Ricorsione

Usa iterazione se:

- La profondità è molto grande (> 1000)
- La ricorsione è tail-recursive (converti in loop)
- Non serve la struttura ricorsiva per chiarezza
- Le prestazioni sono critiche

9.9.3 Quando Preferire la Ricorsione

Usa ricorsione se:

- Il problema è naturalmente ricorsivo (alberi, grafi)
- La soluzione iterativa è molto complessa
- Serve backtracking
- La chiarezza del codice è prioritaria

9.10 Esercizi

1. Implementare ricorsivamente:
   (a) Calcolo di x^n
   (b) Inversione di una stringa
   (c) Verifica se una stringa è palindroma
   (d) Somma delle cifre di un numero
2. Convertire in tail-recursive:
   (a) Somma di lista
   (b) Lunghezza di lista
   (c) Conteggio occorrenze in lista
3. Analizzare la complessità di:
   (a) $T(n) = T(n-1) + T(n-2) + O(1)$
   (b) $T(n) = 2T(n/2) + O(n)$
   (c) $T(n) = T(n/3) + T(2n/3) + O(n)$
4. Implementare con backtracking:
   (a) Generazione di tutte le parentesi bilanciate di lunghezza $2n$
   (b) Generazione di tutti i numeri binari di n bit
   (c) Partizione di un insieme in due sottoinsiemi con somma uguale
5. Ottimizzare con memoization:
   (a) Coefficiente binomiale $C(n, k)$
   (b) Problema del resto (coin change)
   (c) Conteggio modi per salire n scalini (1 o 2 per volta)
6. Implementare iterativamente (senza ricorsione):
   (a) Traversal di albero in-order
   (b) Quick Sort
   (c) Torri di Hanoi
7. Dimostrare per induzione la correttezza di:
   (a) Algoritmo di Euclide per MCD

   (b) Ricerca binaria
   (c) Merge Sort

8. Analizzare spazio e tempo per:

   (a) Fibonacci ricorsivo vs iterativo vs con memoization
   (b) Permutazioni con backtracking vs generazione diretta
   (c) Maximum subarray con divide et impera vs Kadane

Capitolo 10

Programmazione Dinamica

10.1 Introduzione

La **Programmazione Dinamica** (Dynamic Programming, DP) è una tecnica di ottimizzazione che risolve problemi complessi scomponendoli in sottoproblemi più semplici e riutilizzando le soluzioni già calcolate.

10.1.1 Caratteristiche Fondamentali

Un problema è adatto alla programmazione dinamica se possiede:

1. **Sottostruttura ottima**: la soluzione ottima contiene soluzioni ottime dei sottoproblemi
2. **Sottoproblemi sovrapposti**: gli stessi sottoproblemi vengono risolti più volte

10.1.2 Differenza con Divide et Impera

Aspetto	Divide et Impera	Programmazione Dinamica
Sottoproblemi	Indipendenti	Sovrapposti
Riuso	No	Sì (memoization)
Approccio	Top-down	Top-down o Bottom-up
Esempi	Merge Sort, Quick Sort	Fibonacci, Knapsack

10.1.3 Approcci

- **Top-down (Memoization)**: ricorsione + cache dei risultati
- **Bottom-up (Tabulation)**: iterazione + tabella per memorizzare soluzioni

10.2 Fibonacci - Caso Studio

10.2.1 Ricorsione Naive

```
def fib_recursive(n):
    """
    Complessita: O(2^n) - MOLTO INEFFICIENTE
    """
    if n <= 1:
        return n
    return fib_recursive(n-1) + fib_recursive(n-2)
```

Problema: calcola $F(n-2)$ due volte: da $F(n)$ e da $F(n-1)$.

10.2.2 Top-Down con Memoization

```python
def fib_memoization(n, memo=None):
    """
    Complessita: O(n) tempo, O(n) spazio
    """
    if memo is None:
        memo = {}

    # Controlla cache
    if n in memo:
        return memo[n]

    # Caso base
    if n <= 1:
        return n

    # Calcola e memorizza
    memo[n] = fib_memoization(n-1, memo) + \
              fib_memoization(n-2, memo)

    return memo[n]
```

10.2.3 Bottom-Up con Tabulation

```python
def fib_tabulation(n):
    """
    Complessita: O(n) tempo, O(n) spazio
    """
    if n <= 1:
        return n

    # Tabella per memorizzare risultati
    dp = [0] * (n + 1)
    dp[1] = 1

    # Riempi la tabella dal basso
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]
```

10.2.4 Ottimizzazione Spaziale

```python
def fib_optimized(n):
    """
    Complessita: O(n) tempo, O(1) spazio
    """
    if n <= 1:
        return n

    prev2, prev1 = 0, 1

```

```
    for _ in range(2, n + 1):
        current = prev1 + prev2
        prev2, prev1 = prev1, current

    return prev1
```

10.3 Problema dello Zaino (Knapsack)

10.3.1 0/1 Knapsack Problem

Input:

- n oggetti, ognuno con peso w_i e valore v_i
- Capacità massima dello zaino: W

Output: Massimo valore ottenibile scegliendo oggetti il cui peso totale $\leq W$, dove ogni oggetto può essere preso al massimo una volta.

10.3.2 Sottostruttura Ottima

Sia $dp[i][w]$ il massimo valore ottenibile usando i primi i oggetti con capacità w:

$$dp[i][w] = \begin{cases} 0 & \text{se } i = 0 \text{ o } w = 0 \\ dp[i-1][w] & \text{se } w_i > w \\ \max(dp[i-1][w],\ v_i + dp[i-1][w-w_i]) & \text{altrimenti} \end{cases}$$

10.3.3 Pseudocodice

0/1 Knapsack - Bottom-Up [1] Knapsack01$values, weights, W, n$ Crea tabella $dp[0..n][0..W]$ $i \leftarrow 0$ n $w \leftarrow 0$ W $i = 0$ $w = 0$ $dp[i][w] \leftarrow 0$ $weights[i-1] \leq w$ $include \leftarrow values[i-1] + dp[i-1][w - weights[i-1]]$ $exclude \leftarrow dp[i-1][w]$ $dp[i][w] \leftarrow \max(include, exclude)$ $dp[i][w] \leftarrow dp[i-1][w]$ $dp[n][W]$

10.3.4 Analisi di Complessità

- **Tempo**: $O(nW)$ - pseudo-polinomiale (dipende da W)
- **Spazio**: $O(nW)$ (ottimizzabile a $O(W)$)

10.3.5 Implementazioni Python

```
def knapsack_01(values, weights, W):
    """
    0/1 Knapsack con programmazione dinamica.

    Args:
        values: lista dei valori
        weights: lista dei pesi
        W: capacita massima

    Returns:
        massimo valore ottenibile

    Complessita: O(nW) tempo, O(nW) spazio
```

```
    """
    n = len(values)

    # Tabella DP
    dp = [[0] * (W + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, W + 1):
            # Opzione 1: non prendere oggetto i-1
            dp[i][w] = dp[i-1][w]

            # Opzione 2: prendere oggetto i-1 (se possibile)
            if weights[i-1] <= w:
                value_with_item = values[i-1] + \
                                  dp[i-1][w - weights[i-1]]
                dp[i][w] = max(dp[i][w], value_with_item)

    return dp[n][W]


def knapsack_01_optimized(values, weights, W):
    """
    Versione ottimizzata per spazio: O(W) invece di O(nW).
    Usa una sola riga della tabella.
    """
    n = len(values)
    dp = [0] * (W + 1)

    for i in range(n):
        # Scorri da destra per evitare di sovrascrivere
        for w in range(W, weights[i] - 1, -1):
            dp[w] = max(dp[w], values[i] + dp[w - weights[i]])

    return dp[W]


def knapsack_01_with_items(values, weights, W):
    """
    Ritorna anche gli oggetti selezionati.
    """
    n = len(values)
    dp = [[0] * (W + 1) for _ in range(n + 1)]

    # Riempi tabella DP
    for i in range(1, n + 1):
        for w in range(1, W + 1):
            dp[i][w] = dp[i-1][w]
            if weights[i-1] <= w:
                value_with = values[i-1] + dp[i-1][w - weights[i-1]]
                dp[i][w] = max(dp[i][w], value_with)

    # Backtrack per trovare oggetti selezionati
    selected = []
    w = W
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i-1][w]:
            selected.append(i - 1)  # Oggetto i-1 selezionato
            w -= weights[i-1]
```

```
72
73      selected.reverse()
74      return dp[n][W], selected
```

10.3.6 Variante: Unbounded Knapsack

Ogni oggetto può essere preso un numero illimitato di volte:

```
1  def knapsack_unbounded(values, weights, W):
2      """
3      Unbounded Knapsack: ogni oggetto disponibile infinite volte.
4
5      Complessita: O(nW)
6      """
7      n = len(values)
8      dp = [0] * (W + 1)
9
10     for w in range(1, W + 1):
11         for i in range(n):
12             if weights[i] <= w:
13                 dp[w] = max(dp[w],
14                             values[i] + dp[w - weights[i]])
15
16     return dp[W]
```

10.4 Longest Common Subsequence (LCS)

10.4.1 Definizione

Date due sequenze $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$, trovare la più lunga sottosequenza comune (non necessariamente contigua).

Esempio:

- X = "ABCDGH"
- Y = "AEDFHR"
- LCS = "ADH" (lunghezza 3)

10.4.2 Sottostruttura Ottima

Sia $c[i][j]$ la lunghezza della LCS di $X[1..i]$ e $Y[1..j]$:

$$c[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ c[i-1][j-1] + 1 & \text{se } x_i = y_j \\ \max(c[i-1][j], c[i][j-1]) & \text{se } x_i \neq y_j \end{cases}$$

10.4.3 Pseudocodice

Longest Common Subsequence [1] LCSX, Y, m, n Crea tabella $c[0..m][0..n]$ $i \leftarrow 0$ m $c[i][0] \leftarrow 0$ $j \leftarrow 0$ n $c[0][j] \leftarrow 0$ $i \leftarrow 1$ m $j \leftarrow 1$ n $X[i] = Y[j]$ $c[i][j] \leftarrow c[i-1][j-1] + 1$ $c[i][j] \leftarrow \max(c[i-1][j], c[i][j-1])$ $c[m][n]$

10.4.4 Analisi di Complessità

- **Tempo**: $O(mn)$
- **Spazio**: $O(mn)$ (ottimizzabile a $O(\min(m, n))$)

10.4.5 Implementazioni Python

```python
def lcs_length(X, Y):
    """
    Calcola la lunghezza della LCS.

    Complessita: O(mn) tempo, O(mn) spazio
    """
    m, n = len(X), len(Y)

    # Tabella DP
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i-1] == Y[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]


def lcs_string(X, Y):
    """
    Ritorna la stringa LCS effettiva.
    """
    m, n = len(X), len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Riempi tabella DP
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i-1] == Y[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    # Backtrack per ricostruire LCS
    lcs = []
    i, j = m, n

    while i > 0 and j > 0:
        if X[i-1] == Y[j-1]:
            lcs.append(X[i-1])
            i -= 1
            j -= 1
        elif dp[i-1][j] > dp[i][j-1]:
            i -= 1
        else:
            j -= 1

```

```
    return ''.join(reversed(lcs))


def lcs_optimized_space(X, Y):
    """
    Ottimizzazione spaziale: O(min(m,n)) spazio.
    """
    # Assicurati che X sia la piu corta
    if len(X) > len(Y):
        X, Y = Y, X

    m, n = len(X), len(Y)

    # Usa due righe invece di tutta la matrice
    prev = [0] * (n + 1)
    curr = [0] * (n + 1)

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i-1] == Y[j-1]:
                curr[j] = prev[j-1] + 1
            else:
                curr[j] = max(prev[j], curr[j-1])
        prev, curr = curr, prev

    return prev[n]
```

10.4.6 Applicazioni di LCS

```
def longest_palindromic_subsequence(s):
    """
    Trova la piu lunga sottosequenza palindroma.
    Idea: LCS(s, reverse(s))
    """
    return lcs_length(s, s[::-1])


def edit_distance_lcs(X, Y):
    """
    Distanza di edit usando LCS.
    edit_distance = m + n - 2*lcs_length
    """
    lcs_len = lcs_length(X, Y)
    return len(X) + len(Y) - 2 * lcs_len


def diff_tool(X, Y):
    """
    Implementazione semplificata di 'diff' Unix.
    Mostra le differenze tra due sequenze.
    """
    m, n = len(X), len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i-1] == Y[j-1]:
```

```
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    # Backtrack e mostra diff
    i, j = m, n
    diff = []

    while i > 0 or j > 0:
        if i > 0 and j > 0 and X[i-1] == Y[j-1]:
            diff.append(f"  {X[i-1]}")
            i -= 1
            j -= 1
        elif j > 0 and (i == 0 or dp[i][j-1] >= dp[i-1][j]):
            diff.append(f"+ {Y[j-1]}")
            j -= 1
        else:
            diff.append(f"- {X[i-1]}")
            i -= 1

    return '\n'.join(reversed(diff))
```

10.5 Edit Distance (Levenshtein Distance)

10.5.1 Definizione

Minimo numero di operazioni (inserimento, cancellazione, sostituzione) per trasformare una stringa in un'altra.

10.5.2 Ricorrenza

Sia $dp[i][j]$ la distanza tra $X[1..i]$ e $Y[1..j]$:

$$dp[i][j] = \begin{cases} i & \text{se } j = 0 \\ j & \text{se } i = 0 \\ dp[i-1][j-1] & \text{se } X[i] = Y[j] \\ 1 + \min \begin{cases} dp[i-1][j] & \text{(cancella)} \\ dp[i][j-1] & \text{(inserisci)} \\ dp[i-1][j-1] & \text{(sostituisci)} \end{cases} & \text{altrimenti} \end{cases}$$

10.5.3 Implementazione Python

```
def edit_distance(X, Y):
    """
    Calcola la distanza di edit (Levenshtein).

    Complessita: O(mn) tempo, O(mn) spazio
    """
    m, n = len(X), len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Inizializzazione: trasformare stringa vuota
    for i in range(m + 1):
        dp[i][0] = i  # i cancellazioni
```

```
    for j in range(n + 1):
        dp[0][j] = j  # j inserimenti

    # Riempi tabella
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i-1] == Y[j-1]:
                dp[i][j] = dp[i-1][j-1]  # Nessuna operazione
            else:
                dp[i][j] = 1 + min(
                    dp[i-1][j],      # Cancella X[i]
                    dp[i][j-1],      # Inserisci Y[j]
                    dp[i-1][j-1]     # Sostituisci X[i] con Y[j]
                )

    return dp[m][n]


def edit_distance_optimized(X, Y):
    """
    Versione ottimizzata per spazio: O(min(m,n)).
    """
    if len(X) > len(Y):
        X, Y = Y, X

    m, n = len(X), len(Y)
    prev = list(range(n + 1))
    curr = [0] * (n + 1)

    for i in range(1, m + 1):
        curr[0] = i
        for j in range(1, n + 1):
            if X[i-1] == Y[j-1]:
                curr[j] = prev[j-1]
            else:
                curr[j] = 1 + min(prev[j], curr[j-1], prev[j-1])
        prev, curr = curr, prev

    return prev[n]


def edit_distance_with_ops(X, Y):
    """
    Ritorna anche le operazioni da eseguire.
    """
    m, n = len(X), len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i-1] == Y[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
```

```
                dp[i][j] = 1 + min(dp[i-1][j],
                                   dp[i][j-1],
                                   dp[i-1][j-1])

    # Backtrack per operazioni
    operations = []
    i, j = m, n

    while i > 0 or j > 0:
        if i == 0:
            operations.append(f"Insert '{Y[j-1]}'")
            j -= 1
        elif j == 0:
            operations.append(f"Delete '{X[i-1]}'")
            i -= 1
        elif X[i-1] == Y[j-1]:
            i -= 1
            j -= 1
        else:
            # Trova quale operazione e stata fatta
            if dp[i][j] == dp[i-1][j-1] + 1:
                operations.append(
                    f"Replace '{X[i-1]}' with '{Y[j-1]}'"
                )
                i -= 1
                j -= 1
            elif dp[i][j] == dp[i-1][j] + 1:
                operations.append(f"Delete '{X[i-1]}'")
                i -= 1
            else:
                operations.append(f"Insert '{Y[j-1]}'")
                j -= 1

    operations.reverse()
    return dp[m][n], operations
```

10.6 Longest Increasing Subsequence (LIS)

10.6.1 Definizione

Trovare la più lunga sottosequenza strettamente crescente in un array.

10.6.2 Soluzione DP - $O(n^2)$

```
def lis_dp(arr):
    """
    LIS con programmazione dinamica.

    Complessita: O(n^2) tempo, O(n) spazio
    """
    if not arr:
        return 0

    n = len(arr)
    # dp[i] = lunghezza LIS che termina in arr[i]
    dp = [1] * n
```

```
13
14      for i in range(1, n):
15          for j in range(i):
16              if arr[j] < arr[i]:
17                  dp[i] = max(dp[i], dp[j] + 1)
18
19      return max(dp)
20
21
22  def lis_with_sequence(arr):
23      """
24      Ritorna anche la sequenza LIS.
25      """
26      if not arr:
27          return 0, []
28
29      n = len(arr)
30      dp = [1] * n
31      parent = [-1] * n  # Per ricostruire la sequenza
32
33      for i in range(1, n):
34          for j in range(i):
35              if arr[j] < arr[i] and dp[j] + 1 > dp[i]:
36                  dp[i] = dp[j] + 1
37                  parent[i] = j
38
39      # Trova il massimo
40      max_length = max(dp)
41      max_idx = dp.index(max_length)
42
43      # Ricostruisci sequenza
44      lis = []
45      idx = max_idx
46      while idx != -1:
47          lis.append(arr[idx])
48          idx = parent[idx]
49
50      lis.reverse()
51      return max_length, lis
```

10.6.3 Soluzione Ottimizzata - $O(n \log n)$

Usa ricerca binaria per mantenere una sequenza "tails":

```
1   def lis_binary_search(arr):
2       """
3       LIS ottimizzato con binary search.
4
5       Complessita: O(n log n) tempo, O(n) spazio
6       """
7       if not arr:
8           return 0
9
10      # tails[i] = piu piccolo elemento che termina una LIS di lunghezza i
            +1
11      tails = []
12
13      for num in arr:
```

```
        # Trova posizione per num in tails
        left, right = 0, len(tails)

        while left < right:
            mid = (left + right) // 2
            if tails[mid] < num:
                left = mid + 1
            else:
                right = mid

        # Se left == len(tails), estendi
        if left == len(tails):
            tails.append(num)
        else:
            tails[left] = num

    return len(tails)


# Usando bisect (built-in Python)
import bisect

def lis_bisect(arr):
    """
    LIS usando bisect di Python.
    """
    tails = []

    for num in arr:
        pos = bisect.bisect_left(tails, num)
        if pos == len(tails):
            tails.append(num)
        else:
            tails[pos] = num

    return len(tails)
```

10.7 Coin Change Problem

10.7.1 Numero Minimo di Monete

Dato un insieme di denominazioni e un ammontare, trovare il minimo numero di monete per ottenere quell'ammontare.

```
def coin_change_min(coins, amount):
    """
    Minimo numero di monete per raggiungere amount.

    Complessita: O(amount * n) dove n = len(coins)
    """
    # dp[i] = min monete per ammontare i
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0  # 0 monete per ammontare 0

    for i in range(1, amount + 1):
        for coin in coins:
            if coin <= i:
```

```
                dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1


def coin_change_with_coins(coins, amount):
    """
    Ritorna anche le monete usate.
    """
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0
    parent = [-1] * (amount + 1)

    for i in range(1, amount + 1):
        for coin in coins:
            if coin <= i and dp[i - coin] + 1 < dp[i]:
                dp[i] = dp[i - coin] + 1
                parent[i] = coin

    if dp[amount] == float('inf'):
        return -1, []

    # Ricostruisci monete
    result = []
    curr = amount
    while curr > 0:
        coin = parent[curr]
        result.append(coin)
        curr -= coin

    return dp[amount], result
```

10.7.2 Numero di Modi

Contare quanti modi ci sono per ottenere un ammontare:

```
def coin_change_ways(coins, amount):
    """
    Conta il numero di modi per ottenere amount.

    Complessita: O(amount * n)
    """
    dp = [0] * (amount + 1)
    dp[0] = 1  # Un modo per ottenere 0

    # Per ogni moneta
    for coin in coins:
        # Aggiorna tutti gli ammontari >= coin
        for i in range(coin, amount + 1):
            dp[i] += dp[i - coin]

    return dp[amount]
```

10.8 Matrix Chain Multiplication

10.8.1 Problema

Date n matrici $A_1, A_2, \ldots, A_n$ con dimensioni compatibili, trovare il modo di parentesizzare il prodotto che minimizza il numero di moltiplicazioni scalari.

10.8.2 Ricorrenza

Sia $m[i][j]$ il minimo numero di moltiplicazioni per calcolare $A_i \cdots A_j$:

$$m[i][j] = \begin{cases} 0 & \text{se } i = j \\ \min_{i \le k < j}\{m[i][k] + m[k+1][j] + p_{i-1} \cdot p_k \cdot p_j\} & \text{se } i < j \end{cases}$$

dove p_i rappresenta le dimensioni delle matrici.

10.8.3 Implementazione Python

```python
def matrix_chain_order(dimensions):
    """
    Trova l'ordine ottimale per moltiplicare matrici.

    Args:
        dimensions: lista [p0, p1, ..., pn]
                    dove matrice i ha dimensioni pi-1 x pi

    Returns:
        minimo numero di moltiplicazioni

    Complessita: O(n^3) tempo, O(n^2) spazio
    """
    n = len(dimensions) - 1  # Numero di matrici

    # m[i][j] = costo minimo per A[i..j]
    m = [[0] * n for _ in range(n)]

    # l = lunghezza della catena
    for l in range(2, n + 1):
        for i in range(n - l + 1):
            j = i + l - 1
            m[i][j] = float('inf')

            for k in range(i, j):
                # Costo = costo sinistra + costo destra + costo merge
                cost = m[i][k] + m[k+1][j] + \
                       dimensions[i] * dimensions[k+1] * dimensions[j+1]
                m[i][j] = min(m[i][j], cost)

    return m[0][n-1]


def matrix_chain_with_parenthesis(dimensions):
    """
    Ritorna anche la parentesizzazione ottimale.
    """
    n = len(dimensions) - 1
    m = [[0] * n for _ in range(n)]
```

```
    s = [[0] * n for _ in range(n)]  # Split point

    for l in range(2, n + 1):
        for i in range(n - l + 1):
            j = i + l - 1
            m[i][j] = float('inf')

            for k in range(i, j):
                cost = m[i][k] + m[k+1][j] + \
                       dimensions[i] * dimensions[k+1] * dimensions[j+1]
                if cost < m[i][j]:
                    m[i][j] = cost
                    s[i][j] = k

    def print_optimal(i, j):
        if i == j:
            return f"A{i}"
        return f"({print_optimal(i, s[i][j])} * " \
               f"{print_optimal(s[i][j]+1, j)})"

    return m[0][n-1], print_optimal(0, n-1)
```

10.9 Esercizi

1. Implementare e analizzare:
   (a) Subset Sum Problem (esiste un sottoinsieme con somma = target?)
   (b) Partition Problem (dividere in due sottoinsiemi con somma uguale)
   (c) Rod Cutting (tagliare un'asta per massimizzare il profitto)
2. Problemi su stringhe:
   (a) Longest Palindromic Substring
   (b) Wildcard Pattern Matching
   (c) Regular Expression Matching (semplificato)
3. Problemi su griglie:
   (a) Unique Paths (contare cammini in griglia)
   (b) Minimum Path Sum (cammino con costo minimo)
   (c) Maximum Path Sum in Triangle
4. Ottimizzazioni:
   (a) Ottimizzare 0/1 Knapsack a $O(W)$ spazio
   (b) Implementare LCS con $O(\min(m, n))$ spazio
   (c) Analizzare quando conviene top-down vs bottom-up
5. Problemi avanzati:
   (a) Longest Common Substring (contigua)
   (b) Box Stacking (massimizzare altezza stack di scatole)
   (c) Optimal Binary Search Tree

Capitolo 11

Algoritmi Greedy

11.1 Introduzione

Un **algoritmo greedy** (avido o goloso) costruisce una soluzione passo dopo passo, scegliendo ad ogni passo l'opzione localmente ottimale, sperando di arrivare a una soluzione globalmente ottimale.

11.1.1 Caratteristiche

- **Scelta greedy**: ad ogni passo, sceglie l'opzione che sembra migliore al momento
- **Proprietà greedy**: una scelta localmente ottimale porta a una soluzione globalmente ottimale
- **Sottostruttura ottima**: la soluzione ottima contiene soluzioni ottime dei sottoproblemi
- **Irrevocabilità**: le scelte non vengono mai riviste

11.1.2 Greedy vs Dynamic Programming

Aspetto	**Greedy**	**Dynamic Programming**
Scelta	Locale, irrevocabile	Considera tutte le opzioni
Complessità	Generalmente più bassa	Generalmente più alta
Correttezza	Non sempre garantita	Garantita (se applicabile)
Applicabilità	Problemi specifici	Classe più ampia

11.1.3 Quando Usare Greedy

Un algoritmo greedy funziona se il problema ha:

1. **Sottostruttura ottima**
2. **Proprietà greedy choice**: la scelta localmente ottima è sempre corretta

11.2 Activity Selection Problem

11.2.1 Definizione

Dato un insieme di n attività con tempo di inizio s_i e tempo di fine f_i, selezionare il massimo numero di attività mutualmente compatibili (non sovrapposte).

11.2.2 Strategia Greedy

Scelta greedy: Seleziona sempre l'attività con tempo di fine più precoce tra quelle compatibili.

Intuizione: Terminando prima, lasciamo più tempo per altre attività.

11.2.3 Pseudocodice

Activity Selection [1] ActivitySelections, f, n Ordina attività per tempo di fine crescente $A \leftarrow \{a_1\}$ Prima attività $last\_finish \leftarrow f_1$ $i \leftarrow 2$ n $s_i \geq last\_finish$ $A \leftarrow A \cup \{a_i\}$ $last\_finish \leftarrow f_i$ A

11.2.4 Prova di Correttezza

Teorema: La strategia greedy produce una soluzione ottima.

Dimostrazione (per scambio): Sia $A = \{a_1, a_2, \ldots, a_k\}$ la soluzione greedy e $O = \{o_1, o_2, \ldots, o_m\}$ una soluzione ottima (ordinate per tempo di fine).

Vogliamo mostrare che $k = m$ (stesso numero di attività).

Per induzione, possiamo trasformare O in A scambiando attività senza ridurre la dimensione:

- Se $a_1 = o_1$, applica induzione al resto
- Se $a_1 \neq o_1$, $f(a_1) \leq f(o_1)$ per scelta greedy
- Sostituisci o_1 con a_1 in O: rimane ottima
- Continua per induzione

11.2.5 Analisi di Complessità

- **Ordinamento**: $O(n \log n)$
- **Selezione**: $O(n)$
- **Totale**: $O(n \log n)$

11.2.6 Implementazione Python

```python
def activity_selection(activities):
    """
    Seleziona il massimo numero di attivita compatibili.

    Args:
        activities: lista di tuple (start, finish)

    Returns:
        lista di attivita selezionate

    Complessita: O(n log n)
    """
    if not activities:
        return []

    # Ordina per tempo di fine
    sorted_activities = sorted(activities, key=lambda x: x[1])

    selected = [sorted_activities[0]]
    last_finish = sorted_activities[0][1]

    for start, finish in sorted_activities[1:]:
```

```
        # Se compatibile, seleziona
        if start >= last_finish:
            selected.append((start, finish))
            last_finish = finish

    return selected


def activity_selection_weighted(activities, weights):
    """
    Variante pesata: massimizza il peso totale.
    Nota: richiede Dynamic Programming, non greedy!

    Questo e un esempio dove greedy NON funziona.
    """
    # Ordina per tempo di fine
    n = len(activities)
    sorted_indices = sorted(range(n),
                            key=lambda i: activities[i][1])

    # DP: dp[i] = max peso usando attivita 0..i
    dp = [0] * n
    dp[0] = weights[sorted_indices[0]]

    for i in range(1, n):
        # Peso includendo attivita i
        weight_with = weights[sorted_indices[i]]

        # Trova ultima attivita compatibile
        for j in range(i - 1, -1, -1):
            if activities[sorted_indices[j]][1] <= \
               activities[sorted_indices[i]][0]:
                weight_with += dp[j]
                break

        # Peso escludendo attivita i
        weight_without = dp[i - 1]

        dp[i] = max(weight_with, weight_without)

    return dp[n - 1]
```

11.3 Fractional Knapsack

11.3.1 Definizione

Simile a 0/1 Knapsack, ma possiamo prendere frazioni di oggetti.

Input:

- n oggetti con peso w_i e valore v_i
- Capacità W

Output: Massimo valore prendendo frazioni di oggetti con peso totale $\leq W$.

11.3.2 Strategia Greedy

Scelta greedy: Ordina per rapporto valore/peso decrescente, prendi quanto possibile di ogni oggetto.

11.3.3 Pseudocodice

Fractional Knapsack [1] FractionalKnapsack$values, weights, W$ Calcola $ratios[i] \leftarrow values[i]/weights[i]$ Ordina oggetti per $ratios$ decrescente $total\_value \leftarrow 0$ $remaining\_capacity \leftarrow W$ ogni oggetto i in ordine $weights[i] \leq remaining\_capacity$ Prendi tutto l'oggetto i $total\_value \leftarrow total\_value + values[i]$ $remaining\_capacity \leftarrow remaining\_capacity - weights[i]$ Prendi frazione $remaining\_capacity/weights[i]$ di i $total\_value \leftarrow total\_value + values[i] \cdot (remaining\_capacity/weights[i])$ $total\_value$ $total\_value$

11.3.4 Prova di Correttezza

La scelta greedy funziona perché:

- Prendere l'oggetto con massimo valore/peso è sempre ottimale
- Se una soluzione ottima prende meno di questo oggetto, possiamo scambiare parte di un altro oggetto (con rapporto inferiore) con più di questo oggetto, migliorando la soluzione

11.3.5 Implementazione Python

```python
def fractional_knapsack(values, weights, capacity):
    """
    Fractional Knapsack con approccio greedy.

    Complessita: O(n log n)
    """
    n = len(values)

    # Crea lista di (valore, peso, indice, rapporto)
    items = []
    for i in range(n):
        ratio = values[i] / weights[i]
        items.append((values[i], weights[i], i, ratio))

    # Ordina per rapporto decrescente
    items.sort(key=lambda x: x[3], reverse=True)

    total_value = 0
    remaining_capacity = capacity
    fractions = [0.0] * n  # Frazione di ogni oggetto presa

    for value, weight, idx, ratio in items:
        if remaining_capacity == 0:
            break

        if weight <= remaining_capacity:
            # Prendi tutto
            fractions[idx] = 1.0
            total_value += value
            remaining_capacity -= weight
        else:
            # Prendi frazione
```

```
            fraction = remaining_capacity / weight
            fractions[idx] = fraction
            total_value += value * fraction
            remaining_capacity = 0

    return total_value, fractions


# Esempio d'uso
values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50

max_value, fractions = fractional_knapsack(values, weights, capacity)
print(f"Valore massimo: {max_value}")
print(f"Frazioni prese: {fractions}")
# Output: Valore massimo: 240.0
# Frazioni: [1.0, 1.0, 0.6666...]
```

11.4 Huffman Coding

11.4.1 Problema

Costruire un codice a lunghezza variabile ottimale per comprimere dati.

11.4.2 Idea

Assegna codici più corti ai caratteri più frequenti.

11.4.3 Strategia Greedy

1. Crea un nodo foglia per ogni carattere con la sua frequenza
2. Ripetutamente:
   - Trova i due nodi con frequenza minima
   - Crea un nodo padre con frequenza = somma dei figli
   - Rimuovi i due nodi, aggiungi il padre
3. Continua finché rimane un solo nodo (radice)

11.4.4 Pseudocodice

Huffman Coding [1] Huffman$C$ C = caratteri con frequenze $n \leftarrow |C|$ $Q \leftarrow C$ Min-priority queue $i \leftarrow 1$ $n-1$ $z \leftarrow$ nuovo nodo $z.left \leftarrow$ ExtractMin$Q$ $z.right \leftarrow$ ExtractMin$Q$ $z.freq \leftarrow z.left.freq + z.right.freq$ InsertQ, z ExtractMin$Q$ Radice dell'albero

11.4.5 Analisi di Complessità

Con heap binario:

- Costruzione heap: $O(n)$
- $n-1$ iterazioni, ognuna con 2 extract-min e 1 insert: $O(n \log n)$
- **Totale**: $O(n \log n)$

11.4.6 Implementazione Python

```python
import heapq
from collections import defaultdict, Counter


class HuffmanNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq


def huffman_encoding(text):
    """
    Costruisce albero di Huffman e codifica testo.

    Complessita: O(n log n) dove n = numero caratteri unici
    """
    if not text:
        return "", None, {}

    # Calcola frequenze
    freq = Counter(text)

    # Caso speciale: un solo carattere
    if len(freq) == 1:
        char = list(freq.keys())[0]
        codes = {char: '0'}
        encoded = '0' * len(text)
        return encoded, None, codes

    # Crea heap con nodi foglia
    heap = [HuffmanNode(char, f) for char, f in freq.items()]
    heapq.heapify(heap)

    # Costruisci albero
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)

        parent = HuffmanNode(None, left.freq + right.freq)
        parent.left = left
        parent.right = right

        heapq.heappush(heap, parent)

    root = heap[0]

    # Genera codici
    codes = {}

    def generate_codes(node, code=""):
        if node is None:
```

```
57              return
58
59          if node.char is not None:  # Foglia
60              codes[node.char] = code if code else "0"
61              return
62
63          generate_codes(node.left, code + "0")
64          generate_codes(node.right, code + "1")
65
66      generate_codes(root)
67
68      # Codifica testo
69      encoded = ''.join(codes[char] for char in text)
70
71      return encoded, root, codes
72
73
74  def huffman_decoding(encoded, root):
75      """
76      Decodifica testo usando albero di Huffman.
77
78      Complessita: O(m) dove m = lunghezza testo codificato
79      """
80      if not encoded or not root:
81          return ""
82
83      # Caso speciale: un solo carattere
84      if root.char is not None:
85          return root.char * len(encoded)
86
87      decoded = []
88      current = root
89
90      for bit in encoded:
91          if bit == '0':
92              current = current.left
93          else:
94              current = current.right
95
96          # Raggiunto foglia
97          if current.char is not None:
98              decoded.append(current.char)
99              current = root
100
101     return ''.join(decoded)
102
103
104 # Esempio d'uso
105 text = "this is an example for huffman encoding"
106 encoded, tree, codes = huffman_encoding(text)
107
108 print("Codici Huffman:")
109 for char, code in sorted(codes.items()):
110     print(f"  '{char}': {code}")
111
112 print(f"\nTesto originale: {len(text) * 8} bits (ASCII)")
113 print(f"Testo codificato: {len(encoded)} bits")
114 print(f"Compressione: {100 * (1 - len(encoded)/(len(text)*8)):.1f}%")
```

```
115
116 decoded = huffman_decoding(encoded, tree)
117 assert decoded == text
118 print(f"\nDecodifica corretta: {decoded == text}")
```

11.5 Minimum Spanning Tree (MST)

11.5.1 Definizione

Dato un grafo non orientato connesso e pesato $G = (V, E)$, trovare un albero che:

- Connette tutti i vertici
- Ha peso totale minimo

11.5.2 Algoritmo di Kruskal

Strategia

Ordina gli archi per peso crescente e aggiungi archi che non creano cicli.

Pseudocodice

Kruskal's MST [1] Kruskal$G = (V, E)$ $A \leftarrow \emptyset$ ogni vertice $v \in V$ MakeSet$v$ Ordina E per peso crescente ogni arco $(u, v) \in E$ in ordine FindSet$u \neq$ FindSet$v$ $A \leftarrow A \cup \{(u, v)\}$ Unionu, v A

Implementazione Python

```python
class UnionFind:
    """
    Union-Find (Disjoint Set Union) con path compression
    e union by rank.
    """
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        """Trova il rappresentante con path compression."""
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        """Unisci due insiemi con union by rank."""
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x == root_y:
            return False

        # Union by rank
        if self.rank[root_x] < self.rank[root_y]:
            self.parent[root_x] = root_y
        elif self.rank[root_x] > self.rank[root_y]:
            self.parent[root_y] = root_x
        else:
```

```
            self.parent[root_y] = root_x
            self.rank[root_x] += 1

        return True


def kruskal_mst(n, edges):
    """
    Algoritmo di Kruskal per MST.

    Args:
        n: numero di vertici (0..n-1)
        edges: lista di (weight, u, v)

    Returns:
        (peso_totale, lista_archi_mst)

    Complessita: O(E log E) = O(E log V)
    """
    # Ordina archi per peso
    edges.sort()

    uf = UnionFind(n)
    mst = []
    total_weight = 0

    for weight, u, v in edges:
        # Se non crea ciclo, aggiungi
        if uf.union(u, v):
            mst.append((u, v, weight))
            total_weight += weight

            # MST completo quando ha n-1 archi
            if len(mst) == n - 1:
                break

    return total_weight, mst


# Esempio
edges = [
    (1, 0, 1),  # (peso, u, v)
    (2, 0, 2),
    (3, 1, 2),
    (4, 1, 3),
    (5, 2, 3)
]

weight, mst = kruskal_mst(4, edges)
print(f"Peso MST: {weight}")
print(f"Archi MST: {mst}")
```

11.5.3 Algoritmo di Prim

Strategia

Parti da un vertice, espandi l'albero aggiungendo sempre l'arco di peso minimo che connette un nuovo vertice.

Pseudocodice

Prim's MST [1] PrimG, w, r r = radice ogni vertice $u \in V$ $key[u] \leftarrow \infty$ $parent[u] \leftarrow$ NIL $key[r] \leftarrow 0$ $Q \leftarrow V$ Min-priority queue $Q \neq \emptyset$ $u \leftarrow$ Extract$\text{Min}Q$ ogni $v \in Adj[u]$ $v \in Q$ & w(u,v) $<$ key[v]parent[v] $\leftarrow u$ $key[v] \leftarrow w(u, v)$

Implementazione Python

```python
import heapq


def prim_mst(n, graph, start=0):
    """
    Algoritmo di Prim per MST.

    Args:
        n: numero di vertici
        graph: dict {u: [(v, weight), ...]}
        start: vertice di partenza

    Returns:
        (peso_totale, lista_archi_mst)

    Complessita: O(E log V) con binary heap
    """
    visited = [False] * n
    mst = []
    total_weight = 0

    # Min-heap: (peso, vertice_corrente, vertice_precedente)
    heap = [(0, start, -1)]

    while heap:
        weight, u, parent = heapq.heappop(heap)

        if visited[u]:
            continue

        visited[u] = True
        total_weight += weight

        if parent != -1:
            mst.append((parent, u, weight))

        # Aggiungi archi adiacenti
        for v, edge_weight in graph.get(u, []):
            if not visited[v]:
                heapq.heappush(heap, (edge_weight, v, u))

    return total_weight, mst


# Esempio
graph = {
    0: [(1, 1), (2, 2)],
    1: [(0, 1), (2, 3), (3, 4)],
    2: [(0, 2), (1, 3), (3, 5)],
    3: [(1, 4), (2, 5)]
```

```
}

weight, mst = prim_mst(4, graph)
print(f"Peso MST: {weight}")
print(f"Archi MST: {mst}")
```

11.6 Shortest Path - Dijkstra

11.6.1 Problema

Trovare il cammino più breve da una sorgente a tutti gli altri vertici in un grafo con pesi non negativi.

11.6.2 Strategia Greedy

Mantieni un insieme S di vertici con distanza minima già calcolata. Ad ogni passo, aggiungi a S il vertice $u \notin S$ con distanza minima.

11.6.3 Pseudocodice

Dijkstra's Algorithm [1] DijkstraG, w, s ogni vertice $v \in V$ $dist[v] \leftarrow \infty$ $parent[v] \leftarrow$ NIL $dist[s] \leftarrow 0$ $Q \leftarrow V$ $Q \neq \emptyset$ $u \leftarrow$ vertice in Q con $dist[u]$ minima Rimuovi u da Q ogni vicino v di u $alt \leftarrow dist[u] + w(u, v)$ $alt < dist[v]$ $dist[v] \leftarrow alt$ $parent[v] \leftarrow u$

11.6.4 Implementazione Python

```
import heapq


def dijkstra(graph, start, n):
    """
    Algoritmo di Dijkstra per shortest paths.

    Args:
        graph: dict {u: [(v, weight), ...]}
        start: vertice sorgente
        n: numero di vertici

    Returns:
        (distanze, predecessori)

    Complessita: O((V + E) log V) con binary heap
    """
    dist = [float('inf')] * n
    parent = [-1] * n
    dist[start] = 0

    # Min-heap: (distanza, vertice)
    heap = [(0, start)]
    visited = set()

    while heap:
        d, u = heapq.heappop(heap)

        if u in visited:
```

```
            continue

        visited.add(u)

        # Rilassamento degli archi
        for v, weight in graph.get(u, []):
            new_dist = dist[u] + weight

            if new_dist < dist[v]:
                dist[v] = new_dist
                parent[v] = u
                heapq.heappush(heap, (new_dist, v))

    return dist, parent


def get_path(parent, target):
    """Ricostruisce il cammino dalla sorgente a target."""
    path = []
    current = target

    while current != -1:
        path.append(current)
        current = parent[current]

    return list(reversed(path))


# Esempio
graph = {
    0: [(1, 4), (2, 1)],
    1: [(3, 1)],
    2: [(1, 2), (3, 5)],
    3: []
}

distances, parents = dijkstra(graph, 0, 4)

print("Distanze dalla sorgente 0:")
for i, d in enumerate(distances):
    print(f"  Vertice {i}: {d}")
    if d != float('inf'):
        path = get_path(parents, i)
        print(f"    Cammino: {' -> '.join(map(str, path))}")
```

11.7 Job Scheduling

11.7.1 Minimize Maximum Lateness

Problema: Schedule n job su una macchina, ognuno con durata t_i e deadline d_i. Minimizzare il massimo ritardo.

Strategia greedy: Ordina per deadline (Earliest Deadline First).

```
def minimize_lateness(jobs):
    """
    Minimizza il massimo ritardo.

```

```
    Args:
        jobs: lista di (duration, deadline)

    Returns:
        (max_lateness, schedule)

    Complessita: O(n log n)
    """
    # Ordina per deadline
    sorted_jobs = sorted(jobs, key=lambda x: x[1])

    schedule = []
    time = 0
    max_lateness = 0

    for duration, deadline in sorted_jobs:
        start = time
        finish = time + duration
        lateness = max(0, finish - deadline)

        schedule.append({
            'start': start,
            'finish': finish,
            'deadline': deadline,
            'lateness': lateness
        })

        time = finish
        max_lateness = max(max_lateness, lateness)

    return max_lateness, schedule
```

11.8 Intervalli su Linea

11.8.1 Interval Covering

Coprire una linea $[0, L]$ con il minimo numero di intervalli.

```
def interval_covering(intervals, L):
    """
    Copri [0, L] con minimo numero di intervalli.

    Greedy: scegli sempre l'intervallo che estende
    di piu la copertura corrente.

    Complessita: O(n log n)
    """
    # Ordina per inizio
    intervals.sort()

    covered = 0
    selected = []
    i = 0
    n = len(intervals)

    while covered < L and i < n:
        # Salta intervalli che non estendono copertura
```

```
        if intervals[i][0] > covered:
            return None  # Impossibile coprire

        # Trova intervallo che estende di piu
        max_end = covered
        best_interval = None

        while i < n and intervals[i][0] <= covered:
            if intervals[i][1] > max_end:
                max_end = intervals[i][1]
                best_interval = intervals[i]
            i += 1

        if best_interval is None:
            return None

        selected.append(best_interval)
        covered = max_end

    if covered < L:
        return None

    return selected
```

11.9 Quando Greedy NON Funziona

11.9.1 0/1 Knapsack

Greedy (rapporto valore/peso) non funziona:

Esempio:

- Capacità: 10
- Oggetti: (peso=10, valore=100), (peso=1, valore=11) $\times$ 10
- Greedy sceglie il primo: valore = 100
- Ottimo sceglie i 10 piccoli: valore = 110

11.9.2 Longest Path

Trovare il cammino più lungo in un grafo: greedy fallisce, serve DP o esplorazione esaustiva.

11.10 Esercizi

1. Dimostrare formalmente la correttezza di:
   (a) Activity Selection
   (b) Fractional Knapsack
   (c) Huffman Coding
2. Implementare e analizzare:
   (a) Job scheduling con profitti
   (b) Interval partitioning (minimo numero di risorse)

   (c) Gas station problem

3. Confrontare sperimentalmente:

   (a) Kruskal vs Prim su grafi densi e sparsi

   (b) Dijkstra vs Bellman-Ford

4. Problemi avanzati:

   (a) Set Cover con approssimazione greedy

   (b) Vertex Cover con greedy

   (c) Traveling Salesman con euristica greedy

Capitolo 12

Backtracking Avanzato

12.1 Introduzione

Il **backtracking** è una tecnica generale per trovare soluzioni a problemi di ricerca combinatoria costruendo incrementalmente candidati e abbandonando quelli che non possono portare a soluzioni valide.

12.1.1 Schema Generale

Backtracking Template [1] Backtrack$solution, data$ IsComplete$solution$ ProcessSolution$solution$ ogni $candidate$ in GetCandidates$solution, data$ IsValid$solution, candidate$ Add$solution, candidate$ Backtrack$solution, data$ Remove$solution, candidate$ Backtrack!

12.1.2 Componenti Chiave

1. **Scelta**: selezionare un candidato da aggiungere alla soluzione parziale
2. **Vincoli**: verificare se la scelta è valida
3. **Obiettivo**: determinare se la soluzione è completa
4. **Backtrack**: annullare la scelta e provare alternative

12.1.3 Ottimizzazioni

- **Pruning**: tagliare rami che sicuramente non portano a soluzioni
- **Constraint propagation**: dedurre vincoli da scelte precedenti
- **Heuristics**: ordinare le scelte per esplorare prima i rami promettenti
- **Memoization**: evitare di ricalcolare sottoproblemi identici

12.2 N-Queens Problem

12.2.1 Definizione

Posizionare N regine su una scacchiera $N \times N$ in modo che nessuna regina minacci un'altra.

Vincoli: Due regine non possono essere:

- Sulla stessa riga
- Sulla stessa colonna
- Sulla stessa diagonale

12.2.2 Rappresentazione

Usiamo un array `queens[i]` che indica la colonna della regina nella riga i.

12.2.3 Verifica Validità

Due regine in posizioni (r_1, c_1) e (r_2, c_2) si minacciano se:

- $r_1 = r_2$ (stessa riga)
- $c_1 = c_2$ (stessa colonna)
- $|r_1 - r_2| = |c_1 - c_2|$ (stessa diagonale)

12.2.4 Pseudocodice

N-Queens [1] SolveNQueens$n$ $queens \leftarrow$ array di dimensione n PlaceQueens$queens, 0, n$ PlaceQueens$queens, row, n$ $row = n$ PrintSolution$queens$ $col \leftarrow 0$ $n - 1$ IsSafe$queens, row, col$ $queens[row] \leftarrow col$ PlaceQueens$queens, row{+}1, n$ $queens[row] \leftarrow -1$ Backtrack IsSafe$queens, row, col$ $i \leftarrow 0$ $row{-}1$ $other\_col \leftarrow queens[i]$ $other\_col = col$ Stessa colonna $|row{-}i| = |col{-}other\_col|$ Stessa diagonale

12.2.5 Analisi di Complessità

Caso peggiore: $O(N!)$ - dobbiamo esplorare tutte le permutazioni nel peggiore dei casi.

In pratica: Il pruning riduce drasticamente lo spazio di ricerca. Per $N = 8$, solo ~ 2000 nodi invece di $8! = 40320$.

12.2.6 Implementazioni Python

```python
def solve_n_queens(n):
    """
    Risolve il problema delle N regine.

    Args:
        n: dimensione della scacchiera

    Returns:
        lista di tutte le soluzioni

    Complessita: O(N!) worst case
    """
    solutions = []

    def is_safe(queens, row, col):
        """Verifica se posizionare regina in (row, col) e sicuro."""
        for i in range(row):
            other_col = queens[i]

            # Stessa colonna
            if other_col == col:
                return False

            # Stessa diagonale
            if abs(row - i) == abs(col - other_col):
                return False
```

```
27
28          return True
29
30      def backtrack(row, queens):
31          """Piazza regine riga per riga."""
32          if row == n:
33              # Soluzione completa
34              solutions.append(queens[:])
35              return
36
37          for col in range(n):
38              if is_safe(queens, row, col):
39                  queens[row] = col
40                  backtrack(row + 1, queens)
41                  queens[row] = -1  # Backtrack
42
43      backtrack(0, [-1] * n)
44      return solutions
45
46
47  def print_board(queens):
48      """Stampa la scacchiera con le regine."""
49      n = len(queens)
50      for row in range(n):
51          line = []
52          for col in range(n):
53              if queens[row] == col:
54                  line.append('Q')
55              else:
56                  line.append('.')
57          print(' '.join(line))
58      print()
59
60
61  # Versione ottimizzata con set per controlli O(1)
62  def solve_n_queens_optimized(n):
63      """
64      Versione ottimizzata con set per tracking.
65      """
66      solutions = []
67
68      def backtrack(row, queens, cols, diag1, diag2):
69          """
70          cols: colonne occupate
71          diag1: diagonali \ occupate (row - col)
72          diag2: diagonali / occupate (row + col)
73          """
74          if row == n:
75              solutions.append(queens[:])
76              return
77
78          for col in range(n):
79              # Calcola identificatori diagonali
80              d1 = row - col
81              d2 = row + col
82
83              # Verifica se la posizione e sicura
84              if col in cols or d1 in diag1 or d2 in diag2:
```

```
                continue

            # Piazza regina
            queens[row] = col
            cols.add(col)
            diag1.add(d1)
            diag2.add(d2)

            # Ricorsione
            backtrack(row + 1, queens, cols, diag1, diag2)

            # Backtrack
            queens[row] = -1
            cols.remove(col)
            diag1.remove(d1)
            diag2.remove(d2)

    backtrack(0, [-1] * n, set(), set(), set())
    return solutions


# Test
solutions = solve_n_queens_optimized(8)
print(f"Numero di soluzioni per 8-Queens: {len(solutions)}")
# Output: 92

print("\nPrima soluzione:")
print_board(solutions[0])
```

12.2.7 Variante: Count Solutions

Se serve solo il conteggio, non serve salvare tutte le soluzioni:

```
def count_n_queens(n):
    """
    Conta solo il numero di soluzioni.
    Piu efficiente in memoria.
    """
    count = [0]  # Usa lista per mutabilita

    def backtrack(row, cols, diag1, diag2):
        if row == n:
            count[0] += 1
            return

        for col in range(n):
            d1, d2 = row - col, row + col

            if col not in cols and d1 not in diag1 and d2 not in diag2:
                backtrack(row + 1,
                          cols | {col},
                          diag1 | {d1},
                          diag2 | {d2})

    backtrack(0, set(), set(), set())
    return count[0]
```

12.3 Sudoku Solver

12.3.1 Definizione

Riempire una griglia 9×9 con cifre da 1 a 9 rispettando i vincoli:

- Ogni riga contiene ogni cifra esattamente una volta
- Ogni colonna contiene ogni cifra esattamente una volta
- Ogni sottgriglia 3×3 contiene ogni cifra esattamente una volta

12.3.2 Strategia

1. Trova una cella vuota
2. Prova ogni cifra da 1 a 9
3. Se la cifra è valida, inseriscila e ricorri
4. Se la ricorsione fallisce, backtrack e prova la prossima cifra
5. Se nessuna cifra funziona, ritorna False

12.3.3 Pseudocodice

Sudoku Solver [1] SolveSudoku$board$ (row, col) $\leftarrow$ FindEmptyCell$board$ (row, col) $=$ None Risolto! $num \leftarrow 1$ 9 IsValid$board, row, col, num$ $board[row][col] \leftarrow num$ SolveSudoku$board$ $board[row][col] \leftarrow 0$ Backtrack IsValid$board, row, col, num$ $i \leftarrow 0$ 8 $board[row][i] = num$ In riga $board[i][col] = num$ In colonna $box\_row \leftarrow 3 \times \lfloor row/3 \rfloor$ $box\_col \leftarrow 3 \times \lfloor col/3 \rfloor$ $i \leftarrow 0$ 2 $j \leftarrow 0$ 2 $board[box\_row + i][box\_col + j] = num$ In box 3x3

12.3.4 Implementazione Python

```python
def solve_sudoku(board):
    """
    Risolve un Sudoku 9x9.

    Args:
        board: matrice 9x9 con 0 per celle vuote

    Returns:
        True se risolto, False se impossibile

    Complessita: O(9^m) dove m = numero celle vuote
    """
    def is_valid(row, col, num):
        """Verifica se num e valido in (row, col)."""
        # Controlla riga
        if num in board[row]:
            return False

        # Controlla colonna
        if num in [board[i][col] for i in range(9)]:
            return False

        # Controlla box 3x3
        box_row, box_col = 3 * (row // 3), 3 * (col // 3)
```

```
        for i in range(box_row, box_row + 3):
            for j in range(box_col, box_col + 3):
                if board[i][j] == num:
                    return False

        return True

    def find_empty():
        """Trova prossima cella vuota."""
        for i in range(9):
            for j in range(9):
                if board[i][j] == 0:
                    return i, j
        return None

    def backtrack():
        """Risolve con backtracking."""
        cell = find_empty()
        if cell is None:
            return True  # Risolto!

        row, col = cell

        for num in range(1, 10):
            if is_valid(row, col, num):
                board[row][col] = num

                if backtrack():
                    return True

                # Backtrack
                board[row][col] = 0

        return False

    backtrack()
    return board


# Versione ottimizzata con constraint tracking
def solve_sudoku_optimized(board):
    """
    Versione ottimizzata con tracking di possibilita.
    """
    # Track numeri disponibili per riga/col/box
    rows = [set(range(1, 10)) for _ in range(9)]
    cols = [set(range(1, 10)) for _ in range(9)]
    boxes = [set(range(1, 10)) for _ in range(9)]

    # Inizializza constraint sets
    empty_cells = []
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                empty_cells.append((i, j))
            else:
                num = board[i][j]
                rows[i].discard(num)
```

```
                cols[j].discard(num)
                box_idx = (i // 3) * 3 + (j // 3)
                boxes[box_idx].discard(num)

    def backtrack(idx):
        if idx == len(empty_cells):
            return True

        row, col = empty_cells[idx]
        box_idx = (row // 3) * 3 + (col // 3)

        # Intersezione: numeri validi
        possible = rows[row] & cols[col] & boxes[box_idx]

        for num in possible:
            # Piazza numero
            board[row][col] = num
            rows[row].discard(num)
            cols[col].discard(num)
            boxes[box_idx].discard(num)

            if backtrack(idx + 1):
                return True

            # Backtrack
            board[row][col] = 0
            rows[row].add(num)
            cols[col].add(num)
            boxes[box_idx].add(num)

        return False

    backtrack(0)
    return board


# Esempio
sudoku = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]

solution = solve_sudoku_optimized([row[:] for row in sudoku])
for row in solution:
    print(row)
```

12.3.5 Ottimizzazione: MRV Heuristic

Minimum Remaining Values: Scegli la cella con meno valori possibili.

```
def solve_sudoku_mrv(board):
```

```
    """
    Sudoku solver con euristica MRV.
    Sceglie la cella con meno possibilita.
    """
    def get_candidates(row, col):
        """Ritorna numeri possibili per (row, col)."""
        if board[row][col] != 0:
            return set()

        candidates = set(range(1, 10))

        # Rimuovi numeri in riga
        candidates -= set(board[row])

        # Rimuovi numeri in colonna
        candidates -= {board[i][col] for i in range(9)}

        # Rimuovi numeri in box
        box_row, box_col = 3 * (row // 3), 3 * (col // 3)
        for i in range(box_row, box_row + 3):
            for j in range(box_col, box_col + 3):
                candidates.discard(board[i][j])

        return candidates

    def find_best_cell():
        """Trova cella con minimo numero di candidati (MRV)."""
        min_candidates = 10
        best_cell = None

        for i in range(9):
            for j in range(9):
                if board[i][j] == 0:
                    candidates = get_candidates(i, j)
                    if len(candidates) < min_candidates:
                        min_candidates = len(candidates)
                        best_cell = (i, j, candidates)
                        if min_candidates == 0:
                            return best_cell

        return best_cell

    def backtrack():
        result = find_best_cell()
        if result is None:
            return True  # Completato

        row, col, candidates = result

        if not candidates:
            return False  # Nessuna soluzione

        for num in candidates:
            board[row][col] = num

            if backtrack():
                return True

```

```
            board[row][col] = 0

        return False

    backtrack()
    return board
```

12.4 Graph Coloring

12.4.1 Definizione

Assegnare colori ai vertici di un grafo in modo che vertici adiacenti abbiano colori diversi, usando il minimo numero di colori.

12.4.2 Problema Decisionale

Dato un grafo G e un intero k, è possibile colorare G con al massimo k colori?

12.4.3 Pseudocodice

Graph Coloring [1] GraphColoring$graph, k$ $colors \leftarrow$ array di dimensione $|V|$ inizializzato a -1 ColorVertex$graph, colors, 0, k$ ColorVertex$graph, colors, v, k$ $v = |V|$ Tutti i vertici colorati $c \leftarrow 0$ $k-1$ IsSafeColor$graph, colors, v, c$ $colors[v] \leftarrow c$ ColorVertex$graph, colors, v+1, k$ $colors[v] \leftarrow -1$ Backtrack IsSafeColor$graph, colors, v, c$ ogni vicino u di v $colors[u] = c$

12.4.4 Implementazioni Python

```
def graph_coloring(graph, k):
    """
    Colora grafo con al massimo k colori.

    Args:
        graph: dict {vertex: [neighbors]}
        k: numero massimo di colori

    Returns:
        dict {vertex: color} se possibile, None altrimenti

    Complessita: O(k^n) worst case
    """
    n = len(graph)
    colors = {}

    def is_safe(vertex, color):
        """Verifica se color e valido per vertex."""
        for neighbor in graph.get(vertex, []):
            if colors.get(neighbor) == color:
                return False
        return True

    def backtrack(vertices):
        """Colora vertici ricorsivamente."""
        if not vertices:
            return True  # Tutti colorati

```

```
        vertex = vertices[0]
        remaining = vertices[1:]

        for color in range(k):
            if is_safe(vertex, color):
                colors[vertex] = color

                if backtrack(remaining):
                    return True

                del colors[vertex]  # Backtrack

        return False

    vertices = list(graph.keys())
    if backtrack(vertices):
        return colors
    return None


# Esempio: Grafo con 4 vertici
graph = {
    0: [1, 2, 3],
    1: [0, 2],
    2: [0, 1, 3],
    3: [0, 2]
}

coloring = graph_coloring(graph, 3)
print(f"Colorazione: {coloring}")
# Output: {0: 0, 1: 1, 2: 2, 3: 1}


def chromatic_number(graph):
    """
    Trova il numero cromatico (minimo numero di colori).
    """
    n = len(graph)

    # Prova da 1 a n colori
    for k in range(1, n + 1):
        if graph_coloring(graph, k):
            return k

    return n


# Versione con ordinamento dei vertici (euristica)
def graph_coloring_optimized(graph, k):
    """
    Versione con euristica: colora prima vertici con piu vicini.
    """
    colors = {}

    # Ordina vertici per grado decrescente
    vertices = sorted(graph.keys(),
                      key=lambda v: len(graph.get(v, [])),
                      reverse=True)
```

```
87
88      def is_safe(vertex, color):
89          for neighbor in graph.get(vertex, []):
90              if colors.get(neighbor) == color:
91                  return False
92          return True
93
94      def backtrack(idx):
95          if idx == len(vertices):
96              return True
97
98          vertex = vertices[idx]
99
100         for color in range(k):
101             if is_safe(vertex, color):
102                 colors[vertex] = color
103
104                 if backtrack(idx + 1):
105                     return True
106
107                 del colors[vertex]
108
109         return False
110
111     if backtrack(0):
112         return colors
113     return None
```

12.4.5 Greedy Approximation

Un approccio greedy (non ottimo ma veloce):

```
1   def greedy_coloring(graph):
2       """
3       Colorazione greedy (non ottimale).
4
5       Complessita: O(V + E)
6       Garanzia: usa al massimo Delta + 1 colori
7       (Delta = grado massimo)
8       """
9       colors = {}
10
11      # Ordina vertici per grado decrescente (euristica)
12      vertices = sorted(graph.keys(),
13                        key=lambda v: len(graph.get(v, [])),
14                        reverse=True)
15
16      for vertex in vertices:
17          # Trova colori usati dai vicini
18          neighbor_colors = {colors.get(n) for n in graph.get(vertex, [])
19                             if n in colors}
20
21          # Usa il primo colore disponibile
22          color = 0
23          while color in neighbor_colors:
24              color += 1
25
26          colors[vertex] = color
```

```
    return colors


# Test
coloring = greedy_coloring(graph)
print(f"Greedy coloring: {coloring}")
print(f"Numero colori usati: {max(coloring.values()) + 1}")
```

12.5 Altri Problemi Classici

12.5.1 Hamiltonian Path

Trovare un cammino che visita ogni vertice esattamente una volta:

```
def hamiltonian_path(graph, start):
    """
    Trova un cammino hamiltoniano partendo da start.

    Complessita: O(n!)
    """
    n = len(graph)
    path = [start]
    visited = {start}

    def backtrack():
        if len(path) == n:
            return True

        current = path[-1]

        for neighbor in graph.get(current, []):
            if neighbor not in visited:
                path.append(neighbor)
                visited.add(neighbor)

                if backtrack():
                    return True

                path.pop()
                visited.remove(neighbor)

        return False

    if backtrack():
        return path
    return None
```

12.5.2 Word Search

Trovare una parola in una griglia di lettere:

```
def word_search(board, word):
    """
    Cerca word nella board con backtracking.

    Complessita: O(m * n * 4^L) dove L = len(word)
```

```
    """
    rows, cols = len(board), len(board[0])

    def backtrack(row, col, idx):
        # Parola completa trovata
        if idx == len(word):
            return True

        # Fuori limiti o lettera sbagliata
        if (row < 0 or row >= rows or col < 0 or col >= cols or
            board[row][col] != word[idx]):
            return False

        # Marca come visitata temporaneamente
        temp = board[row][col]
        board[row][col] = '#'

        # Esplora 4 direzioni
        found = (backtrack(row + 1, col, idx + 1) or
                 backtrack(row - 1, col, idx + 1) or
                 backtrack(row, col + 1, idx + 1) or
                 backtrack(row, col - 1, idx + 1))

        # Ripristina
        board[row][col] = temp

        return found

    # Prova da ogni cella
    for i in range(rows):
        for j in range(cols):
            if backtrack(i, j, 0):
                return True

    return False
```

12.5.3 Partition Equal Subset Sum

Partizionare un array in due sottoinsiemi con somma uguale:

```
def can_partition(nums):
    """
    Verifica se l'array puo essere partizionato in due
    sottoinsiemi con somma uguale.

    Backtracking (alternativa a DP).
    """
    total = sum(nums)

    if total % 2 != 0:
        return False

    target = total // 2

    def backtrack(idx, current_sum):
        if current_sum == target:
            return True

```

```
        if idx >= len(nums) or current_sum > target:
            return False

        # Include nums[idx]
        if backtrack(idx + 1, current_sum + nums[idx]):
            return True

        # Esclude nums[idx]
        if backtrack(idx + 1, current_sum):
            return True

        return False

    return backtrack(0, 0)
```

12.6 Tecniche di Ottimizzazione

12.6.1 Branch and Bound

Mantiene il miglior risultato trovato finora e taglia rami che non possono migliorarlo:

```
def branch_and_bound_example():
    """
    Esempio di Branch and Bound per minimizzazione.
    """
    best_solution = None
    best_cost = float('inf')

    def backtrack(partial_solution, current_cost, bound):
        nonlocal best_solution, best_cost

        # Pruning: se il bound e peggiore del best, taglia
        if bound >= best_cost:
            return

        if is_complete(partial_solution):
            if current_cost < best_cost:
                best_cost = current_cost
                best_solution = partial_solution[:]
            return

        for choice in get_choices(partial_solution):
            new_cost = current_cost + cost(choice)
            new_bound = compute_bound(partial_solution + [choice])

            backtrack(partial_solution + [choice],
                      new_cost,
                      new_bound)
```

12.6.2 Constraint Propagation

Deduce vincoli da scelte precedenti:

```
def constraint_propagation_example():
    """
    Esempio di constraint propagation.
    """
```

```
    # Mantieni domini di valori possibili
    domains = {var: set(possible_values)
               for var in variables}

    def propagate_constraints(var, value):
        """Riduce domini dopo assegnazione."""
        # Rimuovi value dal dominio di var
        domains[var] = {value}

        # Propaga vincoli ai vicini
        for neighbor in get_neighbors(var):
            if value in domains[neighbor]:
                domains[neighbor].remove(value)

                # Se dominio vuoto, fallimento
                if not domains[neighbor]:
                    return False

        return True
```

12.7 Esercizi

1. Implementare e analizzare:
   (a) Knight's Tour (cammino del cavallo)
   (b) Crossword Puzzle Solver
   (c) Latin Square
2. Ottimizzazioni:
   (a) Implementare N-Queens con bitwise operations
   (b) Confrontare MRV vs altre euristiche per Sudoku
   (c) Analizzare pruning effectiveness
3. Problemi avanzati:
   (a) Generazione di labirinti con backtracking
   (b) Satisfiability (SAT) solver semplificato
   (c) Constraint Satisfaction Problem (CSP) generico
4. Analisi empirica:
   (a) Misurare speedup delle ottimizzazioni
   (b) Confrontare backtracking vs DP su problemi comuni
   (c) Profiling di algoritmi backtracking

Appendice A

Appendice: Tabelle di Complessità

A.1 Introduzione

Questa appendice raccoglie le complessità temporali e spaziali di tutti gli algoritmi trattati nel corso, organizzate per categoria.

A.2 Algoritmi di Ordinamento

A.2.1 Confronto Generale

Tabella A.1: Complessità algoritmi di ordinamento

Algoritmo	**Best**	**Average**	**Worst**	**Spazio**	**Stabile**	**In-place**	**Adattivo**
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sì	Sì	Sì
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Sì	No
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sì	Sì	Sì
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Sì	No	No
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No	Sì	No
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Sì	No

A.2.2 Dettagli Aggiuntivi

A.2.3 Algoritmi Specializzati

A.3 Algoritmi di Ricerca

A.3.1 Ricerca in Strutture Dati

A.4 Programmazione Dinamica

A.4.1 Problemi su Griglie

A.5 Algoritmi Greedy

A.6 Backtracking

A.7 Grafi - Traversal

A.8 Grafi - Shortest Paths

A.9 Alberi

A.10 Ricorsione - Problemi Classici

A.11 Master Theorem

Il **Master Theorem** fornisce soluzioni dirette per ricorrenze della forma:

$$T(n) = aT(n/b) + f(n)$$

dove $a \geq 1$, $b > 1$ e $f(n)$ è asintoticamente positiva.

A.11.1 Esempi di Applicazione

A.12 Notazioni Asintotiche

A.12.1 Definizioni

A.12.2 Gerarchia di Complessità

In ordine crescente:

$$\begin{aligned} O(1) &< O(\log\log n) < O(\log n) < O(\sqrt{n}) < O(n) \\ &< O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n) \end{aligned}$$

A.12.3 Proprietà delle Notazioni

- **Transitività**: Se $f = O(g)$ e $g = O(h)$, allora $f = O(h)$
- **Riflessività**: $f = O(f)$
- **Simmetria** (per Θ): Se $f = \Theta(g)$, allora $g = \Theta(f)$
- **Somma**: $O(f) + O(g) = O(\max(f, g))$
- **Prodotto**: $O(f) \cdot O(g) = O(f \cdot g)$

A.13 Classi di Complessità

A.14 Complessità Spaziali Comuni

A.15 Ottimizzazioni Comuni

A.16 Regole Pratiche

A.16.1 Stima della Complessità dal Codice

1. **Loop singolo su** n: $O(n)$
2. **Loop annidati su** n: $O(n^k)$ dove k = numero di livelli
3. **Divisione per 2 ad ogni passo**: $O(\log n)$
4. **Chiamate ricorsive multiple**: analizzare albero di ricorsione
5. **Ricorsione con memoization**: numero di stati unici

A.16.2 Limiti Pratici per Tempo di Esecuzione

Assumendo 10^8 operazioni/secondo:

Tabella A.2: Caratteristiche dettagliate algoritmi di ordinamento

Algoritmo	**Caratteristiche e Note**
Bubble Sort	[nosep,left=0pt] • Scambi: $O(n^2)$ nel caso peggiore • Confronti: sempre $O(n^2)$ • Ottimizzabile con early stopping • Utile solo per scopi didattici
Selection Sort	[nosep,left=0pt] • Scambi: al massimo $O(n)$ (minimo tra tutti) • Confronti: sempre $\Theta(n^2)$ • Non adattivo: prestazioni costanti • Utile quando scambi sono costosi
Insertion Sort	[nosep,left=0pt] • Ottimo per array piccoli ($n < 50$) • Molto efficiente su array quasi ordinati • Online: può ordinare stream di dati • Usato in algoritmi ibridi (Timsort, Introsort)
Merge Sort	[nosep,left=0pt] • Garantisce $O(n \log n)$ sempre • Eccellente per linked list ($O(1)$ spazio) • Parallelizzabile • Base di Timsort (Python, Java)
Quick Sort	[nosep,left=0pt] • Migliori prestazioni in pratica (costanti piccole) • Randomizzazione: $O(n \log n)$ atteso • Cache-friendly • 3-way partition per duplicati
Heap Sort	[nosep,left=0pt] • Combina vantaggi di Merge Sort (garantito) e Quick Sort (in-place) • BuildMaxHeap richiede solo $O(n)$ • Non cache-friendly • Usato in priority queue

Tabella A.3: Algoritmi di ordinamento non basati su confronti

Algoritmo	**Tempo**	**Spazio**	**Stabile**	**Vincoli/Note**
Counting Sort	$O(n+k)$	$O(k)$	Sì	k = range valori. Ottimo per $k = O(n)$
Radix Sort	$O(d(n+k))$	$O(n+k)$	Sì	d = numero cifre, k = base
Bucket Sort	$O(n)$	$O(n)$	Sì	Distribuzione uniforme richiesta

Tabella A.4: Complessità algoritmi di ricerca

Algoritmo	**Best**	**Average**	**Worst**	**Precondizioni**
Linear Search	$O(1)$	$O(n)$	$O(n)$	Nessuna
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$	Array ordinato
Interpolation	$O(1)$	$O(\log \log n)$	$O(n)$	Ordinato + distribuzione uniforme
Exponential	$O(1)$	$O(\log i)$	$O(\log i)$	Ordinato, i = posizione
Jump Search	$O(1)$	$O(\sqrt{n})$	$O(\sqrt{n})$	Array ordinato

Tabella A.5: Complessità ricerca in strutture dati

Struttura	**Search**	**Insert**	**Delete**	**Spazio**
Array non ordinato	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Array ordinato	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
Linked List	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Hash Table (avg)	$O(1)$	$O(1)$	$O(1)$	$O(n)$
Hash Table (worst)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
BST (avg)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
BST (worst)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
AVL Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
B-Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

Tabella A.6: Complessità problemi di programmazione dinamica

Problema	**Tempo**	**Spazio**	**Note**
Fibonacci	$O(n)$	$O(n)$	Ottimizzabile a $O(1)$ spazio
0/1 Knapsack	$O(nW)$	$O(nW)$	W = capacità. Pseudo-polinomiale
Unbounded Knapsack	$O(nW)$	$O(W)$	Spazio ottimizzato
LCS	$O(mn)$	$O(mn)$	m, n = lunghezze stringhe
Edit Distance	$O(mn)$	$O(mn)$	Ottimizzabile a $O(\min(m,n))$
LIS	$O(n^2)$	$O(n)$	DP classico
LIS (ottimizzato)	$O(n \log n)$	$O(n)$	Con binary search
Matrix Chain	$O(n^3)$	$O(n^2)$	n = numero matrici
Coin Change	$O(nS)$	$O(S)$	S = somma target
Rod Cutting	$O(n^2)$	$O(n)$	n = lunghezza asta
Subset Sum	$O(nS)$	$O(nS)$	Pseudo-polinomiale

Tabella A.7: DP su griglie

Problema	**Tempo**	**Spazio**
Unique Paths	$O(mn)$	$O(mn)$ ottimizzabile a $O(n)$
Min Path Sum	$O(mn)$	$O(mn)$ ottimizzabile a $O(n)$
Maximal Square	$O(mn)$	$O(mn)$
Dungeon Game	$O(mn)$	$O(mn)$

Tabella A.8: Complessità algoritmi greedy

Problema	**Tempo**	**Spazio**	**Note**
Activity Selection	$O(n \log n)$	$O(1)$	Ordinamento per tempo di fine
Fractional Knapsack	$O(n \log n)$	$O(1)$	Ordinamento per ratio valore/peso
Huffman Coding	$O(n \log n)$	$O(n)$	Con binary heap
Kruskal MST	$O(E \log E)$	$O(V)$	$= O(E \log V)$, con Union-Find
Prim MST	$O(E \log V)$	$O(V)$	Con binary heap
Prim (Fibonacci)	$O(E + V \log V)$	$O(V)$	Con Fibonacci heap
Dijkstra	$O((V + E) \log V)$	$O(V)$	Con binary heap, pesi ≥ 0
Dijkstra (Fibonacci)	$O(E + V \log V)$	$O(V)$	Con Fibonacci heap
Job Scheduling	$O(n \log n)$	$O(1)$	Minimize lateness
Interval Covering	$O(n \log n)$	$O(n)$	Ordinamento necessario

Tabella A.9: Complessità algoritmi backtracking

Problema	**Tempo (worst)**	**Spazio**	**Note**
N-Queens	$O(N!)$	$O(N)$	Pruning riduce drasticamente
Sudoku	$O(9^m)$	$O(m)$	m = celle vuote
Graph Coloring	$O(k^V)$	$O(V)$	k = colori, V = vertici
Hamiltonian Path	$O(N!)$	$O(N)$	NP-completo
Subset Sum	$O(2^n)$	$O(n)$	Con pruning migliora
Permutazioni	$O(n \cdot n!)$	$O(n)$	Genera tutte le permutazioni
Combinazioni $C(n, k)$	$O(C(n, k) \cdot k)$	$O(k)$	$C(n, k) = \frac{n!}{k!(n-k)!}$
Sottoinsiemi	$O(2^n \cdot n)$	$O(n)$	Genera tutti i sottoinsiemi

Tabella A.10: Algoritmi di attraversamento grafi

Algoritmo	**Tempo**	**Spazio**	**Uso**
DFS (ricorsivo)	$O(V + E)$	$O(V)$	Stack ricorsivo, rilevamento cicli
DFS (iterativo)	$O(V + E)$	$O(V)$	Stack esplicito
BFS	$O(V + E)$	$O(V)$	Shortest path (non pesato), livelli
Topological Sort	$O(V + E)$	$O(V)$	DAG, ordinamento dipendenze

Tabella A.11: Algoritmi shortest path

Algoritmo	**Tempo**	**Spazio**	**Caratteristiche**
BFS	$O(V + E)$	$O(V)$	Grafi non pesati
Dijkstra (binary heap)	$O((V + E) \log V)$	$O(V)$	Pesi ≥ 0
Dijkstra (Fibonacci heap)	$O(E + V \log V)$	$O(V)$	Pesi ≥ 0, ottimale
Bellman-Ford	$O(VE)$	$O(V)$	Pesi negativi, rileva cicli negativi
Floyd-Warshall	$O(V^3)$	$O(V^2)$	All-pairs, pesi negativi OK
Johnson	$O(V^2 \log V + VE)$	$O(V^2)$	All-pairs, pesi negativi

Tabella A.12: Operazioni su alberi

Operazione	**Tempo**	**Spazio**	**Note**
Traversal (in/pre/post-order)	$O(n)$	$O(h)$	h = altezza
Height	$O(n)$	$O(h)$	Ricorsivo
Count Nodes	$O(n)$	$O(h)$	Ricorsivo
Search BST (avg)	$O(\log n)$	$O(1)$	Iterativo
Search BST (worst)	$O(n)$	$O(1)$	Albero sbilanciato
Insert BST	$O(\log n)$ avg	$O(h)$	Può richiedere bilanciamento
Delete BST	$O(\log n)$ avg	$O(h)$	Tre casi da considerare

Tabella A.13: Complessità problemi ricorsivi

Problema	**Tempo**	**Spazio**	**Ricorrenza**
Fattoriale	$O(n)$	$O(n)$	$T(n) = T(n-1) + O(1)$
Fibonacci (naive)	$O(2^n)$	$O(n)$	$T(n) = T(n-1)+T(n-2)$
Fibonacci (memo)	$O(n)$	$O(n)$	Con memoization
Hanoi	$O(2^n)$	$O(n)$	$T(n) = 2T(n-1)+O(1)$
Binary Search	$O(\log n)$	$O(\log n)$	$T(n) = T(n/2) + O(1)$
Merge Sort	$O(n \log n)$	$O(n)$	$T(n) = 2T(n/2) + O(n)$
Quick Sort (avg)	$O(n \log n)$	$O(\log n)$	$T(n) = 2T(n/2) + O(n)$
Quick Sort (worst)	$O(n^2)$	$O(n)$	$T(n) = T(n-1) + O(n)$

Tabella A.14: Casi del Master Theorem

Caso	**Condizione**	**Soluzione**
1	$f(n) = O(n^{\log_b a - \epsilon})$ per qualche $\epsilon > 0$	$T(n) = \Theta(n^{\log_b a})$
2	$f(n) = \Theta(n^{\log_b a} \log^k n)$ per qualche $k \geq 0$	$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3	$f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche $\epsilon > 0$ e $af(n/b) \leq cf(n)$ per $c < 1$	$T(n) = \Theta(f(n))$

Tabella A.15: Master Theorem - Esempi

Ricorrenza	a	b	**Caso**	**Soluzione**
$T(n) = 2T(n/2) + O(1)$	2	2	1	$\Theta(n)$
$T(n) = 2T(n/2) + O(n)$	2	2	2	$\Theta(n \log n)$
$T(n) = 2T(n/2) + O(n^2)$	2	2	3	$\Theta(n^2)$
$T(n) = 4T(n/2) + O(n)$	4	2	1	$\Theta(n^2)$
$T(n) = T(n/2) + O(n)$	1	2	3	$\Theta(n)$

Tabella A.16: Notazioni asintotiche

Notazione	**Definizione**	**Significato**
$O(g(n))$	$\exists c, n_0 : 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0$	Upper bound
$\Omega(g(n))$	$\exists c, n_0 : 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0$	Lower bound
$\Theta(g(n))$	$f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$	Tight bound
$o(g(n))$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$	Strict upper bound
$\omega(g(n))$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$	Strict lower bound

Tabella A.17: Classi di complessità comuni

Classe	**Tempo**	**Esempi**
Costante	$O(1)$	Accesso array, operazioni aritmetiche
Logaritmica	$O(\log n)$	Binary search, operazioni su heap
Lineare	$O(n)$	Linear search, traversal semplici
Linearithmic	$O(n \log n)$	Merge sort, heap sort, sorting ottimali
Quadratica	$O(n^2)$	Bubble sort, selezione sort, nested loops
Cubica	$O(n^3)$	Floyd-Warshall, matrix multiplication naive
Polinomiale	$O(n^k)$	Algoritmi efficienti, classe P
Esponenziale	$O(2^n)$	Subset sum (forza bruta), backtracking
Fattoriale	$O(n!)$	Permutazioni, traveling salesman (brute force)

Tabella A.18: Spazio ausiliario per algoritmi comuni

Tecnica/Algoritmo	**Spazio**	**Note**
Iterazione semplice	$O(1)$	Solo variabili locali
Ricorsione lineare	$O(n)$	Stack depth = n
Ricorsione logaritmica	$O(\log n)$	Binary search, binary tree balanced
Memoization	$O(n)$ o più	Tabella per sottoproblemi
DP (1D)	$O(n)$	Array 1D
DP (2D)	$O(n^2)$ o $O(mn)$	Matrice per stati
Backtracking	$O(h)$	h = profondità ricorsione
Queue/Stack	$O(n)$	Per BFS/DFS

Tabella A.19: Tecniche di ottimizzazione

Tecnica	**Da**	**A**
Memoization	$O(2^n)$ ricorsione	$O(n)$ o $O(n^2)$
Two pointers	$O(n^2)$ nested loops	$O(n)$
Sliding window	$O(nk)$	$O(n)$
Binary search	$O(n)$ linear	$O(\log n)$
Hash table	$O(n)$ search	$O(1)$ average
Heap	$O(n)$ find min	$O(\log n)$ extract, $O(1)$ peek
Union-Find	$O(n)$ per op	$O(\alpha(n)) \approx O(1)$
Segment Tree	$O(n)$ range query	$O(\log n)$

Tabella A.20: Dimensioni gestibili per complessità diverse

Complessità	**Max n (1 sec)**	**Max n (1 min)**
$O(\log n)$	$\sim 10^{30}$	Praticamente illimitato
$O(n)$	10^8	6×10^9
$O(n \log n)$	5×10^6	2×10^8
$O(n^2)$	10^4	2.4×10^5
$O(n^3)$	460	3900
$O(2^n)$	26	32
$O(n!)$	11	12

Appendice B

Appendice: Esercizi con Soluzioni

B.1 Introduzione

Questa appendice contiene esercizi selezionati con soluzioni dettagliate, analisi di complessità e implementazioni Python.

B.2 Ordinamento

B.2.1 Esercizio 1: Dutch National Flag

Problema: Dato un array contenente solo 0, 1 e 2, ordinarlo in tempo $O(n)$ e spazio $O(1)$.

Esempio:

```
Input:  [2, 0, 1, 2, 1, 0]
Output: [0, 0, 1, 1, 2, 2]
```

Soluzione: Usa tre puntatori (three-way partition)

```python
def dutch_flag(arr):
    """
    Ordina array di 0, 1, 2.

    Complessita: O(n) tempo, O(1) spazio

    Invarianti:
    - arr[0..low-1] contiene 0
    - arr[low..mid-1] contiene 1
    - arr[high+1..n-1] contiene 2
    """
    low = 0      # Prossima posizione per 0
    mid = 0      # Elemento corrente
    high = len(arr) - 1  # Prossima posizione per 2

    while mid <= high:
        if arr[mid] == 0:
            # Scambia con low e avanza entrambi
            arr[low], arr[mid] = arr[mid], arr[low]
            low += 1
            mid += 1
        elif arr[mid] == 1:
            # Gia nella posizione corretta
            mid += 1
        else:  # arr[mid] == 2
            # Scambia con high e arretra high
```

```
            arr[mid], arr[high] = arr[high], arr[mid]
            high -= 1
            # Non incrementare mid (elemento scambiato non esaminato)

    return arr
```

Analisi:

- Ogni elemento viene visitato al massimo una volta
- Complessità temporale: $O(n)$
- Complessità spaziale: $O(1)$ - solo puntatori

B.2.2 Esercizio 2: Kth Largest Element

Problema: Trovare il k-esimo elemento più grande in un array non ordinato.

Soluzione 1: QuickSelect (average $O(n)$)

```
import random

def kth_largest(arr, k):
    """
    Trova k-esimo elemento piu grande usando QuickSelect.

    Complessita: O(n) average, O(n^2) worst case
    """
    def partition(left, right, pivot_idx):
        pivot = arr[pivot_idx]
        # Sposta pivot alla fine
        arr[pivot_idx], arr[right] = arr[right], arr[pivot_idx]

        store_idx = left
        for i in range(left, right):
            if arr[i] > pivot:  # Ordine decrescente
                arr[store_idx], arr[i] = arr[i], arr[store_idx]
                store_idx += 1

        # Metti pivot nella posizione finale
        arr[right], arr[store_idx] = arr[store_idx], arr[right]
        return store_idx

    def select(left, right, k):
        if left == right:
            return arr[left]

        # Randomized pivot
        pivot_idx = random.randint(left, right)
        pivot_idx = partition(left, right, pivot_idx)

        if k == pivot_idx:
            return arr[k]
        elif k < pivot_idx:
            return select(left, pivot_idx - 1, k)
        else:
            return select(pivot_idx + 1, right, k)

    return select(0, len(arr) - 1, k - 1)
```

Soluzione 2: Min-Heap ($O(n \log k)$)

```python
import heapq

def kth_largest_heap(arr, k):
    """
    Usa min-heap di dimensione k.

    Complessita: O(n log k)
    Spazio: O(k)
    """
    # Mantieni min-heap di k elementi piu grandi
    heap = arr[:k]
    heapq.heapify(heap)  # O(k)

    for num in arr[k:]:
        if num > heap[0]:
            heapq.heapreplace(heap, num)

    return heap[0]
```

B.3 Ricerca

B.3.1 Esercizio 3: First and Last Position

Problema: Dato un array ordinato con duplicati, trova le posizioni iniziale e finale di un target.
Esempio:

```
Input:  arr = [5,7,7,8,8,10], target = 8
Output: [3, 4]
```

Soluzione: Due ricerche binarie

```python
def search_range(arr, target):
    """
    Trova primo e ultimo indice di target.

    Complessita: O(log n)
    """
    def binary_search_left(arr, target):
        """Trova primo indice >= target."""
        left, right = 0, len(arr)

        while left < right:
            mid = left + (right - left) // 2
            if arr[mid] < target:
                left = mid + 1
            else:
                right = mid

        return left

    def binary_search_right(arr, target):
        """Trova primo indice > target."""
        left, right = 0, len(arr)

        while left < right:
            mid = left + (right - left) // 2
```

```
            if arr[mid] <= target:
                left = mid + 1
            else:
                right = mid

        return left

    start = binary_search_left(arr, target)

    # Target non trovato
    if start >= len(arr) or arr[start] != target:
        return [-1, -1]

    end = binary_search_right(arr, target) - 1

    return [start, end]
```

Prova di Correttezza:

- `binary_search_left` trova il primo indice i tale che $arr[i] \geq target$
- `binary_search_right` trova il primo indice j tale che $arr[j] > target$
- Quindi $arr[i..j-1]$ contiene tutti gli elementi uguali a $target$

B.3.2 Esercizio 4: Search in Rotated Array

Problema: Ricerca in array ordinato ruotato senza duplicati.

Soluzione:

```
def search_rotated(arr, target):
    """
    Ricerca in array ordinato ruotato.

    Complessita: O(log n)
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid

        # Determina quale meta e ordinata
        if arr[left] <= arr[mid]:  # Meta sinistra ordinata
            if arr[left] <= target < arr[mid]:
                right = mid - 1  # Target in meta ordinata
            else:
                left = mid + 1   # Target in meta non ordinata
        else:  # Meta destra ordinata
            if arr[mid] < target <= arr[right]:
                left = mid + 1   # Target in meta ordinata
            else:
                right = mid - 1  # Target in meta non ordinata

    return -1
```

B.4 Programmazione Dinamica

B.4.1 Esercizio 5: Longest Palindromic Substring

Problema: Trova la più lunga sottostringa palindroma.

Esempio:

```
Input:  "babad"
Output: "bab" o "aba"
```

Soluzione 1: DP ($O(n^2)$ tempo e spazio)

```python
def longest_palindrome_dp(s):
    """
    DP: dp[i][j] = True se s[i..j] e palindroma.

    Complessita: O(n^2) tempo, O(n^2) spazio
    """
    n = len(s)
    if n < 2:
        return s

    dp = [[False] * n for _ in range(n)]

    # Ogni singolo carattere e palindromo
    for i in range(n):
        dp[i][i] = True

    start = 0
    max_len = 1

    # Sottostringhe di lunghezza 2
    for i in range(n - 1):
        if s[i] == s[i + 1]:
            dp[i][i + 1] = True
            start = i
            max_len = 2

    # Sottostringhe di lunghezza >= 3
    for length in range(3, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1

            # s[i..j] e palindroma se:
            # - s[i] == s[j]
            # - s[i+1..j-1] e palindroma
            if s[i] == s[j] and dp[i + 1][j - 1]:
                dp[i][j] = True
                start = i
                max_len = length

    return s[start:start + max_len]
```

Soluzione 2: Expand Around Center ($O(n^2)$ tempo, $O(1)$ spazio)

```python
def longest_palindrome_expand(s):
    """
    Espandi intorno a ogni possibile centro.

    Complessita: O(n^2) tempo, O(1) spazio
```

```
    """
    def expand_around_center(left, right):
        """Espandi finche palindroma."""
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return right - left - 1  # Lunghezza

    if not s:
        return ""

    start = 0
    max_len = 0

    for i in range(len(s)):
        # Centro singolo (lunghezza dispari)
        len1 = expand_around_center(i, i)
        # Centro doppio (lunghezza pari)
        len2 = expand_around_center(i, i + 1)

        length = max(len1, len2)

        if length > max_len:
            max_len = length
            start = i - (length - 1) // 2

    return s[start:start + max_len]
```

B.4.2 Esercizio 6: Word Break

Problema: Determina se una stringa può essere segmentata in parole da un dizionario.

Esempio:

```
s = "leetcode"
wordDict = ["leet", "code"]
Output: True (can be segmented as "leet code")
```

Soluzione: DP

```
def word_break(s, word_dict):
    """
    Verifica se s puo essere segmentata in parole.

    dp[i] = True se s[0..i-1] puo essere segmentata

    Complessita: O(n^2 * m) dove m = lunghezza parola max
    """
    n = len(s)
    word_set = set(word_dict)  # O(1) lookup
    dp = [False] * (n + 1)
    dp[0] = True  # Stringa vuota

    for i in range(1, n + 1):
        for j in range(i):
            # Se s[0..j-1] segmentabile e s[j..i-1] nel dizionario
            if dp[j] and s[j:i] in word_set:
                dp[i] = True
                break
```

```
    return dp[n]


# Ottimizzazione: controlla solo lunghezze valide
def word_break_optimized(s, word_dict):
    """
    Ottimizzato: considera solo lunghezze nel dizionario.
    """
    word_set = set(word_dict)
    max_len = max(len(word) for word in word_dict)

    n = len(s)
    dp = [False] * (n + 1)
    dp[0] = True

    for i in range(1, n + 1):
        # Controlla solo finestre di lunghezza valida
        for j in range(max(0, i - max_len), i):
            if dp[j] and s[j:i] in word_set:
                dp[i] = True
                break

    return dp[n]
```

B.5 Algoritmi Greedy

B.5.1 Esercizio 7: Jump Game

Problema: Dato un array dove $arr[i]$ indica il massimo salto da posizione i, determina se è possibile raggiungere l'ultima posizione.

Esempio:

```
Input:  [2,3,1,1,4]
Output: True (0 -> 1 -> 4)
```

Soluzione: Greedy

```
def can_jump(nums):
    """
    Verifica se raggiungibile ultima posizione.

    Greedy: mantieni la posizione piu lontana raggiungibile.

    Complessita: O(n) tempo, O(1) spazio
    """
    max_reach = 0

    for i in range(len(nums)):
        # Se posizione corrente non raggiungibile
        if i > max_reach:
            return False

        # Aggiorna massima portata
        max_reach = max(max_reach, i + nums[i])

        # Early exit se gia raggiunta la fine
```

```
        if max_reach >= len(nums) - 1:
            return True

    return True


def min_jumps(nums):
    """
    Minimo numero di salti per raggiungere la fine.

    Complessita: O(n)
    """
    if len(nums) <= 1:
        return 0

    jumps = 0
    current_end = 0    # Fine dell'intervallo corrente
    farthest = 0       # Piu lontano raggiungibile

    for i in range(len(nums) - 1):
        farthest = max(farthest, i + nums[i])

        # Raggiunta fine intervallo corrente
        if i == current_end:
            jumps += 1
            current_end = farthest

            # Gia possibile raggiungere la fine
            if current_end >= len(nums) - 1:
                break

    return jumps
```

Prova di Correttezza (can_jump):
Invariante: All'iterazione i, max_reach è la posizione più lontana raggiungibile da $0..i$.
Base: $i = 0$, $max\_reach = nums[0]$ (corretto).
Passo: Se $i \leq max\_reach$, allora i è raggiungibile. Da i possiamo saltare fino a $i + nums[i]$, quindi:

$$max\_reach = \max(max\_reach, i + nums[i])$$

Terminazione: Se alla fine $max\_reach \geq n - 1$, l'ultima posizione è raggiungibile.

B.5.2 Esercizio 8: Meeting Rooms II

Problema: Dato un insieme di intervalli di meeting, trova il minimo numero di sale conferenze richieste.
Esempio:

```
Input:  [[0,30],[5,10],[15,20]]
Output: 2
```

Soluzione:

```
import heapq

def min_meeting_rooms(intervals):
    """
    Minimo numero di sale conferenze.
```

```
6
7       Strategia: mantieni un min-heap dei tempi di fine.
8
9       Complessita: O(n log n)
10      """
11      if not intervals:
12          return 0
13
14      # Ordina per tempo di inizio
15      intervals.sort(key=lambda x: x[0])
16
17      # Min-heap per tempi di fine
18      rooms = []  # Contiene tempi di fine meeting correnti
19      heapq.heappush(rooms, intervals[0][1])
20
21      for i in range(1, len(intervals)):
22          start, end = intervals[i]
23
24          # Se il meeting piu vecchio e finito, riusa la sala
25          if start >= rooms[0]:
26              heapq.heappop(rooms)
27
28          # Aggiungi il nuovo meeting
29          heapq.heappush(rooms, end)
30
31      return len(rooms)
32
33
34  # Soluzione alternativa: eventi
35  def min_meeting_rooms_events(intervals):
36      """
37      Approccio con eventi di inizio/fine.
38
39      Complessita: O(n log n)
40      """
41      events = []
42
43      for start, end in intervals:
44          events.append((start, 1))    # Inizio meeting
45          events.append((end, -1))     # Fine meeting
46
47      # Ordina: se stesso tempo, fine prima di inizio
48      events.sort(key=lambda x: (x[0], x[1]))
49
50      max_rooms = 0
51      current_rooms = 0
52
53      for time, delta in events:
54          current_rooms += delta
55          max_rooms = max(max_rooms, current_rooms)
56
57      return max_rooms
```

B.6 Backtracking

B.6.1 Esercizio 9: Generate Parentheses

Problema: Genera tutte le combinazioni di n coppie di parentesi bilanciate.

Esempio:

```
Input:  n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
```

Soluzione:

```python
def generate_parentheses(n):
    """
    Genera tutte le parentesi bilanciate.

    Complessita: O(4^n / sqrt(n)) - n-esimo numero di Catalan
    """
    result = []

    def backtrack(current, open_count, close_count):
        # Caso base: generata una soluzione completa
        if len(current) == 2 * n:
            result.append(current)
            return

        # Aggiungi '(' se possibile
        if open_count < n:
            backtrack(current + '(', open_count + 1, close_count)

        # Aggiungi ')' se bilancia una '('
        if close_count < open_count:
            backtrack(current + ')', open_count, close_count + 1)

    backtrack('', 0, 0)
    return result
```

Invariante: In ogni momento, $close\_count \leq open\_count \leq n$.

B.6.2 Esercizio 10: Letter Combinations of Phone Number

Problema: Dato una stringa di cifre, ritorna tutte le possibili combinazioni di lettere (come tastiera telefono).

Esempio:

```
Input:  "23"
Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

Soluzione:

```python
def letter_combinations(digits):
    """
    Genera combinazioni di lettere da cifre.

    Complessita: O(4^n) worst case (n = len(digits))
    """
    if not digits:
        return []

```

```
    # Mappa cifre -> lettere
    phone = {
        '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
        '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
    }

    result = []

    def backtrack(index, current):
        # Caso base: combinazione completa
        if index == len(digits):
            result.append(current)
            return

        # Prova ogni lettera per la cifra corrente
        for letter in phone[digits[index]]:
            backtrack(index + 1, current + letter)

    backtrack(0, '')
    return result


# Versione iterativa
def letter_combinations_iterative(digits):
    """
    Versione iterativa con queue.
    """
    if not digits:
        return []

    phone = {
        '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
        '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
    }

    result = ['']

    for digit in digits:
        temp = []
        for combination in result:
            for letter in phone[digit]:
                temp.append(combination + letter)
        result = temp

    return result
```

B.7 Problemi Misti

B.7.1 Esercizio 11: Trapping Rain Water

Problema: Dato un array di altezze, calcola quanta acqua può essere intrappolata dopo la pioggia.

Esempio:

```
Input:  [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
```

Soluzione 1: DP ($O(n)$ tempo, $O(n)$ spazio)

```python
def trap_dp(height):
    """
    Calcola acqua intrappolata.

    Idea: per ogni posizione, acqua = min(max_left, max_right) - height

    Complessita: O(n) tempo, O(n) spazio
    """
    if not height:
        return 0

    n = len(height)

    # Calcola max a sinistra per ogni posizione
    left_max = [0] * n
    left_max[0] = height[0]
    for i in range(1, n):
        left_max[i] = max(left_max[i - 1], height[i])

    # Calcola max a destra per ogni posizione
    right_max = [0] * n
    right_max[n - 1] = height[n - 1]
    for i in range(n - 2, -1, -1):
        right_max[i] = max(right_max[i + 1], height[i])

    # Calcola acqua
    water = 0
    for i in range(n):
        water += min(left_max[i], right_max[i]) - height[i]

    return water
```

Soluzione 2: Two Pointers ($O(n)$ tempo, $O(1)$ spazio)

```python
def trap_two_pointers(height):
    """
    Two pointers ottimizzato.

    Complessita: O(n) tempo, O(1) spazio
    """
    if not height:
        return 0

    left, right = 0, len(height) - 1
    left_max, right_max = 0, 0
    water = 0

    while left < right:
        if height[left] < height[right]:
            if height[left] >= left_max:
                left_max = height[left]
            else:
                water += left_max - height[left]
            left += 1
        else:
            if height[right] >= right_max:
                right_max = height[right]
            else:
```

```
                water += right_max - height[right]
            right -= 1

    return water
```

Analisi Two Pointers:

- Manteniamo $left\_max$ e $right\_max$ per i lati sinistro e destro
- Spostiamo il puntatore dalla parte con altezza minore
- L'acqua in posizione i dipende solo dal minimo tra il max a sinistra e a destra
- Poiché processiamo dal lato più basso, sappiamo che il max dall'altro lato è almeno uguale

B.7.2 Esercizio 12: Longest Consecutive Sequence

Problema: Trova la lunghezza della più lunga sequenza consecutiva in un array non ordinato, in tempo $O(n)$.

Esempio:

```
Input:  [100, 4, 200, 1, 3, 2]
Output: 4 (sequenza [1, 2, 3, 4])
```

Soluzione:

```
def longest_consecutive(nums):
    """
    Trova lunghezza massima sequenza consecutiva.

    Strategia: usa set per O(1) lookup

    Complessita: O(n) tempo, O(n) spazio
    """
    if not nums:
        return 0

    num_set = set(nums)
    max_length = 0

    for num in num_set:
        # Inizia sequenza solo se num e l'inizio
        # (cioe num-1 non esiste)
        if num - 1 not in num_set:
            current_num = num
            current_length = 1

            # Estendi sequenza finche possibile
            while current_num + 1 in num_set:
                current_num += 1
                current_length += 1

            max_length = max(max_length, current_length)

    return max_length
```

Prova di $O(n)$: Anche se ci sono loop annidati, ogni numero viene visitato al massimo due volte:

- Una volta nel loop esterno

- Una volta nel loop interno (solo se è parte di una sequenza)

Quindi complessità totale è $O(n)$.

B.8 Tecniche di Analisi

B.8.1 Analisi Ammortizzata

Esempio: Array dinamico con raddoppio

```python
class DynamicArray:
    """
    Analisi ammortizzata: insert e O(1) ammortizzato
    anche se occasionalmente richiede O(n).
    """
    def __init__(self):
        self.capacity = 1
        self.size = 0
        self.array = [None] * self.capacity

    def insert(self, value):
        """
        Insert e O(1) ammortizzato.

        Dimostrazione:
        - Costo totale per n inserimenti:
          n + (2 + 4 + 8 + ... + n) < n + 2n = 3n
        - Costo ammortizzato per operazione: 3n/n = O(1)
        """
        if self.size == self.capacity:
            self._resize()

        self.array[self.size] = value
        self.size += 1

    def _resize(self):
        """Raddoppia capacita."""
        self.capacity *= 2
        new_array = [None] * self.capacity
        for i in range(self.size):
            new_array[i] = self.array[i]
        self.array = new_array
```

Questa appendice fornisce una base solida per comprendere e applicare gli algoritmi studiati. Ogni esercizio illustra tecniche chiave e pattern di soluzione comuni in informatica.