Università di Pisa
Corso di Laurea in Informatica

LA BIBBIA Di
PROGRAMMAZIONE ED ALGORITMICA

Appunti Completi
Informatica - Università di Pisa

Docenti titolari:
[Nome Docente]

Autore del riassunto:
Joseph Zucchelli

Anno Accademico 2024 - 2025

P&A

Indice

Parte I

Lezione 1(13/10/2025)

1 Definizione di Algoritmo

Un algoritmo è una sequenza finita di operazioni elementari (passi), univocamente determinata (non ambiguo), che, se eseguita su un calcolatore, porta alla risoluzione di un problema.

1.1 Modello RAM (Random Access Machine)

Nel modello RAM, si assume che le seguenti operazioni elementari abbiano costo "unitario" (costante):

- **Operazioni aritmetiche**: +, -, *, /, %
- **Operazioni di confronto**: $<, >, ==, ! =$
- **Operazioni logiche**: AND, OR, NOT
- **Operazioni di trasferimento**: load/store/assegnamento
- **Operazioni di controllo**: chiamata di funzione, RETURN

2 Analisi di Complessità

Si analizza il costo computazionale (Tempo o Spazio) in funzione della dimensione dell'input, n.

- **Complessità in Tempo** $T(n)$: Numero di operazioni elementari eseguite.
- **Complessità in Spazio** $S(n)$: Numero di celle di memoria utilizzate (oltre a quelle dell'input).

Ci si concentra sull' **ordine di grandezza** della funzione $T(n)$, ignorando costanti moltiplicative e termini di ordine inferiore. Ad esempio, $T(n) = 3n + 2$ e $T(n) = 5n + \log n + 4$ sono entrambe considerate di complessità **Lineare**. $T(n) = 8n^2$ è **Quadratica**.

2.1 Caso Ottimo, Pessimo, Medio

- **Caso Ottimo**: L'istanza di input che richiede il minor tempo.
- **Caso Pessimo**: L'istanza di input che richiede il maggior tempo.
- **Caso Medio**: Complessità media su tutte le possibili istanze.

Ci si concentra sul **caso pessimo** perché fornisce un limite superiore al costo: l'algoritmo non impiegherà mai più di $T(n)$.

3 Esempio 1: Minimo in Vettore

- **Input**: Array $A[1..n]$ di interi.
- **Output**: Il valore minimo contenuto in A.

Algoritmo

```
Procedure MINIMO(A, n)
    min = A[1]                              ▷ Costo costante c_1
    For i = 2 → n do                        ▷ Eseguito n − 1 volte
        If A[i] < min then                  ▷ Costo c_2
            min = A[i]                      ▷ Costo c_3
        end If
    end For
    Return min                              ▷ Costo costante c_4
end Procedure
```

Ricerca del Minimo

Algoritmo iterativo standard:

- Assume che il primo elemento sia il minimo provvisorio.
- Scorre il resto dell'array: se trova un elemento minore, aggiorna il minimo.
- Costo sempre lineare $\Theta(n)$.

Analisi: Il costo totale è $T(n) = c_1 + (n-1)(c_2$ (confronto)$+c_3$ (assegn. caso pessimo)$)+c_4$. $T(n) = c'n + b$. La complessità è **Lineare**, $T(n) \in \Theta(n)$, sia nel caso ottimo che in quello pessimo.

4 Esempio 2: Cerca K

- **Input**: Array $A[1..n]$ di interi, k intero.
- **Output**: i tale che $A[i] = k$, o -1 se $k \notin A$.

Algoritmo

```
Procedure CERCA-K(A, n, k)
    i = 1
    trovato = false
    While (not trovato) and (i ≤ n) do
        If A[i] == k then
            trovato = true
        Else
            i = i + 1
        end If
    end While
    If trovato then
        Return i
    Else
        Return −1
    end If
end Procedure
```

Ricerca Lineare (con K)

Simile alla ricerca del minimo, ma si ferma appena trova l'elemento k. Il caso ottimo è $O(1)$ (trovato subito), il pessimo è $O(n)$ (in fondo o assente).

Analisi:

- **Caso Ottimo**: $k = A[1]$. Il ciclo `while` esegue 1 iterazione. $T(n) \in \Theta(1)$ (Costante).
- **Caso Pessimo**: $k \notin A$ (o $k = A[n]$). Il ciclo `while` esegue n iterazioni. $T(n) \in \Theta(n)$ (Lineare).

5 Esempio 3: Minimo in Vettore Ordinato

- **Input**: Array $A[1..n]$ di interi, **ordinato**.
- **Output**: Il valore minimo contenuto in A.

Algoritmo

```
Procedure MINIMO-ORDINATO(A, n)
    Return A[1]
end Procedure
```

Analisi: $T(n) \in \Theta(1)$ (Costante).

6 Esempio 4: Cerca K in Vettore Ordinato (Ricerca Binaria)

- **Input**: Array $A[1..n]$ di interi **ordinato**, k intero.
- **Output**: i tale che $A[i] = k$, o -1 se $k \notin A$.

L'idea è di confrontare k con l'elemento centrale $A[q]$ e dimezzare lo spazio di ricerca.

Algoritmo

```
Procedure BS-IT(A, p, r, k)
    If (k < A[p]) or (k > A[r]) then          ▷ Controllo opzionale
        Return −1
    end If
    While p ≤ r do
        q = ⌊(p + r)/2⌋
        If A[q] == k then
            Return q
        Else If A[q] > k then
            r = q − 1
        Else
            p = q + 1
        end If
    end While
    Return −1
end Procedure
```

Ricerca Binaria Iterativa

Versione non ricorsiva della Binary Search. Usa un ciclo `while` per aggiornare gli estremi p e r. È più efficiente in termini di spazio (niente stack ricorsivo) ma identica come complessità temporale $O(\log n)$.

Analisi:

- **Caso Ottimo**: $k = A[q]$ al primo ciclo. $T(n) \in \Theta(1)$ (Costante).
- **Caso Pessimo**: $k \notin A$. Il numero di iterazioni è $\log_2 n$. $T(n) \in \Theta(\log n)$ (Logaritmica).

Precisazione

Guida alla Complessità Computazionale (Notazione O-Grande) La complessità computazionale è un modo per descrivere l'efficienza di un algoritmo. Non misura il tempo esatto in secondi, ma stima come il numero di operazioni (tempo) o l'uso della memoria (spazio) cresce all'aumentare della dimensione dell'input (indicato con n). La notazione O-grande si concentra sull'ordine di grandezza asintotico, ignorando le costanti moltiplicative e i termini di ordine inferiore.

Ordine Asintotico

L'ordine asintotico descrive come si comporta il tempo (o lo spazio) richiesto da un algoritmo quando la dimensione dell'input (n) diventa estremamente grande. È come guardare la "forma" generale della curva di crescita da molto lontano.
Si ignorano i dettagli iniziali e la ripidità iniziale della curva, per concentrarci unicamente sul termine che cresce più velocemente, visto che sarà il più impattante quando n sarà enorme. Ad esempio, se un algoritmo impiega $3n^2 + 10n + 5$ operazioni, il suo ordine asintotico è $O(n^2)$.
Perché? Perché quando n diventa grandissimo (es. un milione), il termine n^2 è talmente più grande di n e di 5 che gli altri diventano irrilevanti per capire l'andamento generale.

7 Differenza Chiave: $O(n)$ (Lineare) vs. $O(\log n)$ (Logaritmico)

Spesso si crea confusione tra un costo come $n/2$ e uno come $\log n$, ma appartengono a due universi di efficienza completamente diversi.

- **Lineare $O(n)$:** Un algoritmo con un costo proporzionale a n (come n, $n/2$ o $2n$) ha una complessità Lineare, $O(n)$. Questo significa che il numero di operazioni è direttamente proporzionale alla dimensione dell'input. Se l'input raddoppia, anche il tempo di esecuzione (circa) raddoppia. Nella notazione O-grande, le costanti (come $1/2$) vengono ignorate. Ad esempio, la ricerca lineare in un array non ordinato richiede, nel caso medio, $n/2$ controlli. La sua complessità è comunque $O(n)$.
- **Logaritmico $O(\log n)$:** Un algoritmo con costo $\log n$ (logaritmo in base 2, $\log_2 n$) ha una complessità Logaritmica, $O(\log n)$. Questo tipo di algoritmo è estremamente efficiente perché, ad ogni passo, è in grado di scartare una frazione significativa del problema (di solito la metà). L'esempio classico è la ricerca binaria.

Confronto Pratico: $O(n)$ vs $O(\log n)$

Su un input di $n = 1.000.000$ di elementi:

- Un algoritmo lineare ($n/2$) richiederebbe circa **500.000** operazioni.
- Un algoritmo logaritmico ($\log_2 n$) ne richiederebbe circa **20**.

$O(\log n)$ è drasticamente più veloce di $O(n)$.

8 Caso Pessimo, Medio e Ottimo

La performance di un algoritmo può cambiare non solo in base alla dimensione dell'input (n), ma anche in base a come è fatto l'input.

Caso Pessimo, Medio e Ottimo

- **Caso Pessimo (Worst Case):** Rappresenta lo scenario che richiede il massimo numero di operazioni. È l'input "peggiore" possibile. È la metrica più importante e quasi sempre quella che si utilizza, perché fornisce una garanzia sulla performance: l'algoritmo non farà mai peggio di così.
- **Caso Medio (Average Case):** Descrive la performance "tipica" calcolata come media su tutti i possibili input.
- **Caso Ottimo (Best Case):** Descrive lo scenario più veloce in assoluto (ma spesso poco utile, perché si verifica solo in condizioni molto specifiche).

9 Classi di Complessità (dal più veloce al più lento)

Gerarchia delle Complessità

$O(1)$ **- Costante:** Il tempo non dipende da n. (Es. Accesso a un array `array[i]`).

$O(\log n)$ **- Logaritmico:** Il tempo cresce molto lentamente. (Es. Ricerca binaria).

$O(n)$ **- Lineare:** Il tempo cresce linearmente con n. (Es. Un singolo ciclo for, trovare il massimo).

$O(n \log n)$ **- Linearitmico:** Ottima complessità per gli algoritmi di ordinamento. (Es. Merge Sort, Heapsort).

$O(n^2)$ **- Quadratico:** Il tempo cresce con il quadrato di n. (Es. Due cicli for annidati, Bubble Sort, Insertion Sort).

$O(n^k)$ **- Polinomiale:** Il tempo cresce con n elevato a una costante k. (Es. Tre cicli annidati $O(n^3)$).

$O(2^n)$ **- Esponenziale:** Diventa intrattabile molto rapidamente. (Es. Soluzioni "brute force" che provano tutte le combinazioni).

$O(n!)$ **- Fattoriale:** Il peggior caso possibile. (Es. "Brute force" al problema del commesso viaggiatore).

10 Regole di Calcolo (Come Combinare)

Per calcolare la complessità di un programma intero, si combinano i costi delle sue parti usando due regole fondamentali.

Regole di Calcolo Asintotico

- **Regola della Somma (Operazioni in sequenza):** Se hai un blocco A seguito da un blocco B, la complessità totale è $O(A)+O(B)$. Si tiene solo il termine dominante. Ad esempio, un ciclo $O(n)$ seguito da due cicli annidati $O(n^2)$ ha una complessità totale $O(n) + O(n^2)$, che si semplifica in $O(n^2)$.
- **Regola del Prodotto (Operazioni annidate):** Se un blocco B è all'interno di un blocco A, le complessità si moltiplicano: $O(A) \times O(B)$. L'esempio classico è un `for` $O(n)$ che contiene un altro `for` $O(n)$: la complessità totale è $O(n \times n) = O(n^2)$.

11 Impatto dell'Ordinamento: Array Ordinato vs. Non Ordinato

Avere un array di input già ordinato (o decidere di ordinarlo) può cambiare drasticamente la complessità. L'operazione di ordinamento in sé ha un costo, tipicamente $O(n \log n)$.

Ricerca di un elemento: $O(n)$ vs $O(\log n)$

- **Non Ordinato:** Ricerca lineare (controllarli uno per uno). Caso pessimo: $O(n)$.
- **Ordinato:** Ricerca binaria (dimezzando l'intervallo). Caso pessimo: $O(\log n)$.

Ricerca di duplicati: $O(n^2)$ vs $O(n)$

- **Non Ordinato:** Confrontare ogni elemento con ogni altro. Caso pessimo: $O(n^2)$.
- **Ordinato:** Basta una singola scansione lineare. Se $A[i] == A[i+1]$, esiste un duplicato. Caso pessimo: $O(n)$.

Trovare il minimo o il massimo: $O(n)$ vs $O(1)$

- **Non Ordinato:** Bisogna scorrere tutto l'array. Caso pessimo: $O(n)$.
- **Ordinato:** Il minimo è il primo elemento ($A[0]$) e il massimo è l'ultimo ($A[n-1]$). Caso pessimo: $O(1)$.

11.1 Quando ha senso ordinare l'array?

Ordinare (costo $O(n \log n)$) ha senso quando il costo totale è inferiore a quello dell'operazione sull'array non ordinato.

Scenario 1: Operazione singola (Trova max)

- **Costo (Non ordinato):** $O(n)$.
- **Costo (Ordinando prima):** $O(n \log n)$ (sort) $+ O(1)$ (accesso) $= O(n \log n)$.
- **Verdetto:** $O(n)$ è molto meglio. Non ordinare.

Scenario 2: Operazioni multiple (k ricerche)

Se devi effettuare k ricerche diverse su n elementi.

- **Costo (Non ordinato):** k ricerche lineari $\implies k \times O(n) = O(k \cdot n)$.
- **Costo (Ordinando prima):** $O(n \log n)$ (una tantum)$+k \times O(\log n) \implies O(n \log n + k \log n)$.
- **Verdetto:** Se k è grande (es. $k \approx O(n)$), il costo $O(n \log n)$ è drasticamente migliore di $O(n^2)$. Ha senso ordinare.

12 Complessità Spaziale (Cenno)

Oltre al tempo, la complessità spaziale misura quanta memoria ausiliaria (spazio extra oltre all'input) usa l'algoritmo. Può essere $O(1)$ (costante), se usa solo un numero fisso di variabili, o $O(n)$ (lineare), se ha bisogno di creare una struttura dati (come un array di supporto) grande quanto l'input.

13 Esempi di Analisi (Linguaggio MAO)

Analizziamo il costo (caso pessimo) di alcuni frammenti di codice MAO.

Esempio 1: Ciclo Singolo (Lineare)

```
int i=0;
int s=0;
while (i < n) {
s:= s + i;
i:= i + 1;
}
```

Analisi ($O(n)$): Il codice esegue due comandi iniziali $O(1)$. Segue un ciclo `while`. Il corpo del ciclo ha costo costante $O(1)$. La guardia viene valutata $n+1$ volte e il corpo viene eseguito n volte. La complessità totale è (Regola della Somma): $O(1) + O(n \times 1)$. Il termine dominante è $O(n)$.

Esempio 2: Cicli Annidati (Quadratico)

```
int i=0;
int r=0;
while (i < n) {
int j=0;
while (j < n) {
r:= r + 1;
j:= j + 1;
}
i:= i + 1;
}
```

Analisi ($O(n^2)$): Abbiamo due cicli annidati. Il ciclo esterno (while $i < n$) esegue il suo corpo n volte. Il corpo contiene un ciclo interno (while $j < n$) che viene eseguito n volte per ogni iterazione esterna. Per la Regola del Prodotto, la complessità è $O(n \times n) = O(n^2)$.

Esempio 3: Sequenza di Cicli (Regola della Somma)

```
int s=0;
int i=0;
while (i < n) {
s:= s + i;
i:= i + 1;
}
int j=0;
while (j < n) {
int k=0;
while (k < n) {
s:= s + 1;
k:= k + 1;
}
j:= j + 1;
}
```

Analisi ($O(n^2)$): Questo codice è una sequenza di due blocchi.

- Il primo blocco è un ciclo singolo: costo $O(n)$.
- Il secondo blocco è composto da due cicli annidati: costo $O(n^2)$.

Per la Regola della Somma, la complessità totale è $O(n) + O(n^2)$. Si considera solo il termine dominante, quindi la complessità è $O(n^2)$.

Esempio 4: Ciclo Logaritmico

```
int i=1;
while (i < n) {
skip;
i:= i * 2;
}
```

Analisi ($O(\log n)$): La variabile di controllo i non viene incrementata linearmente $(i + 1)$, ma viene moltiplicata per 2. I valori di i saranno $1, 2, 4, 8, 16, ..., 2^k$ fino a superare n. Il numero di iterazioni (k) è il più piccolo intero tale che $2^k \geq n$. Questo k è esattamente $\log_2 n$. Poiché il corpo del ciclo ha costo $O(1)$, la complessità totale è $O(\log n)$.

Esempio 5: Condizionale nel Caso Pessimo

```
int i=0;
int r=0;
while (i < n) {
if (i < 10) {
r:= r + 1;
// Costo O(1)
} else {
int j=0;
while (j < n) {
r:= r + j;
// Costo O(n)
j:= j + 1;
}
}
i:= i + 1;
}
```

Analisi ($O(n^2)$): Stiamo analizzando il caso pessimo. Il ciclo esterno (while $i < n$) viene eseguito n volte. All'interno c'è un `if`. Dobbiamo considerare il costo del ramo più pesante.

- Il ramo `if` $(i < 10)$ ha costo $O(1)$.
- Il ramo `else` contiene un ciclo lineare, costo $O(n)$.

Nell'analisi del caso pessimo, assumiamo che venga sempre eseguito il ramo più costoso. Quasi tutte le iterazioni (per $i \geq 10$) eseguiranno il ramo `else`, che costa $O(n)$. Applicando la Regola del Prodotto, abbiamo il ciclo esterno $O(n)$ che contiene un blocco che (nel caso peggiore) costa $O(n)$. La complessità totale è $O(n \times n) = O(n^2)$.

Parte II

Lezione 2 (16/10/2025)

14 Selection Sort (Analisi)

Pseudocodice (identico a Lez12).

Algoritmo

```
Procedure SelectionSort(A)
    For i = 1 → n − 1 do
        min = i
        For j = i + 1 → n do                ▷ Il loop interno fa (n − i) iterazioni
            If A[j] < A[min] then
                min = j
            end If
        end For
        Swap(A[i], A[min])
    end For
end Procedure
```

Selection Sort

L'algoritmo seleziona iterativamente il minimo dalla parte non ordinata e lo sposta alla fine della parte ordinata.

- **Ciclo Esterno**: Avanza il confine tra ordinato e non ordinato.
- **Ciclo Interno**: Cerca il minimo nel sottoarray destro.
- **Swap**: Scambia il minimo trovato con l'elemento corrente.

14.1 Analisi Complessità (Numero Confronti)

Il costo è dominato dal numero di confronti ($A[j] < A[min]$). Il ciclo esterno `for i` esegue $n-1$ iterazioni. Il ciclo interno `for j` esegue $n - i$ iterazioni per ogni i. Il numero totale di confronti $C(n)$ è:

$$C(n) = \sum_{i=1}^{n-1} (n - i)$$

$$C(n) = (n - 1) + (n - 2) + \cdots + 2 + 1$$

Questa è la somma dei primi $n - 1$ numeri naturali.

$$C(n) = \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

La complessità è **Quadratica**, $T(n) \in \Theta(n^2)$.

Osservazione

A differenza di Insertion Sort, la complessità di Selection Sort è $\Theta(n^2)$ *sempre*, sia nel caso ottimo, medio e pessimo, perché i cicli `for` vengono eseguiti sempre lo stesso numero di volte.

14.2 Invariante di Ciclo

Invariante: Selection Sort

Invariante: All'inizio dell'iterazione i-esima del ciclo FOR esterno (per $i = 1..n-1$):

1. Il sottoarray $A[1..i-1]$ contiene gli $i-1$ elementi più piccoli di A.
2. Il sottoarray $A[1..i-1]$ è ordinato.

(Si dimostra per induzione).

15 Esercizi

Esercizio 1: Cerca A[i

= i (Array non ordinato)]

- **Input**: Array $A[1..n]$ di interi.
- **Output**: TRUE se $\exists i$ t.c. $A[i] = i$, FALSE altrimenti.

Algoritmo

```
Procedure CERCA-INDICE(A, n)
    i = 1
    trovato = false
    While (not trovato) and (i ≤ n) do
        If A[i] == i then
            trovato = true
        Else
            i = i + 1
        end If
    end While
    Return trovato
end Procedure
```

Ricerca Lineare

Scansiona l'array elemento per elemento. Se trova $A[i] == i$, si ferma e ritorna TRUE. Nel caso pessimo (non trovato), scorre tutto l'array (n passi).

Analisi:

- Caso Ottimo: $A[1] = 1$. $T(n) \in \Theta(1)$.
- Caso Pessimo: Nessun i t.c. $A[i] = i$. $T(n) \in \Theta(n)$ (Lineare).

Esercizio 2: Cerca A[i

= i (Array ordinato)]

- **Input**: Array $A[1..n]$ di interi, **ordinato**.
- **Output**: TRUE se $\exists i$ t.c. $A[i] = i$.

Si può usare una modifica della Ricerca Binaria. Si calcola $q = \lfloor (p + r)/2 \rfloor$.

- Se $A[q] == q$: Trovato.
- Se $A[q] > q$: L'elemento i (se esiste) non può essere a destra di q. Si cerca a sinistra ($r = q - 1$).
- Se $A[q] < q$: L'elemento i (se esiste) non può essere a sinistra di q. Si cerca a destra ($p = q + 1$).

Analisi: $T(n) \in O(\log n)$.

Esercizio 3: Cerca A[i

= i (Ordinato, positivi, distinti)]

- **Input**: Array $A[1..n]$ ordinato, di interi **positivi** e **distinti**.
- **Output**: TRUE se $\exists i$ t.c. $A[i] = i$.

Se $A[1] = 1$: Ritorna TRUE. Se $A[1] > 1$: (cioè $A[1] \geq 2$). Allora $A[i] \geq A[1] + (i - 1) \geq 2 + i - 1 = i + 1$. Quindi $A[i] > i$ per ogni i. Ritorna FALSE. L'algoritmo corretto è:

Algoritmo

```
Procedure CERCA-I-POSITIVI(A)
    Return (A[1] == 1)
end Procedure
```

Analisi: $T(n) \in \Theta(1)$ (Costante).

Esercizio 4: Somma K

- **Input**: Array $A[1..n]$ di interi, k intero.
- **Output**: TRUE se $\exists i, j$ t.c. $A[i] + A[j] = k$.

Soluzione 1 (Brute force):

Algoritmo

```
For i = 1 → n − 1 do
    For j = i + 1 → n do
        If A[i] + A[j] == k then
            Return true
        end If
    end For
end For
Return false
```

Analisi 1: Caso pessimo $\Theta(n^2)$ (Quadratico). **Soluzione 2 (se A è ordinato):** Si usano due indici, $L = 1$ e $R = n$.

Algoritmo

```
L = 1, R = n
While L < R do
    somma = A[L] + A[R]
    If somma == k then
        Return true
    Else If somma < k then
        L = L + 1                ▷ Serve una somma più grande
    Else
        R = R − 1                ▷ Serve una somma più piccola
    end If
end While
Return false
```

Tecnica dei Due Indici

Poiché l'array è ordinato, possiamo restringere la ricerca:

- $Somma < K \to$ Incremento L (serve valore più grande).
- $Somma > K \to$ Decremento R (serve valore più piccolo).

Questo riduce la complessità da quadratica a lineare.

Analisi 2: $T(n) \in \Theta(n)$ (Lineare).

Esercizio 5: Array Palindromo

- **Input**: Array $A[1..n]$.
- **Output**: TRUE se A è palindromo, FALSE altrimenti. (E.g., `[3, 7, 21, 40, 21, 7, 3]`).

Soluzione (con due indici):

`Algoritmo`

```
i = 1, j = n
While i < j do
    If A[i] ≠ A[j] then
        Return false
    end If
    i = i + 1
    j = j − 1
end While
Return true
```

Verifica Palindromo

Confronta gli estremi convergendo verso il centro. Se una coppia non corrisponde, non è palindromo.

Analisi: $T(n) \in \Theta(n)$.

Parte III

Lezione 14 (20/10/2025)

16 Notazione Asintotica

La complessità $T(n)$ si esprime in **ordine di grandezza**, ignorando costanti moltiplicative e termini di ordine inferiore.

- $T(n) = 3n^2 + 2n + 5 \rightarrow$ Quadratica ($\Theta(n^2)$)
- $T(n) = 7n + 24 \rightarrow$ Lineare ($\Theta(n)$)
- $T(n) = 5 \rightarrow$ Costante ($\Theta(1)$)
- $T(n) = \log_3 n + 2 \rightarrow$ Logaritmica ($\Theta(\log n)$)

Si usano funzioni di riferimento semplici $g(n)$ (es. n^2, n, $\log n$) per classificare $f(n) = T(n)$.

17 Notazione Θ (Theta) - Limite Stretto

Notazione Θ (Theta)

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0 : \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Si dice "$f(n)$ è in Theta di $g(n)$". $g(n)$ è un **limite asintotico stretto** per $f(n)$. Graficamente, da n_0 in poi, $f(n)$ è "intrappolata" tra $c_1 g(n)$ e $c_2 g(n)$.

- Esempio: Selection Sort è $\Theta(n^2)$.
- Esempio: $\frac{1}{2}n^2 - 2n \in \Theta(n^2)$.

18 Notazione O (O-grande) - Limite Superiore

Notazione O (O-grande)

$$O(g(n)) = \{f(n) \mid \exists c, n_0 > 0 : \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$

Si dice "$f(n)$ è in O-grande di $g(n)$". $g(n)$ è un **limite asintotico superiore** per $f(n)$. Graficamente, da n_0 in poi, $f(n)$ non cresce più velocemente di $cg(n)$.

- Esempio: $f(n) = an^2 + bn + c \in O(n^2)$.
- Esempio: $f(n) = an^2 + bn + c \in O(n^3)$.
- Esempio: $f(n) = an^2 + bn + c \notin O(n)$.

Proprietà: $f(n) \in \Theta(g(n)) \implies f(n) \in O(g(n))$.

19 Notazione Ω (Omega) - Limite Inferiore

Notazione Ω (Omega)

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_0 > 0 : \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$

Si dice "$f(n)$ è in Omega di $g(n)$". $g(n)$ è un **limite asintotico inferiore** per $f(n)$. Graficamente, da n_0 in poi, $f(n)$ non cresce più lentamente di $cg(n)$.

- Esempio: $an^2 + bn + c \in \Omega(n^2)$.
- Esempio: $an^2 + bn + c \in \Omega(n)$.
- Esempio: $an^2 + bn + c \notin \Omega(n^3)$.

Proprietà: $f(n) \in \Theta(g(n)) \implies f(n) \in \Omega(g(n))$.

19.1 Teorema

Teorema

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \text{ e } f(n) \in \Omega(g(n))$$

20 Proprietà e Gerarchia

Proprietà della Notazione Asintotica

- **Riflessività**: $f(n) \in \Theta(f(n))$, $f(n) \in O(f(n))$, $f(n) \in \Omega(f(n))$.
- **Simmetria (Theta)**: $f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$.
- **Trasposta (O/Omega)**: $f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$.
- **Transitività**: Vale per O, Ω, Θ. Es: $f_1 \in O(f_2)$ e $f_2 \in O(f_3) \implies f_1 \in O(f_3)$.
- **Somma**: $f_1 \in O(g_1)$ e $f_2 \in O(g_2) \implies f_1 + f_2 \in O(\max(g_1, g_2))$.
- **Prodotto**: $f_1 \in O(g_1)$ e $f_2 \in O(g_2) \implies f_1 \cdot f_2 \in O(g_1 \cdot g_2)$.

Equivalenza dei Logaritmi

Tutte le basi dei logaritmi sono asintoticamente equivalenti. Dalla formula del cambio di base: $\log_a n = \frac{\log_b n}{\log_b a}$. Poiché $\frac{1}{\log_b a}$ è una costante, si ha $\Theta(\log_a n) = \Theta(\log_b n)$. Per questo motivo, si scrive genericamente $O(\log n)$.

20.1 Gerarchia degli ordini di grandezza

Gerarchia di Crescita

Per $0 < h \leq k$ and $1 < a < b$:

$$\Theta(1) \subset \cdots \subset \Theta(\log n) \subset \cdots \subset \Theta(n^h) \subset \Theta(n^k) \subset \Theta(n^k \log n) \subset \cdots \subset \Theta(a^n) \subset \Theta(b^n) \subset \ldots$$

Ordinando le funzioni in per ordine crescente: 1 (costante), 4^5 (costante), $\log n$, $\log^2 n$, $n^{1/2}$ (o $\sqrt{n}$), n, $n \log n$, $n^4 - 7n^3 (\sim n^4)$, $n^5 - 5n^2 (\sim n^5)$, 2^n, 3^n.

Parte IV

Lezione 21 (06/11/2025)

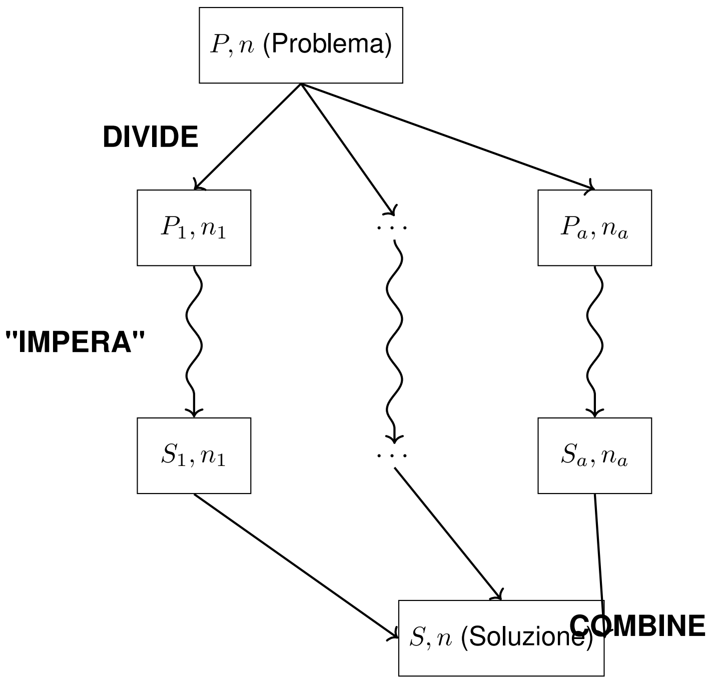
21 Paradigma Divide et Impera

Il paradigma "Divide et Impera" (Dividi e Conquista) è una tecnica per progettare algoritmi, tipicamente ricorsivi, che si articola in tre fasi:

Paradigma Divide et Impera

1. **DIVIDE**: Il problema di dimensione n viene suddiviso in a sottoproblemi dello stesso tipo, ma di dimensione minore (n/b).
2. **IMPERA**: I sottoproblemi vengono risolti. Se sono abbastanza piccoli (casi base), vengono risolti direttamente. Altrimenti, vengono risolti ricorsivamente con la stessa tecnica.
3. **COMBINE**: Le soluzioni degli a sottoproblemi vengono combinate per ottenere la soluzione del problema originale.

21.1 Diagramma Concettuale



22 Analisi Complessità D&I

Analisi Complessità D&I

Sia $T(n)$ il costo per risolvere un problema di dimensione n. Sia $D(n)$ il costo della fase DIVIDE. Sia $C(n)$ il costo della fase COMBINE. L'equazione di ricorrenza generale è:

$$T(n) = \sum_{i=1}^{a} T(n_i) + D(n) + C(n)$$

Caso particolare: **Divisione Bilanciata**. Il problema è diviso in a sottoproblemi, ognuno di dimensione n/b. Sia $f(n) = D(n) + C(n)$ il costo di divide e combine.

$$T(n) = aT(n/b) + f(n)$$

23 Esempio: Ricerca Binaria (D&I)

Ricerca Binaria: Setup

- **Input**: $A[p..r]$ ordinato, chiave k.
- **Output**: Indice i t.c. $A[i] = k$, o -1.

Algoritmo

```
Procedure BinarySearch(A, p, r, k)
    If p > r then                                  ▷ Caso Base 1: array vuoto
        Return −1
    end If
    q = ⌊(p + r)/2⌋                                ▷ DIVIDE
    If A[q] == k then                              ▷ IMPERA (Caso Base 2)
        Return q
    Else If A[q] > k then                          ▷ IMPERA (Ricorsione)
        Return BinarySearch(A, p, q-1, k)
    Else
        Return BinarySearch(A, q+1, r, k)
    end If
                                  ▷ COMBINE: non necessario, costo Θ(1)
end Procedure
```

Ricerca Binaria

Ad ogni passo dimezza lo spazio di ricerca ($n \to n/2 \to n/4 \ldots$).

- Se $A[q] > k$, cerco a sinistra.
- Se $A[q] < k$, cerco a destra.
- Costo logaritmico $\Theta(\log n)$.

Analisi: Ricerca Binaria

Analisi Ricorrenza BS: C'è $a = 1$ sottoproblema di dimensione $n/b = n/2$. $f(n) = D(n) + C(n) = \Theta(1) + \Theta(1) = \Theta(1)$.

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(n/2) + \Theta(1) & \text{se } n > 1 \end{cases}$$

Soluzione (Metodo Iterativo): $T(n) = T(n/2) + c\, T(n) = (T(n/4) + c) + c = T(n/4) + 2c$ $T(n) = (T(n/8) + c) + 2c = T(n/8) + 3c$ ... dopo i passi... $T(n) = T(n/2^i) + i \cdot c$ Ci si ferma al caso base quando $n/2^i = 1 \implies i = \log_2 n$. $T(n) = T(1) + c \cdot \log_2 n = \Theta(1) + \Theta(\log n) = \Theta(\log n)$.

24 Esempio: Minimo/Massimo (D&I)

Minimo/Massimo: Setup

- **Input**: $A[1..n]$.
- **Output**: Coppia $\langle min, max \rangle$ di A.

Algoritmo

```
Procedure MinMax(A, p, r)
    If r − p ≤ 1 then                                ▷ Caso Base: 1 o 2 elementi
        If A[p] ≤ A[r] then
            Return ⟨A[p], A[r]⟩
        Else
            Return ⟨A[r], A[p]⟩
        end If
    Else
        q = ⌊(p + r)/2⌋                              ▷ DIVIDE
        ⟨min₁, max₁⟩ = MinMax(A, p, q)               ▷ IMPERA
        ⟨min₂, max₂⟩ = MinMax(A, q + 1, r)           ▷ IMPERA
        min = min(min₁, min₂)                        ▷ COMBINE
        max = max(max₁, max₂)
        Return ⟨min, max⟩
    end If
end Procedure
```

Minimo e Massimo Simultanei

Per trovare min e max con meno confronti ($3\lfloor n/2 \rfloor$ invece di $2n$):

- Divide l'array in due metà.
- Risolve ricorsivamente.
- Combina confrontando i minimi tra loro e i massimi tra loro.

Analisi: Minimo/Massimo

Analisi Ricorrenza MinMax: $a = 2$ sottoproblemi, $n/b = n/2$. $f(n) = D(n)$ (cost.) $+$ $C(n)$ (2 confr.) $= \Theta(1)$.

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 2 \\ 2T(n/2) + \Theta(1) & \text{se } n \geq 3 \end{cases}$$

(Questa ricorrenza si risolve in $T(n) = \Theta(n)$).

Parte V

Lezione 22 (10/11/2025)

25 Mergesort

Mergesort è un algoritmo di ordinamento basato su Divide et Impera.

Mergesort: Paradigma D&I

- **DIVIDE**: Divide l'array $A[p..r]$ in due metà, $A[p..q]$ e $A[q+1..r]$, dove $q = \lfloor (p+r)/2 \rfloor$.
- **IMPERA**: Ordina ricorsivamente le due metà chiamando `Mergesort(A, p, q)` e `Mergesort(A, q+1, r)`.
- **COMBINE**: Combina (fonde) i due sottoarray ordinati $A[p..q]$ e $A[q+1..r]$ in un unico array ordinato $A[p..r]$ tramite la procedura `Merge(A, p, q, r)`.

25.1 Pseudocodice Mergesort

Algoritmo

Procedure MERGESORT(A, p, r)
    If $p < r$ **then**
        $q = \lfloor (p+r)/2 \rfloor$ ▷ DIVIDE
        MERGESORT(A, p, q) ▷ IMPERA
        MERGESORT(A, q+1, r) ▷ IMPERA
        MERGE(A, p, q, r) ▷ COMBINE
    end If
end Procedure

Logica Mergesort

Divide et Impera puro:

1. **Divide**: Calcola il punto medio q.
2. **Impera**: Due chiamate ricorsive su metà array.
3. **Combine**: La procedura `Merge` unisce i risultati.

25.2 Procedura Merge

Procedura Merge

La procedura `Merge` fonde due sottoarray contigui $A[p..q]$ e $A[q+1..r]$, che si assumono **già ordinati**. Ha complessità **Lineare** $T(n) = \Theta(n)$, dove $n = r - p + 1$. Utilizza due array di appoggio, L e R, e due "sentinelle" (∞) per evitare controlli sull'indice.

Algoritmo

```
Procedure MERGE(A, p, q, r)
    n1 = q − p + 1                                  ▷ Dim. primo sottoarray
    n2 = r − q                                      ▷ Dim. secondo sottoarray
    Crea array L[1..n1 + 1] e R[1..n2 + 1]
                                                    ▷ Copia i dati negli array di appoggio
    For i = 1 → n1 do
        L[i] = A[p + i − 1]
    end For
    For j = 1 → n2 do
        R[j] = A[q + j]
    end For
    L[n1 + 1] = +∞                                  ▷ Sentinella
    R[n2 + 1] = +∞                                  ▷ Sentinella
    i = 1                                           ▷ Indice per L
    j = 1                                           ▷ Indice per R
                                                    ▷ Fondi L e R nell'array A
    For k = p → r do
        If L[i] ≤ R[j] then
            A[k] = L[i]
            i = i + 1
        Else
            A[k] = R[j]
            j = j + 1
        end If
    end For
end Procedure
```

Procedura Merge

Fonde due sotto-array ordinati in uno solo.

- Usa due array di appoggio L e R.
- Le "sentinelle" (∞) evitano di dover controllare se un array è finito mentre si confrontano gli elementi.
- Complessità $\Theta(n)$.

25.3 Analisi Complessità Mergesort

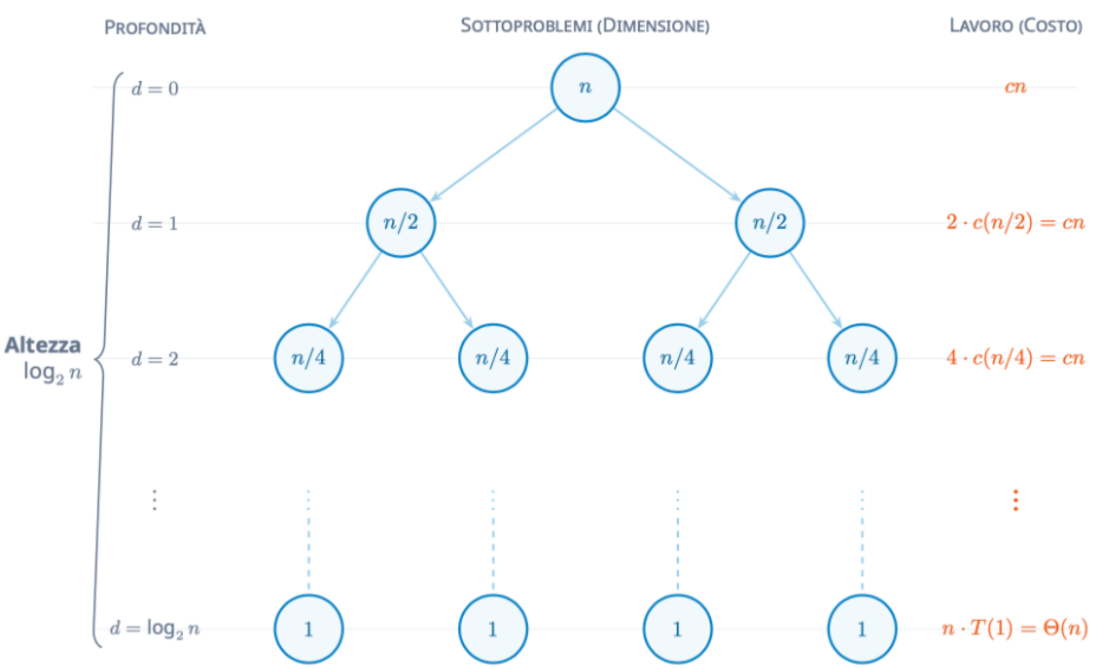
Analisi Mergesort: Ricorrenza

Equazione di Ricorrenza: $a = 2$ sottoproblemi, $n/b = n/2$. $f(n) = D(n)$ (cost.) $+ C(n)$ (Merge) $= \Theta(1) + \Theta(n) = \Theta(n)$.

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases}$$

Soluzione 1: Albero di Ricorsione L'albero di ricorsione mostra il costo $f(n_i)$ ad ogni livel-

Risoluzione della ricorrenza: $T(n) = 2T(n/2) + cn$



lo.

Il costo per ogni livello è cn. L'albero ha $\log_2 n$ livelli. Il costo totale è $cn \cdot \log_2 n = \Theta(n \log n)$.

Analisi Mergesort: Metodo Iterativo

Soluzione 2: Metodo Iterativo (Sostituzione) $T(n) = 2T(n/2)+cn$ $T(n) = 2(2T(n/4)+c(n/2)) + cn = 4T(n/4) + cn + cn = 4T(n/4) + 2cn$ $T(n) = 4(2T(n/8) + c(n/4)) + 2cn = 8T(n/8) + cn + 2cn = 8T(n/8) + 3cn$ ... dopo i passi... $T(n) = 2^i T(n/2^i) + i \cdot cn$ Ci si ferma al caso base $n/2^i = 1 \implies i = \log_2 n$. $T(n) = 2^{\log_2 n} T(1) + (\log_2 n) \cdot cn$ $T(n) = n \cdot \Theta(1) + cn \log_2 n = \Theta(n \log n)$.

Complessità in Spazio: Mergesort

Mergesort **non** ordina "in loco", poiché richiede $\Theta(n)$ spazio ausiliario per gli array L e R ad ogni chiamata di `Merge`.

25.4 Esempio: Albero delle Chiamate

Per $A[1..7]$, l'ordine delle chiamate ricorsive è:

$MS(1,7)$, $n = 7$
- $MS(1,4)$, $n = 4$
  - $MS(1,2)$, $n = 2$
    - $MS(1,1)$, $n = 1$
    - $MS(2,2)$, $n = 1$
  - $MS(3,4)$, $n = 2$
    - $MS(3,3)$, $n = 1$
    - $MS(4,4)$, $n = 1$
- $MS(5,7)$, $n = 3$
  - $MS(5,6)$, $n = 2$
    - $MS(5,5)$, $n = 1$
    - $MS(6,6)$, $n = 1$
  - $MS(7,7)$, $n = 1$

Spiegazione della Lezione 23 (12/10/2025)

Introduzione: Relazioni di Ricorrenza

Questi appunti della Lezione 23 affrontano un argomento cruciale nell'analisi degli algoritmi: le **Relazioni di Ricorrenza.** In breve, queste sono equazioni matematiche usate per descrivere il tempo di esecuzione, $T(n)$, di un algoritmo che chiama sé stesso (cioè un algoritmo ricorsivo).

Tipi di Relazioni di Ricorrenza

Gli appunti ne identificano tre tipi principali.

Relazioni Bilanciate (Divide et Impera)

Sono le più comuni negli algoritmi "Divide et Impera" (come Mergesort). Hanno una forma specifica:

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

- a è il numero di sotto-problemi in cui dividiamo il problema principale.
- n/b è la dimensione di ciascun sotto-problema.
- $f(n)$ è chiamata la "forzante" e rappresenta il lavoro "extra" fatto per dividere e ricombinare i risultati.

2. **Relazioni di Ordine K:** Queste dipendono dai valori immediatamente precedenti, come $T(n-1)$, $T(n-2)$, ecc. (es. Fibonacci).
3. **Caso Generale:** Una forma più complessa dove i sotto-problemi potrebbero non avere dimensioni uguali.

Esempi Concreti di Relazioni Bilanciate

Mergesort

Per ordinare un array, lo divide in 2 metà ($a = 2$), le ordina ricorsivamente (ciascuna di dimensione $n/2$, quindi $b = 2$) e poi le fonde (un'operazione che costa $\Theta(n)$). La sua relazione è: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$.

Ricerca Binaria

Per cercare in un array ordinato, fa un confronto, poi chiama ricorsivamente sé stessa su *una* sola metà ($a = 1$) di dimensione $n/2$ ($b = 2$). Il costo del confronto è costante, $\Theta(1)$. La sua relazione è: $T(n) \leq T\left(\frac{n}{2}\right) + \Theta(1)$.

Come Risolvere Queste Relazioni?

Una volta che abbiamo l'equazione, come troviamo la complessità finale? Gli appunti elencano quattro metodi:

1. **Metodo Iterativo**
2. **Metodo di Sostituzione** (Induzione)

3. **Albero di Ricorsione** (Metodo grafico)
4. **Teorema Principale (Master Theorem)**

Il Cuore della Lezione: Il Master Theorem

Il Teorema Principale (Master Theorem) è una "ricetta" che funziona solo per le relazioni bilanciate $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. L'idea centrale è **confrontare due "forze"**:

1. Il costo della **ricorsione** (quanti sotto-problemi si creano).
2. Il costo del **lavoro extra** $f(n)$ (la "forzante").

Il Master Theorem

Data $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, si calcola la **"Funzione Spartiacque"**: $n^{\log_b a}$. Confrontando $f(n)$ con $n^{\log_b a}$ si ricade in uno dei tre casi:

- **Caso 1:** $f(n)$ **polinomialmente minore** ($f(n) = O(n^{\log_b a-\epsilon})$)
  - **Logica:** Il costo è dominato dalla ricorsione (dalle foglie).
  - **Soluzione:** $T(n) = \Theta(n^{\log_b a})$.
- **Caso 2:** $f(n)$ **circa uguale** ($f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$)
  - **Logica:** Le forze sono bilanciate; il costo è lo stesso ad ogni livello.
  - **Soluzione:** $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$. (Se $k = 0$, la soluzione è $\Theta(n^{\log_b a} \cdot \log n)$).
- **Caso 3:** $f(n)$ **polinomialmente maggiore** ($f(n) = \Omega(n^{\log_b a+\epsilon})$)
  - **Logica:** Il costo è dominato dal lavoro extra $f(n)$ (il collo di bottiglia).
  - **Controllo:** Richiede la "Condizione di Regolarità" ($af(n/b) \leq cf(n)$).
  - **Soluzione:** $T(n) = \Theta(f(n))$.

Applicazioni del Master Theorem

Mergesort

$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

- $a = 2, b = 2$. Spartiacque: $n^{\log_2 2} = n$.
- Confronto: $f(n) = \Theta(n)$ è *uguale* allo spartiacque (Caso 2 con $k = 0$).
- **Soluzione:** $\Theta(n \log n)$.

Ricerca Binaria

$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$

- $a = 1, b = 2$. Spartiacque: $n^{\log_2 1} = n^0 = 1$.
- Confronto: $f(n) = \Theta(1)$ è *uguale* allo spartiacque (Caso 2 con $k = 0$).
- **Soluzione:** $\Theta(1 \cdot \log n) = \Theta(\log n)$.

Esempio (Min/Max)

$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1)$

- $a = 2, b = 2$. Spartiacque: $n^{\log_2 2} = n$.
- Confronto: $f(n) = \Theta(1)$ è *polinomialmente minore* di n (Caso 1).
- **Soluzione:** $\Theta(n)$.

Esempio 1 (dal testo)

$T(n) = 9T\left(\frac{n}{3}\right) + n$

- $a = 9, b = 3$. Spartiacque: $n^{\log_3 9} = n^2$.
- Confronto: $f(n) = n$ è *polinomialmente minore* di n^2 (Caso 1).
- **Soluzione:** $\Theta(n^2)$.

Esempio 2 (dal testo)

$T(n) \leq T\left(\frac{2n}{3}\right) + 1$

- $a = 1, b = 3/2$. Spartiacque: $n^{\log_{3/2} 1} = n^0 = 1$.
- Confronto: $f(n) = 1$ è *uguale* allo spartiacque (Caso 2 con $k = 0$).
- **Soluzione:** $\Theta(\log n)$.

Esempio 3 (dal testo)

$T(n) = 3T\left(\frac{n}{4}\right) + n \log n$

- $a = 3, b = 4$. Spartiacque: $n^{\log_4 3} \approx n^{0.792}$.
- Confronto: $f(n) = n \log n$ è *polinomialmente maggiore* (Caso 3).
- (Si verifica la condizione di regolarità).
- **Soluzione:** $\Theta(f(n)) = \Theta(n \log n)$.

Riepilogo della Lezione

Gli appunti della Lezione 23 introducono le **Relazioni di Ricorrenza** per analizzare $T(n)$ degli algoritmi ricorsivi. Si concentrano sulle **Relazioni Bilanciate** ($T(n) = aT(n/b) + f(n)$). Dopo aver elencato quattro metodi di risoluzione, si focalizzano sul **Master Theorem**.
Il teorema confronta $f(n)$ con lo "spartiacque" $n^{\log_b a}$ e definisce tre casi:

1. **Caso 1 ($f(n)$ minore):** Soluzione: $\Theta(n^{\log_b a})$.
2. **Caso 2 ($f(n)$ uguale):** Soluzione: $\Theta(n^{\log_b a} \cdot \log n)$ (o $\log^{k+1} n$).
3. **Caso 3 ($f(n)$ maggiore):** Soluzione: $\Theta(f(n))$ (con C.R.).

ASA (Esercizi per casa)

Esercizi ASA

Risolvere le seguenti relazioni di ricorrenza:

1. $T(n) = 3T(\frac{n}{2}) + n^2$
2. $T(n) = 3T(\frac{n}{4}) + \frac{n}{5} \log n$

Parte VI

Lezione 24 (13/11/2025)

26 Dimostrazione del Teorema Principale

L'obiettivo è risolvere la relazione di ricorrenza $T(n) = aT(n/b) + f(n)$. Si può derivare la formula generale usando l'albero di ricorsione o il metodo iterativo.

26.1 Metodo Iterativo (Derivazione della Formula)

Si espande la ricorrenza sostituendo $T(n)$ dentro sé stessa.

Formula Generale (Metodo Iterativo)

Partiamo dalla ricorrenza:

$$T(n) = aT(n/b) + f(n)$$

Sostituiamo $T(n/b)$ nell'equazione:

$$T(n) = a\left[aT(n/b^2) + f(n/b)\right] + f(n) = a^2T(n/b^2) + af(n/b) + f(n)$$

Sostituiamo $T(n/b^2)$ nell'equazione:

$$T(n) = a^2\left[aT(n/b^3) + f(n/b^2)\right] + af(n/b) + f(n) = a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n)$$

Dopo i passi, la formula generale è:

$$T(n) = a^iT(n/b^i) + \sum_{j=0}^{i-1} a^j f(n/b^j)$$

Ci si ferma al caso base quando la dimensione del problema è 1, cioè $n/b^i = 1$, che avviene quando $i = \log_b n$. Sostituendo $i = \log_b n$:

$$T(n) = a^{\log_b n}T(1) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

Usando l'identità $a^{\log_b n} = n^{\log_b a}$ (dimostrata sotto) e sapendo che $T(1) = \Theta(1)$, la formula finale del costo è:

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

Questo costo totale è la somma di due parti:

- $\Theta(n^{\log_b a})$: Il costo per la soluzione dei casi base (le foglie dell'albero).
- $\sum a^j f(n/b^j)$: Il costo totale del lavoro di "Divide" e "Combine" speso a tutti i livelli della ricorsione.

Identità delle Foglie: $a^{\log_b n} = n^{\log_b a}$

Dimostrazione: Si parte da $a^{\log_b n}$. Si applica la proprietà $x = n^{\log_n x}$:

$$a^{\log_b n} = (n^{\log_n a})^{\log_b n}$$

Si applica la formula del cambio di base $\log_n a = \frac{\log_b a}{\log_b n}$:

$$a^{\log_b n} = \left(n^{\frac{\log_b a}{\log_b n}}\right)^{\log_b n}$$

Moltiplicando gli esponenti:

$$a^{\log_b n} = n^{\frac{\log_b a}{\log_b n} \cdot \log_b n} = n^{\log_b a}$$

26.2 Analisi dei Casi del Teorema

L'analisi consiste nel determinare quale dei due termini della formula $T(n) = \Theta(n^{\log_b a}) + \sum \dots$ domina.

Caso 1: $f(n)$ polinomialmente minore

- **Condizione:** $f(n) \in O(n^{\log_b a - \epsilon})$ per $\epsilon > 0$.
- **Analisi:** La sommatoria $\sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$ può essere analizzata come una serie geometrica. Sostituendo la condizione, si dimostra che la somma cresce più lentamente del primo termine (la ragione della serie è $r = b^\epsilon > 1$).
- **Logica:** Il costo è dominato dal lavoro svolto nei casi base (le foglie).
- **Soluzione:** $T(n) \in \Theta(n^{\log_b a})$.

Caso 2: $f(n)$ bilanciato (caso $k = 0$)

- **Condizione:** $f(n) = \Theta(n^{\log_b a})$.
- **Analisi:** Partiamo dalla formula $T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$. Sostituiamo la condizione $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$ nella sommatoria:

  $$\sum_{j=0}^{\log_b n - 1} a^j \cdot \left(\frac{n}{b^j}\right)^{\log_b a} = \sum_{j=0}^{\log_b n - 1} a^j \cdot \frac{n^{\log_b a}}{(b^{\log_b a})^j} = \sum_{j=0}^{\log_b n - 1} a^j \cdot \frac{n^{\log_b a}}{a^j}$$

  Semplificando a^j, otteniamo:

  $$\sum_{j=0}^{\log_b n - 1} n^{\log_b a} = n^{\log_b a} \sum_{j=0}^{\log_b n - 1} 1 = n^{\log_b a} \cdot (\log_b n)$$

- **Logica:** Il costo del lavoro extra è bilanciato con il costo delle foglie. Il costo totale è il costo di un livello ($n^{\log_b a}$) moltiplicato per il numero di livelli ($\log n$).
- **Soluzione:** $T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \cdot \log n) = \Theta(n^{\log_b a} \cdot \log n)$.

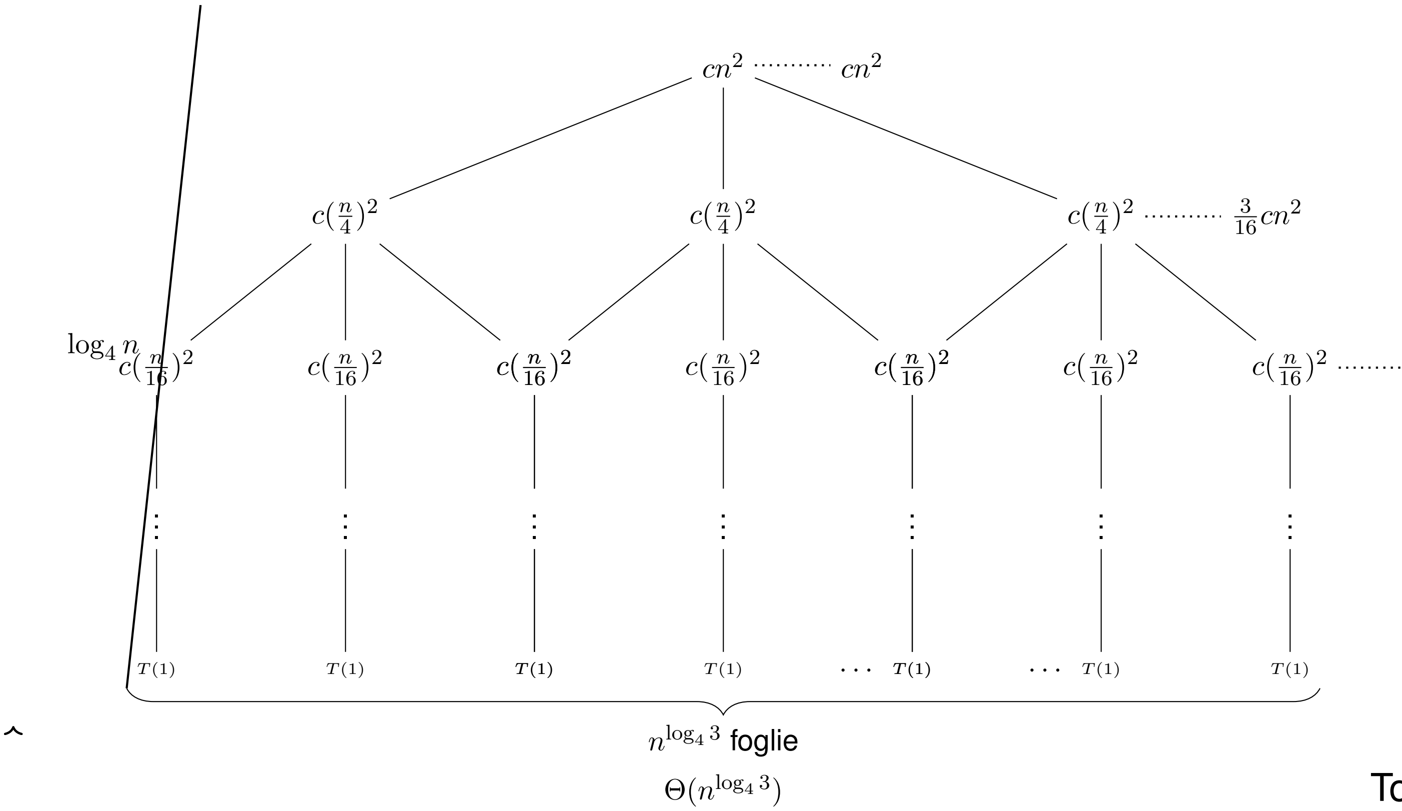


Figura 1: Visualizzazione dell'albero di ricorsione per $T(n) = 3T(n/4) + cn^2$. Questo è un esempio del **Caso 3** del Master Theorem, dove il costo è dominato dalla radice (root).

27 Esercizio (Compitino 24-25)

Analizzare la complessità di un algoritmo la cui struttura (semplificata) è la seguente, ipotizzando diversi costi per il lavoro $f(n)$.

Analisi Algoritmo Ricorsivo

Dato il seguente algoritmo:

Algoritmo

```
Procedure ALGO(A, p, r)
    If p < r then
        q = ⌊(p + r)/2⌋
        ALGO(A, p, q)                        ▷ Costo T(n/2)
        ALGO(A, q+1, r)                      ▷ Costo T(n/2)
        ALGO(A, p, q)                        ▷ Costo T(n/2)
        ALGO(A, q+1, r)                      ▷ Costo T(n/2)
        ... (Lavoro extra con costo f(n))...
    end If
end Procedure
```

Analisi Ricorsiva

L'algoritmo divide il problema in 2 metà ($n/2$) ma effettua **4 chiamate ricorsive** ($a = 4$). Questo porta a un costo elevato che domina il lavoro locale $f(n)$, a meno che $f(n)$ non sia molto pesante.

L'algoritmo fa $a = 4$ chiamate ricorsive su sottoproblemi di dimensione $n/2$ (quindi $b = 2$).

Caso Pessimo: $f(n) = n^2$

La relazione di ricorrenza è: $T(n) = 4T(n/2) + n^2$.

- $a = 4$, $b = 2$.
- **Spartiacque:** $n^{\log_b a} = n^{\log_2 4} = n^2$.
- **Confronto:** $f(n) = n^2$ è uguale allo spartiacque.
- Siamo nel **Caso 2** (con $k = 0$).
- **Soluzione:** $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n^2 \cdot \log n)$.

Caso Ottimo (ipotetico): $f(n) = n$

La relazione di ricorrenza è: $T(n) = 4T(n/2) + n$.

- $a = 4$, $b = 2$. **Spartiacque:** n^2.
- **Confronto:** $f(n) = n$ è polinomialmente minore di n^2 (poiché $n = O(n^{2-\epsilon})$ per $\epsilon = 1$).
- Siamo nel **Caso 1**.
- **Soluzione:** $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

Parte VII
Master's Theorem

Quando si tratta di risolvere equazioni di ricorrenza **bilanciate**, è possibile utilizzare il Master's Theorem.

$$T(n) = \begin{cases} \Theta(1) & n \leq k \\ a \cdot T(\frac{n}{b}) + f(n) & n > k \end{cases} \tag{1}$$

L'intuizione consiste nel fare un confronto tra $f(n)$ e $n^{\log_b a}$.

Master Theorem: I Tre Casi

Ci sono tre casi possibili:

- **Minore**: $f(n) = O(n^{\log_b a - \epsilon})$ per qualche costante $\epsilon > 0$. $f(n)$ cresce **polinomialmente** più lentamente di $n^{\log_b a}$. *Soluzione*: $T(n) = \Theta(n^{\log_b a})$.

Esempio

Data la seguente equazione di ricorrenza:

$$T(n) = 9 \cdot T(\frac{n}{3}) + n \tag{2}$$

Abbiamo che $a = 9$, $b = 3$, $f(n) = n$, $n^{\log_3 9} = n^2$.Possiamo dedurre quindi che, per un $\epsilon = 1$:

$$f(n) = n = O(n^{\log_3 9 - \epsilon}) = O(n) \tag{3}$$

- **Uguale**: $f(n) = \Theta(n^{\log_b a} \cdot \ln^k n)$ per qualche costante $k \geq 0$. $f(n)$ e $n^{\log_b a}$ crescono allo stesso modo. *Soluzione*: $T(n) = \Theta(n^{\log_b a} \cdot \ln^{k+1} n)$.

- **Maggiore**: $f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche costante $\epsilon > 0$. $f(n)$ cresce **polinomialmente** più in fretta e rispetta la **condizione di regolarità**: $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ per $c < 1$. *Soluzione*: $T(n) = \Theta(f(n))$.

Osservazione

Il Master's Theorem si può usare solamente quando $f(x)$ cresce **polinomialmente** più in fretta o lentamente di $n^{\log_b a}$.

28 Guida Pratica all'Applicazione del Master's Theorem

Il Teorema Master è uno strumento potente per risolvere equazioni di ricorrenza **bilanciate.** La forma standard è:

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

Logica: "Tradurre" l'Equazione

Per usare il teorema, devi prima "tradurre" la tua equazione:

- a: Il numero di **sotto-problemi** ($a \geq 1$).
- n/b: La dimensione di **ciascun sotto-problema** ($b > 1$).
- $f(n)$: Il costo "extra" per **dividere** e **combinare**.

L'idea centrale è confrontare $f(n)$ con $n^{\log_b a}$.

Come Applicarlo in un Esercizio: Passo Passo

Guida Passo-Passo: $T(n) = 9 \cdot T(\frac{n}{3}) + n$

Passo 1: Identificare a**,** b**, e** $f(n)$ Dall'equazione $T(n) = 9 \cdot T(\frac{n}{3}) + n$:

- $a = 9$ (sotto-problemi)
- $b = 3$ (dimensione 1/3)
- $f(n) = n$ (costo extra)

Passo 2: Calcolare il "Confine" Ricorsivo Calcola il valore $n^{\log_b a}$.

- $n^{\log_3 9} = n^2$

Passo 3: Confrontare $f(n)$ **con** $n^{\log_b a}$ Confrontiamo $f(n) = n$ con n^2. **I Tre Casi del Teorema Caso 1:** $f(n)$ **cresce più LENTAMENTE**

- **Logica:** Il costo è dominato dalla ricorsione.
- **Condizione:** $f(n) = O(n^{\log_b a - \epsilon})$ per $\epsilon > 0$.
- **Soluzione:** $T(n) = \Theta(n^{\log_b a})$.

Verifica del nostro Esempio: $f(n) = n$ e $n^{\log_b a} = n^2$. $f(n) = n$ è $O(n^{2-\epsilon})$ scegliendo $\epsilon = 1$. Rientriamo nel **Caso 1**. *Soluzione:* $T(n) = \Theta(n^2)$.
Caso 2: $f(n)$ **cresce alla STESSA VELOCITÀ**

- **Logica:** Costo bilanciato.
- **Condizione:** $f(n) = \Theta(n^{\log_b a} \cdot \ln^k n)$ per $k \geq 0$.
- **Soluzione:** $T(n) = \Theta(n^{\log_b a} \cdot \ln^{k+1} n)$.

Caso 3: $f(n)$ **cresce più VELOCEMENTE**

- **Logica:** Costo dominato dal lavoro extra $f(n)$.
- **Condizione 1:** $f(n) = \Omega(n^{\log_b a + \epsilon})$ per $\epsilon > 0$.
- **Condizione 2 (Regolarità):** $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ per $c < 1$.
- **Soluzione:** $T(n) = \Theta(f(n))$.

Limiti del Teorema

Il Teorema Master **non si può usare** se $f(n)$ non cresce *polinomialmente* più velocemente o più lentamente di $n^{\log_b a}$. Ad esempio, se $f(x) = \log n$ non potremmo utilizzarlo (perché non è polinomialmente diverso da $n^0 = 1$).

Lezione 9 (20/10/2025)

Esercitazione Master's Theorem

Domande

(A) $T(n) = 7T(n/2) + n^2 \quad \forall n \geq n_0$

(A') $T'(n) = aT(n/4) + n^2 \quad \forall n \geq n'_0$

Domanda (A): Qual è il più grande valore di a per cui T' è asintoticamente $<$ (*minore del*) valore di T?

Domanda (A'): Qual è il più grande valore di a per cui T' è asintoticamente uguale al valore di T?

Introduzione all'esercizio

In questa parte dell'esercizio abbiamo come scopo riportare in una forma adeguata i nostri valori

Costo in tempo di A

$$T(n) = 7T(n/2) + n^2$$

- $a = 7, b = 2, f(n) = n^2$
- $\log_b a \implies \log_2 7$
- $f(n) = n^2 \in O(n^{\log_2 7 - \epsilon})$
- $0 < \epsilon \leq \log_2 7 - 2$
- 1° CASO $\implies T(n) = \Theta(n^{\log_2 7})$ (*circa* $n^{2.8\dots}$)

Costo in tempo di A'

$$T'(n) = aT(n/4) + n^2$$

- $a = a, b = 4, f(n) = n^2$
- $\log_b a \implies \log_4 a$
- Quale caso del Teorema?
  - Ramo 1: $\log_4 a < 2 \implies a < 16$ (*Caso 3?*)
  - Ramo 2: $\log_4 a = 2 \implies a = 16$ (*Caso 2?*)
  - Ramo 3: $\log_4 a > 2 \implies a > 16$ (*Caso 1*)

Logica per risolvere

L'obiettivo è usare il Teorema Master per risolvere le ricorrenze. Il teorema si applica a ricorrenze della forma: Si calcola il valore critico $\log_b a$ e lo si confronta con l'esponente di $f(n)$ (supponendo $f(n) = n^k$).

Step 1: Analisi di T(n) (Ricorrenza A)

La prima ricorrenza è $T(n) = 7T(n/2) + n^2$.

- **Identificazione parametri:**
  - $a = 7$
  - $b = 2$
  - $f(n) = n^2$
- **Calcolo esponente critico:**
  - Calcoliamo $\log_b a = \log_2 7$.
  - Sappiamo che $2^2 = 4$ e $2^3 = 8$, quindi $\log_2 7$ è un numero tra 2 e 3 (circa 2.81).
- **Confronto e applicazione Teorema Master:**
  - Confrontiamo $f(n) = n^2$ con $n^{\log_b a} = n^{\log_2 7}$.
  - Poiché $2 < \log_2 7$, $f(n)$ è polinomialmente più piccola di $n^{\log_b a}$.
  - Questo corrisponde al **Caso 1** del Teorema Master: $f(n) = O(n^{\log_b a - \epsilon})$, dove $\epsilon = \log_2 7 - 2 > 0$.
  - La soluzione è quindi $T(n) = \Theta(n^{\log_b a})$.

  Risultato per T(n): $T(n) = \Theta(n^{\log_2 7})$

Step 2: Analisi di T'(n) (Ricorrenza A')

La seconda ricorrenza è $T'(n) = aT(n/4) + n^2$.

- **Identificazione parametri:**
  - $a = a$ (sconosciuto)
  - $b = 4$
  - $f(n) = n^2$
- **Calcolo esponente critico:**
  - L'esponente critico è $\log_b a = \log_4 a$.
- **Confronto e applicazione Teorema Master:**
  - Dobbiamo confrontare $\log_4 a$ con l'esponente di $f(n)$, che è 2.
  - Questo crea tre scenari possibili:
  - **Scenario 1 (Caso 1 del Teorema):** $\log_4 a > 2$
    * Questo succede quando $a > 4^2$, cioè $a > 16$.
    * La soluzione è dominata dalla ricorsione: $T'(n) = \Theta(n^{\log_4 a})$.
  - **Scenario 2 (Caso 2 del Teorema):** $\log_4 a = 2$
    * Questo succede quando $a = 4^2$, cioè $a = 16$.
    * La soluzione è: $T'(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^2 \log n)$.
  - **Scenario 3 (Caso 3 del Teorema):** $\log_4 a < 2$
    * Questo succede quando $a < 16$.
    * La soluzione è dominata dal costo $f(n)$: $T'(n) = \Theta(n^2)$ (assumendo la condizione di regolarità, che è soddisfatta).

Step 3: Risposta alle Domande

Ora usiamo i risultati degli Step 1 e 2 per rispondere alle domande.

Domanda (A'): Trovare a t.c. $T'(n)$ è uguale a $T(n)$ Vogliamo trovare il più grande a per cui $T'(n)$ è asintoticamente uguale a $T(n)$.

Controlliamo quale dei nostri 3 scenari per $T'(n)$ può soddisfare questa uguaglianza:

- **Se $a > 16$ (Scenario 1):** $T'(n) = \Theta(n^{\log_4 a})$.
  - Dobbiamo avere $\Theta(n^{\log_4 a}) = \Theta(n^{\log_2 7})$.
  - Questo richiede che gli esponenti siano uguali: $\log_4 a = \log_2 7$.
  - Risolviamo per a (usando il cambio di base: $\log_4 a = \frac{\log_2 a}{\log_2 4} = \frac{\log_2 a}{2}$):

$$\frac{\log_2 a}{2} = \log_2 7$$
$$\log_2 a = 2\log_2 7$$
$$\log_2 a = \log_2(7^2)$$
$$a = 49$$

  - Questo valore $a = 49$ è coerente con la condizione $a > 16$.
- **Se $a = 16$ (Scenario 2):** $T'(n) = \Theta(n^2 \log n)$.
  - $n^2 \log n$ non è asintoticamente uguale a $n^{\log_2 7}$ (che è $\approx n^{2.81}$).
- **Se $a < 16$ (Scenario 3):** $T'(n) = \Theta(n^2)$.
  - n^2 non è asintoticamente uguale a $n^{\log_2 7}$.

Risposta (A'): L'unico valore (e quindi il più grande) per cui $T'(n)$ è asintoticamente uguale a $T(n)$ è $a = 49$.

Domanda (A): Trovare a t.c. $T'(n)$ è minore di $T(n)$ Vogliamo trovare il più grande a per cui $T'(n)$ è asintoticamente minore di $T(n)$ (cioè $T'(n) = o(T(n))$).

Controlliamo di nuovo i nostri 3 scenari:

- **Se $a > 16$ (Scenario 1):** $T'(n) = \Theta(n^{\log_4 a})$.
  - Vogliamo $n^{\log_4 a} = o(n^{\log_2 7})$.
  - Questo è vero se l'esponente $\log_4 a$ è strettamente minore di $\log_2 7$.
  - $\log_4 a < \log_2 7 \implies a < 49$ (dal calcolo precedente).
  - Questo scenario è valido per l'intervallo $16 < a < 49$.
- **Se $a = 16$ (Scenario 2):** $T'(n) = \Theta(n^2 \log n)$.
  - Vogliamo $n^2 \log n = o(n^{\log_2 7})$.
  - Poiché $2 < \log_2 7 \approx 2.81$, $n^2 \log n$ cresce più lentamente di $n^{\log_2 7}$. L'affermazione è vera.
  - Quindi $a = 16$ è una soluzione.
- **Se $a < 16$ (Scenario 3):** $T'(n) = \Theta(n^2)$.
  - Vogliamo $n^2 = o(n^{\log_2 7})$.

- Poiché $2 < \log_2 7$, n^2 cresce più lentamente di $n^{\log_2 7}$. L'affermazione è vera.
- Questo scenario è valido per $1 \leq a < 16$.

Unendo tutti i casi validi, $T'(n)$ è asintoticamente minore di $T(n)$ per ogni a nell'intervallo $1 \leq a < 49$.

Risposta (A): La domanda chiede il più grande valore di a. Se a può essere un numero reale, non esiste un "più grande" valore (il limite è 49). Se si intende il più grande valore intero, la risposta è $a = 48$.

Ricorda bene che $\log n$ va sempre più veloce di qualsiasi polinomio di n. Se dovessimo cercare qualcosa, $\log n$ va più veloce di qualunque polinomio di n.

Parte VIII

Approfondimento: Limiti Inferiori alla Difficoltà di un Problema

29 Limiti Inferiori alla Difficoltà di un Problema

Vogliamo stabilire quanto è "difficile" un problema $\mathcal{P}$ intrinsecamente, indipendentemente dall'algoritmo usato.

Limite Inferiore

Il **Limite Inferiore** $L(n)$ misura la difficoltà di un problema $\mathcal{P}$ in funzione della dimensione n dell'input. Rappresenta la complessità al **caso pessimo** del **miglior algoritmo possibile** che risolve $\mathcal{P}$.

$$\forall \text{ algoritmo } A \text{ che risolve } \mathcal{P}, \quad T_A(n) \geq L(n) \quad \text{(nel caso pessimo)}$$

Ovvero, $L(n)$ è il minimo numero di operazioni necessarie per risolvere il caso pessimo.

29.1 Esempio Introduttivo: Ricerca

Consideriamo il problema $\mathcal{P}$: Ricerca di una chiave in un vettore **ordinato**.

- **Approccio 1: Scansione Lineare**. Complessità $O(n)$. Un algoritmo che risolve $\mathcal{P}$ fornisce un **Limite Superiore** alla difficoltà di $\mathcal{P}$.
- **Approccio 2: Ricerca Binaria**. Complessità $O(\log n)$. Migliora il limite superiore.
- **Domanda:** Posso fare di meglio? Qual è il **Limite Inferiore** (la "pavimentazione") sotto il quale non posso scendere?

30 Criteri per Stabilire i Limiti Inferiori

Esistono diverse tecniche per individuare $L(n)$.

30.1 1° Criterio: Dimensione dell'Input

Se la soluzione di un problema richiede, nel caso pessimo, l'esame di tutti i dati in ingresso, allora la dimensione dell'input n è un limite inferiore.

$$L(n) = \Omega(n)$$

Questo esclude la possibilità che si possa "fare meno" di leggere l'input.

Ricerca MAX in Vettore Non Ordinato

Per trovare il massimo in un vettore non ordinato, devo necessariamente analizzare tutti gli n elementi (altrimenti il massimo potrebbe essere proprio l'elemento saltato).

- **Limite Inferiore:** $L(n) = n$.
- **Algoritmo noto:** Scansione Lineare, costo $O(n)$.

Poiché il costo dell'algoritmo (n) coincide con il limite inferiore (n), l'algoritmo di **Scansione Lineare è OTTIMO**.

30.2 2° Criterio: Albero di Decisione

Questo criterio si applica a problemi risolvibili attraverso una sequenza di "decisioni" (es. confronti tra valori) che riducono via via lo spazio delle soluzioni possibili.

30.2.1 Struttura dell'Albero

Possiamo modellare l'esecuzione come un albero dove:

- **Nodo Interno:** Rappresenta un confronto/decisione (es. $x > y$?).
- **Foglia:** Rappresenta una possibile **soluzione** finale.
- **Cammino Radice-Foglia:** Rappresenta una specifica esecuzione.
- **Altezza dell'Albero:** Rappresenta il **caso pessimo** (il cammino più lungo).

Limite Inferiore basato sulle Soluzioni

Sia $SOL(n)$ il numero di possibili soluzioni distinte per un problema di dimensione n (ovvero il numero di foglie dell'albero). In un albero binario (o ternario), l'altezza h deve soddisfare:

$$h \geq \log_2(\#\text{foglie})$$

Pertanto, il limite inferiore è dato dal logaritmo del numero delle possibili soluzioni:

$$L(n) = \Omega(\log_2(SOL(n)))$$

Osservazione

L'algoritmo migliore al caso pessimo è quello che minimizza l'altezza dell'albero di decisione, ovvero quello che mantiene l'albero **bilanciato** (altezza logaritmica rispetto al numero di foglie).

30.2.2 Applicazione: Ricerca in Vettore Ordinato

Analizziamo il problema della ricerca di una chiave k in un array ordinato $A[1..n]$.

- **Possibili Soluzioni ($SOL(n)$):** L'elemento può trovarsi in una delle n posizioni, oppure non esserci.

$$\#\text{Soluzioni} = n + 1$$

- **Limite Inferiore:**

$$L(n) = \log_2(n + 1) \approx \log_2 n$$

- **Confronto:** L'algoritmo **Ricerca Binaria** ha costo $O(\log n)$.

Poiché il costo dell'algoritmo coincide con il limite inferiore, la **Ricerca Binaria è OTTIMA**.

30.3 3° Criterio: Eventi Contabili

Se la ripetizione di un certo evento è indispensabile per risolvere il problema, allora:

$$L(n) = (\#\text{volte che si deve ripetere}) \times (\text{costo evento})$$

Ricerca MAX in Array con Confronti

Evento necessario: Un elemento, per non essere il massimo, deve "uscire perdente" da un confronto con un altro valore.

- Abbiamo n candidati al massimo.
- Alla fine deve rimanere 1 solo vincitore.
- Devono esserci quindi $n-1$ "perdenti".
- Ogni confronto elimina al massimo 1 candidato (il perdente).

Necessari almeno $n-1$ confronti.

$$L(n) = \Omega(n)$$

31 Approfondimento: Limite Significativo

Non tutti i limiti inferiori sono utili. Un limite inferiore è utile solo se è "stretto", ovvero vicino alla complessità del miglior algoritmo conosciuto.

Confronto tra Criteri: Ricerca Non Ordinata

Consideriamo la ricerca di k in un vettore **non ordinato**.

1. **Criterio Albero di Decisione:**

$$\#\text{Soluzioni} = n + 1 \implies L(n) = \Omega(\log n)$$

   Questo limite è matematicamente vero, ma **non significativo**. Non esiste un algoritmo che risolva questo problema in $\log n$. È un limite troppo basso.

2. **Criterio Dimensione Input:** Bisogna guardare tutti gli elementi per essere sicuri.

$$L(n) = \Omega(n)$$

Conclusione: Il limite derivato dalla dimensione dell'input (n) è più alto ("alza l'asticella") e coincide con il costo della Scansione Lineare ($O(n)$). Quando Limite Inferiore e Superiore coincidono, il limite è **Significativo** e possiamo dire che l'algoritmo è **Ottimo**.

Parte IX
Lezione 25 (17/11/2025)

32 Esercizio su Teorema Master (Confronto Asintotico)

Consideriamo due algoritmi caratterizzati dalle seguenti ricorrenze:

(A) $T(n) = 7T\left(\frac{n}{2}\right) + n^2$

(A') $T'(n) = aT'\left(\frac{n}{4}\right) + n^2$

Domanda: Qual è il più grande valore di a per cui T' è asintoticamente più veloce di T?

32.1 Costo in Tempo di A

Analizziamo $T(n) = 7T(n/2) + n^2$ con il Teorema Master.

- $a = 7, b = 2, f(n) = n^2$.
- Calcoliamo lo spartiacque: $n^{\log_b a} = n^{\log_2 7} \approx n^{2.8}$.
- Confrontiamo con $f(n)$: $n^2 = O(n^{\log_2 7-\epsilon})$ (con $\epsilon \approx 0.8$).
- Siamo nel **Caso 1**.

$$T(n) = \Theta(n^{\log_2 7})$$

32.2 Costo in Tempo di A'

Analizziamo $T'(n) = aT'(n/4) + n^2$.

- $a = a, b = 4, f(n) = n^2$.
- Spartiacque: $n^{\log_4 a}$.

Dobbiamo confrontare $\log_4 a$ con l'esponente di $f(n)$ (che è 2). Ci sono 3 casi possibili per a:

1. **Caso 3 ($\log_4 a < 2 \iff a < 16$):** La forzante n^2 domina. $T'(n) = \Theta(n^2)$. Verifica condizione regolarità: $a(n/4)^2 \leq cn^2 \implies a/16 \leq c$. Vera per $a < 16$. In questo caso $T'(n) = \Theta(n^2)$, che è sicuramente più veloce di $\Theta(n^{2.8})$.
2. **Caso 2 ($\log_4 a = 2 \iff a = 16$):** Equilibrio. $T'(n) = \Theta(n^2 \log n)$. Anche questo è più veloce di $\Theta(n^{2.8})$.
3. **Caso 1 ($\log_4 a > 2 \iff a > 16$):** Le foglie dominano. $T'(n) = \Theta(n^{\log_4 a})$. Affinché T' sia più veloce di T, deve valere:

   $$n^{\log_4 a} < n^{\log_2 7} \implies \log_4 a < \log_2 7$$

   Usando il cambio di base ($\log_4 a = \frac{\log_2 a}{2}$):

   $$\frac{\log_2 a}{2} < \log_2 7 \implies \log_2 a < 2\log_2 7 \implies \log_2 a < \log_2 49 \implies a < 49$$

Soluzione: A' è più veloce di A per $a < 49$. Il valore intero massimo è **48**.

33 Analisi di Algoritmi (Esercizi Vari)

33.1 Esercizio "Mistero"

Algoritmo

```
Procedure MISTERO(n)
    If n < 10 then Return 1
    end If
    x = MISTERO(⌊n/4⌋) + MISTERO(⌊n/4⌋)        ▷ 2 chiamate ricorsive
    L = 1
    While i < n do                               ▷ Ciclo esterno
        j = 1
        While j < n do                           ▷ Ciclo interno
            ...
            j = j + 1
        end While
        i = i · 3
    end While
    y = MISTERO(⌊n/4⌋)                           ▷ 1 chiamata ricorsiva
    Return x + y
end Procedure
```

Analisi Ricorsiva

La funzione combina chiamate ricorsive e cicli annidati:

- **Ricorsione**: 3 chiamate su dimensione $n/4$ (x ne fa 2, y ne fa 1). $T(n) = 3T(n/4) + f(n)$.
- **Costo locale ($f(n)$)**: I cicli annidati. Il ciclo interno lavora su j, l'esterno su i che cresce esponenzialmente (3^k).

Analisi:

- Chiamate ricorsive: 3 chiamate su $n/4$. Quindi $a = 3, b = 4$.
- Costo f(n): Il ciclo interno è $\Theta(n)$, quello esterno è logaritmico? (Dagli appunti sembra indicato come $\Theta(n \log n)$).
- Ricorrenza: $T(n) = 3T(n/4) + \Theta(n \log n)$.
- Master Theorem:
  - $n^{\log_4 3} \approx n^{0.79}$.
  - $f(n) = n \log n$ è polinomialmente maggiore ($n^1 > n^{0.79}$).
  - **Caso 3.**
- Soluzione: $T(n) = \Theta(n \log n)$.

33.2 Algo 1 (Radice Quadrata)

- Ricorrenza data: $T(n) = 2T(n/4) + \sqrt{n}$.
- $a = 2, b = 4 \implies n^{\log_4 2} = n^{0.5} = \sqrt{n}$.

- $f(n) = \sqrt{n}$.
- Siamo nel **Caso 2** (uguali).
- Soluzione: $T(n) = \Theta(\sqrt{n}\log n)$.

33.3 Algo 2 (Somma Ricorsiva)

- Divide l'array in due parti (p, q e $q + 1, r$).
- Fa 2 chiamate ricorsive: $a = 2, b = 2$.
- Costo di combinazione (somma): $\Theta(1)$.
- Ricorrenza: $T(n) = 2T(n/2) + \Theta(1)$.
- Master Theorem: $n^{\log_2 2} = n^1$. $f(n) = n^0$.
- **Caso 1**.
- Soluzione: $T(n) = \Theta(n)$.

33.4 Confronto Finale

Confrontiamo:

1. $T_A(n) = 9T(n/3) + 2n^2$.
2. $T_{A'}(n) = 3T(n/2) + n^2 \log^2 n$.

Analisi A: $a = 9, b = 3 \implies n^{\log_3 9} = n^2$. $f(n) = 2n^2$. Caso 2. $T_A(n) = \Theta(n^2 \log n)$.
Analisi A': $a = 3, b = 2 \implies n^{\log_2 3} \approx n^{1.58}$. $f(n) = n^2 \log^2 n$. $f(n)$ è maggiore dello spartiacque. Caso 3. $T_{A'}(n) = \Theta(n^2 \log^2 n)$.
Conclusione: T_A è asintoticamente migliore (più veloce) di $T_{A'}$.

Parte X

Lezione 26 (19/11/2025)

34 Limiti Inferiori alla Difficoltà di un Problema

Vogliamo stabilire quanto è "difficile" un problema $\mathcal{P}$ intrinsecamente, indipendentemente dall'algoritmo usato.

Limite Inferiore

Il **Limite Inferiore** $L(n)$ misura la difficoltà di un problema $\mathcal{P}$ in funzione della dimensione n dell'input. Rappresenta la complessità al **caso pessimo** del **miglior algoritmo possibile** che risolve $\mathcal{P}$.

$$\forall \text{ algoritmo } A \text{ che risolve } \mathcal{P}, \quad T_A(n) \geq L(n) \quad (\text{nel caso pessimo})$$

Ovvero, $L(n)$ è il minimo numero di operazioni necessarie per risolvere il caso pessimo.

34.1 Esempio Introduttivo: Ricerca

Consideriamo il problema $\mathcal{P}$: Ricerca di una chiave in un vettore **ordinato**.

- **Approccio 1: Scansione Lineare**. Complessità $O(n)$. Un algoritmo che risolve $\mathcal{P}$ fornisce un **Limite Superiore** alla difficoltà di $\mathcal{P}$.
- **Approccio 2: Ricerca Binaria**. Complessità $O(\log n)$. Migliora il limite superiore.
- **Domanda:** Posso fare di meglio? Qual è il **Limite Inferiore** (la "pavimentazione") sotto il quale non posso scendere?

35 Criteri per Stabilire i Limiti Inferiori

35.1 1° Criterio: Dimensione dell'Input

Se la soluzione di un problema richiede, nel caso pessimo, l'esame di tutti i dati in ingresso, allora la dimensione dell'input n è un limite inferiore.

$$L(n) = \Omega(n)$$

Ricerca MAX in Vettore Non Ordinato

Per trovare il massimo in un vettore non ordinato, devo necessariamente analizzare tutti gli n elementi (altrimenti il massimo potrebbe essere proprio l'elemento saltato).

- **Limite Inferiore:** $L(n) = n$.
- **Algoritmo noto:** Scansione Lineare, costo $O(n)$.

Poiché il costo dell'algoritmo (n) coincide con il limite inferiore (n), l'algoritmo di **Scansione Lineare è OTTIMO**.

35.2 2° Criterio: Albero di Decisione

Questo criterio si applica a problemi risolvibili attraverso una sequenza di "decisioni" (es. confronti tra valori) che riducono via via lo spazio delle soluzioni possibili.

35.2.1 Struttura dell'Albero di Decisione

Possiamo modellare l'esecuzione di un algoritmo basato su confronti come un albero:

- **Nodo Interno:** Rappresenta un confronto/decisione (es. $x > y$?).
- **Foglia:** Rappresenta una possibile **soluzione** finale.
- **Cammino Radice-Foglia:** Rappresenta una specifica esecuzione dell'algoritmo su un dato input.

35.2.2 Relazione con la Complessità

- **Caso Ottimo:** Cammino più breve dalla radice a una foglia.
- **Caso Pessimo:** Cammino più lungo dalla radice a una foglia, ovvero l'**Altezza dell'Albero**.

Per minimizzare il caso pessimo, vogliamo che l'albero sia il più **bilanciato** possibile (altezza minima per un dato numero di foglie).

Limite Inferiore basato sulle Soluzioni

Sia $SOL(n)$ il numero di possibili soluzioni distinte per un problema di dimensione n (ovvero il numero di foglie dell'albero). In un albero binario (o ternario), l'altezza h deve soddisfare:

$$h \geq \log_2(\#\text{foglie})$$

Pertanto, il limite inferiore è dato dal logaritmo del numero delle possibili soluzioni:

$$L(n) = \Omega(\log_2(SOL(n)))$$

Osservazione

L'algoritmo migliore al caso pessimo è quello che minimizza l'altezza dell'albero di decisione, ovvero quello che ha altezza logaritmica rispetto al numero di foglie.

35.2.3 Applicazione: Ricerca in Vettore Ordinato

Analizziamo il problema della ricerca di una chiave k in un array ordinato $A[1..n]$ usando il criterio dell'Albero di Decisione.

- **Possibili Soluzioni ($SOL(n)$):** L'elemento può trovarsi in una delle n posizioni, oppure non esserci.
$$\#\text{Soluzioni} = n + 1$$
- **Limite Inferiore:**
$$L(n) = \log_2(n + 1) \approx \log_2 n$$
- **Confronto:** L'algoritmo **Ricerca Binaria** ha costo $O(\log n)$.

Poiché il costo dell'algoritmo coincide con il limite inferiore, la **Ricerca Binaria è OTTIMA**. Non è necessario (né possibile) fare di meglio basandosi sui confronti.

35.3 3° Criterio: Eventi Contabili (Avversario)

Se la ripetizione di un certo evento è indispensabile per risolvere il problema, allora:

$$L(n) = (\#\text{volte che si deve ripetere}) \times (\text{costo evento})$$

Ricerca MAX in Array con Confronti

Evento necessario: Un elemento, per non essere il massimo, deve "uscire perdente" da un confronto con un altro valore.

- Abbiamo n candidati al massimo.
- Alla fine deve rimanere 1 solo vincitore.
- Devono esserci quindi $n-1$ "perdenti".
- Ogni confronto elimina al massimo 1 candidato (il perdente).

Necessari almeno $n-1$ confronti.

$$L(n) = \Omega(n)$$

36 Osservazione Finale: Confronto tra Criteri

Consideriamo il problema: **Ricerca di k in Vettore NON Ordinato**.

- **Criterio Albero di Decisione:**

$$\#\text{Soluzioni} = n + 1 \implies L(n) = \Omega(\log n)$$

  Questo è un limite inferiore valido, ma è troppo basso ("largo").

- **Criterio Dimensione Input:** Bisogna guardare tutti gli elementi.

$$L(n) = \Omega(n)$$

Il limite inferiore "vero" (o più significativo) è il più alto tra quelli trovati. In questo caso $\Omega(n)$. Quindi, per la ricerca non ordinata, la **Scansione Lineare** (costo n) è ottima, mentre un ipotetico algoritmo logaritmico (suggerito dal criterio dell'albero) non è realizzabile.

Parte XI
Confronto tra Algoritmi di Ordinamento

37 Confronto tra Algoritmi di Ordinamento

Viene presentato un confronto sulla complessità temporale degli algoritmi principali studiati.

Algoritmo	Caso Ottimo	Caso Medio	Caso Pessimo
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Quick Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$

Tabella 1: Confronto complessità temporale

38 Quick Sort: L'Idea

L'algoritmo segue l'approccio *"Divide et Impera"* in tre fasi:

1. **Scelta del Perno (Pivot):** Si sceglie un elemento, ad esempio l'ultimo elemento dell'array: $P = A[r]$.
2. **Partizione (Partition):** L'array A viene diviso in due metà (non necessariamente uguali) rispetto al pivot:
$$A_1 = \{x \in A \mid x \leq P\}$$
$$A_2 = \{x \in A \mid x > P\}$$
Il pivot P viene posizionato tra A_1 e A_2.
3. **Ricorsione:** Si ordinano ricorsivamente i sotto-array A_1 e A_2.

Algorithm 1 QuickSort(A, p, r)

Algoritmo

▷ Goal: Ordina A[p...r]. Prima chiamata: p=1, r=n
If $p < r$ **then**
    $q \leftarrow$ PARTITION(A, p, r)
    QUICKSORT$(A, p, q-1)$
    QUICKSORT$(A, q+1, r)$
end If
▷ #elementi = $r - p + 1$

Logica QuickSort

Algoritmo ricorsivo basato su *Divide et Impera*:

1. **Partition**: Trova la posizione finale del pivot q.
2. **Ricorsione**: Ordina i sotto-array a sinistra e a destra di q.
3. **Base**: Se $p \geq r$, l'array ha 0 o 1 elemento ed è già ordinato.

39 Confronto: Merge Sort vs Quick Sort

Entrambi usano la strategia *Divide et Impera*, ma in modo opposto:

Fase	**Merge Sort**	**Quick Sort**
Divide	**Banale:** $q = \lfloor (p+r)/2 \rfloor$. Divide a metà perfetta.	**Complesso:** $q =$ PARTITION$(\dots)$. Il lavoro "pesante" viene fatto qui.
Impera	MS(A, p, q), MS$(A, q+1, r)$	QS$(A, p, q-1)$, QS$(A, q+1, r)$
Combine	**Complesso:** MERGE$(\dots)$. Necessaria per unire i risultati.	**Banale:** Non necessaria (l'array è ordinato "in loco").

40 La Procedura Partition

La funzione `Partition` riorganizza l'array in loco (senza array di appoggio) in modo lineare $\Theta(n)$.

Algorithm 2 PARTITION(A, p, r)

Algoritmo

$x \leftarrow A[r]$ ▷ Pivot
$i \leftarrow p - 1$
For $j \leftarrow p$ **to** $r - 1$ **do**
    If $A[j] \leq x$ **then**
        $i \leftarrow i + 1$
        Swap$(A[i], A[j])$
    end If
end For
Swap$(A[i+1], A[r])$ ▷ Posiziona il pivot
return $i + 1$

Partition (Lomuto)

Scansiona l'array con un indice j. Se $A[j] \leq Pivot$, incrementa la regione dei "minori" (i) e scambia. Alla fine, il pivot viene messo esattamente tra i minori e i maggiori.

Invariante del ciclo

Durante l'esecuzione, l'array è diviso in regioni:

- $A[p \dots i]$: Elementi $\leq$ Pivot.
- $A[i+1 \dots j-1]$: Elementi $>$ Pivot.
- $A[j \dots r-1]$: Elementi ancora da esaminare.
- $A[r]$: Pivot.

Traccia Partition

Array iniziale: 2 8 7 1 3 5 6 4 (Pivot = 4).
Durante la partizione, gli elementi minori o uguali a 4 vengono spostati a sinistra. Alla fine, il 4 si troverà nella sua posizione corretta definitiva.

41 Variante: Hoare Partition

Viene presentata una variante dell'algoritmo di partizione (Partizione di Hoare) che usa due indici che convergono dagli estremi.

Algorithm 3 HOARE-PARTITION(A, p, r)

Algoritmo

```
x ← A[p]                                ▷ Pivot (qui preso come primo elemento)
i ← p − 1
j ← r + 1
While true do
    repeat
        j ← j − 1
    until A[j] ≤ x
    repeat
        i ← i + 1
    until A[i] ≥ x
    If i < j then
        exchange A[i] with A[j]
    Else
        return j
    end If
end While
```

Hoare Partition

Due indici (i, j) partono dagli estremi e convergono.

- i avanza finché trova elementi "piccoli".
- j indietreggia finché trova elementi "grandi".
- Quando si bloccano entrambi (trovati elementi fuori posto), si scambiano.

È spesso più efficiente di Lomuto in pratica (meno swap).

42 Analisi della Complessità

Sia $n = r - p + 1$. Il tempo di esecuzione $T(n)$ dipende da come il pivot divide l'array (q). La relazione di ricorrenza generale è:

$$T(n) = T(q-1) + T(n-q) + \Theta(n)$$

dove $\Theta(n)$ è il costo della Partition.

42.1 Caso Pessimo

Si verifica quando l'array è **già ordinato** (o ordinato al contrario). In questo caso, il pivot (essendo il massimo o il minimo) divide l'array in un sotto-problema di dimensione $n-1$ e uno di dimensione 0. L'albero di ricorsione diventa una lista lunga n.

Calcolo:

$$T(n) = T(n-1) + T(0) + c \cdot n$$

Sviluppando la somma:

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

42.2 Caso Ottimo

Si verifica quando il pivot divide l'array sempre a metà (come nel Merge Sort), ovvero $q = n/2$.

Complessità: $O(n \log n)$

42.3 Caso Medio

Anche nel caso medio la complessità si dimostra essere:

$$O(n \log n)$$

Osservazione

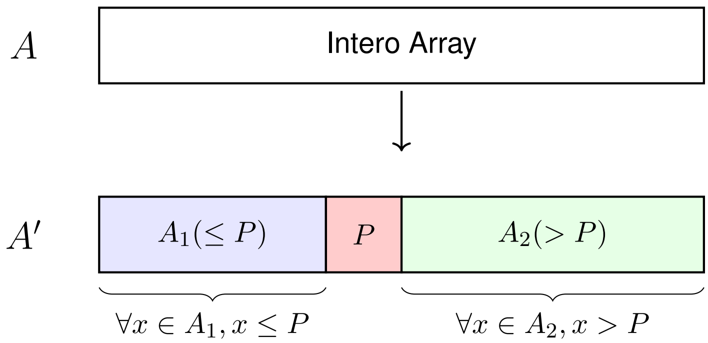
Questo è il motivo per cui il Quick Sort viene utilizzato nella pratica: spesso è più veloce del Merge Sort grazie alle costanti nascoste minori e all'uso della memoria in loco (non richiede array ausiliari per il merge).

Quick Sort: Rappresentazione Grafica

1. L'Idea della Partizione

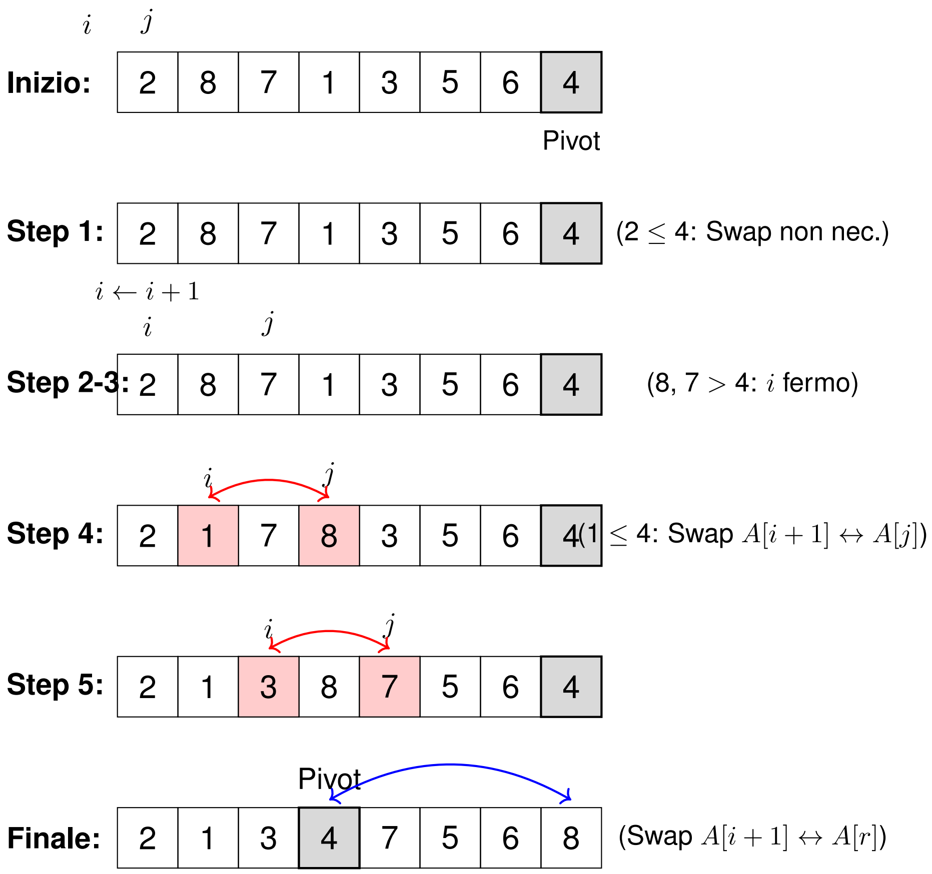
L'array viene diviso in due parti (non necessariamente uguali) rispetto a un elemento "perno" (Pivot) P.

- A_1: elementi $\leq P$
- A_2: elementi $> P$



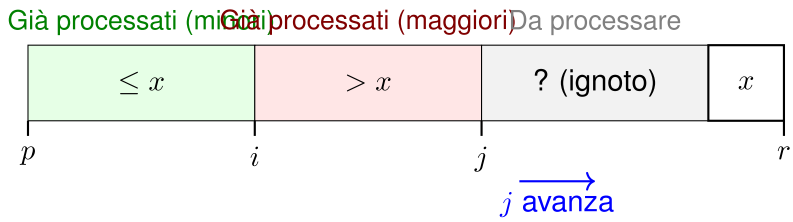
2. Esempio di Traccia (Partition)

Esecuzione della partizione sull'array: `2 8 7 1 3 5 6 4`. Il Pivot è l'ultimo elemento (4).



3. Invariante della Procedura Partition

Durante la scansione, l'array è diviso in 4 regioni dinamiche gestite dagli indici i e j.



Lezione (27/11/2025)

43 Confronto Ordinamenti

Riepilogo Complessità

Ordine di grandezza del:

$\swarrow \quad \searrow$

Algoritmo	**COSTO IN TEMPO**			**COSTO IN SPAZIO**	**Commenti**
	CASO OTTIMO	CASO MEDIO	CASO PESSIMO		
Insertion Sort	n	n^2	n^2	in loco	
Selection Sort	n^2	n^2	n^2	in loco	
Merge Sort	$n \log n$	$n \log n$	$n \log n$	n	OTTIMO in tempo
Quick Sort	$n \log n$	$n \log n$	n^2	in loco + gestione RICORSIONE	OTTIMO in tempo al caso medio
Heapsort	$n \log n$	$n \log n$	$n \log n$	in loco	OTTIMO in tempo e spazio

$\hookrightarrow$ **Heapsort** utilizza una struttura dati specifica: lo **HEAP**.

---

44 Struttura Dati: HEAP (di Massimo)

44.1 Definizione

L'Heap rappresenta un **albero binario quasi completo**.

- **Quasi completo** significa che l'albero è riempito completamente in tutti i livelli, tranne eventualmente l'ultimo.
- Sull'ultimo livello, i nodi sono tutti **accumulati a sinistra**.

44.2 Rappresentazione in Array

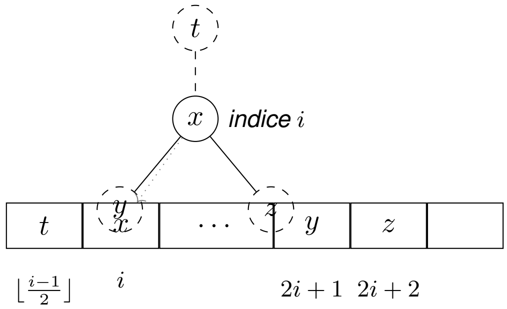
Anche se concettualmente è un albero, l'Heap viene solitamente rappresentato utilizzando un **array**.

- Non serve memorizzare puntatori espliciti a padre e figli.
- La mappatura dall'albero all'array avviene tramite una **visita per livelli** (breadth-first).
- La radice si trova in $A[0]$.
- Gli elementi successivi seguono l'ordine della visita.

Sia $A.heap\_size$ il numero di elementi dell'heap memorizzati in A. Gli elementi validi dell'heap si trovano negli indici da 0 a $A.heap\_size - 1$.

44.3 Regole di Posizionamento (Indici)

Dato un nodo x che corrisponde all'elemento di indice i nell'array A:



Le formule per navigare l'albero muovendosi tra gli indici dell'array sono:

$$\begin{aligned}
\textbf{Parent}(i) &\longrightarrow \text{return } \left\lfloor \frac{i-1}{2} \right\rfloor \\
\textbf{Left}(i) &\longrightarrow \text{return } 2i+1 \\
\textbf{Right}(i) &\longrightarrow \text{return } 2i+2
\end{aligned}$$

45 Proprietà degli Heap: Verifica ed Efficienza

45.1 Correttezza delle Regole di Posizionamento

Possiamo verificare che le formule per navigare l'array siano coerenti. In particolare, applicando la funzione `Parent` al risultato di `Left` o `Right`, dobbiamo tornare al nodo di partenza i.

Verifica. Ricordiamo che Parent$(k) = \lfloor \frac{k-1}{2} \rfloor$.

- **Figlio Sinistro:** $k = 2i + 1$.

$$\text{Parent}(\text{Left}(i)) = \left\lfloor \frac{(2i+1)-1}{2} \right\rfloor = \left\lfloor \frac{2i}{2} \right\rfloor = i$$

- **Figlio Destro:** $k = 2i + 2$.

$$\text{Parent}(\text{Right}(i)) = \left\lfloor \frac{(2i+2)-1}{2} \right\rfloor = \left\lfloor \frac{2i+1}{2} \right\rfloor = \lfloor i + 0.5 \rfloor = i$$

Le regole sono corrette: si "risale" esattamente al genitore. □

45.2 Efficienza in Memoria (Spazio)

Memorizzare l'heap in un vettore di dimensione n è molto più efficiente rispetto a una rappresentazione esplicita con nodi e puntatori.

- **Rappresentazione Array:** Richiede spazio n (solo i dati).
- **Rappresentazione a Puntatori:** Richiederebbe spazio $3n$ (per ogni nodo: il dato + puntatore left + puntatore right + eventuale puntatore parent).

46 Proprietà fondamentali

46.1 Proprietà di Max-Heap

Un **Max-Heap** è un albero binario quasi completo che soddisfa la seguente invariante:

Max-Heap Property

Per ogni nodo i diverso dalla radice ($i > 0$):

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

Ovvero: il valore di un nodo è sempre minore o uguale al valore del padre.

Osservazione

Da questa proprietà deriva che:

1. L'elemento massimo dell'intero heap si trova sempre nella radice ($A[0]$).
2. In ogni sotto-albero, la radice del sotto-albero contiene il valore massimo tra tutti i nodi di quel sotto-albero.

46.2 Altezza e Profondità

È fondamentale distinguere tra altezza e profondità dei nodi:

- **Profondità (Depth) di un nodo:** La lunghezza del cammino (numero di archi) dalla radice al nodo. (La radice ha profondità 0).
- **Altezza (Height) di un nodo:** La lunghezza del cammino più lungo dal nodo a una foglia. (Le foglie hanno altezza 0).
- **Altezza dell'Heap (h):** Corrisponde all'altezza della radice, ovvero la massima distanza dalla radice a una foglia.

Proprietà Matematiche degli Heap

Analizziamo le tre proprietà fondamentali che legano la dimensione dell'input n alla struttura dell'albero.

1. Altezza dell'Heap

Proprietà

Un heap di n elementi ha altezza $h = \lfloor \log_2 n \rfloor$. In notazione asintotica:

$$h = O(\log n)$$

Dimostrazione. Consideriamo i limiti sul numero di nodi n per un albero binario di altezza h:

- **Caso minimo:** L'albero è completo fino al livello $h-1$ e ha una sola foglia al livello h.

$$n \geq 2^h$$

- **Caso massimo:** L'albero è pieno (tutti i livelli completi fino ad h).

$$n \leq 2^{h+1} - 1 < 2^{h+1}$$

Combinando le disuguaglianze otteniamo:

$$2^h \leq n < 2^{h+1}$$

Applicando il logaritmo in base 2:

$$h \leq \log_2 n < h + 1$$

Poiché h deve essere un intero, l'unica soluzione è:

$$h = \lfloor \log_2 n \rfloor$$

□

2. Numero di Foglie

Proprietà

Un heap di n nodi contiene esattamente $\lceil n/2 \rceil$ foglie.

Dimostrazione. Possiamo derivare il numero di foglie sottraendo il numero di **nodi interni** dal totale n. Un nodo i è un nodo interno se ha almeno un figlio (il sinistro). La condizione di esistenza del figlio sinistro è:

$$\text{Left}(i) < n \implies 2i + 1 \leq n - 1$$

Risolvendo per i:

$$2i \leq n - 2 \implies i \leq \frac{n}{2} - 1$$

Essendo i un intero:

$$i \leq \left\lfloor \frac{n}{2} \right\rfloor - 1$$

I nodi interni sono quindi quelli con indice da 0 a $\lfloor n/2 \rfloor - 1$. Il loro numero è $\lfloor n/2 \rfloor$. Il numero di foglie è:

$$\#\text{foglie} = n - \#\text{interni} = n - \left\lfloor \frac{n}{2} \right\rfloor = \left\lceil \frac{n}{2} \right\rceil$$

□

3. Nodi di Altezza h

Proprietà

In un heap di n nodi, ci sono al più:

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil$$

nodi di altezza h.

Caso dell'Albero Pieno - ABCB

Consideriamo un **ABCB** (Albero Binario Completamente Bilanciato), ovvero un heap "pieno" su tutti i livelli. Sia H l'altezza totale dell'albero. Il numero totale di nodi è:

$$n = 2^{H+1} - 1$$

Analizziamo il numero di nodi per ogni altezza h:

- **Altezza $h = 0$ (Foglie):** Circa metà dei nodi sono foglie ($n/2$).
- **Altezza $h = 1$ (Padri delle foglie):** Sopra le foglie c'è un livello con la metà dei nodi rispetto al livello 0 ($n/4$).
- **Altezza generica h:** Generalizzando, il numero di nodi decresce esponenzialmente con l'altezza: $\approx n/2^{h+1}$.

Manutenzione dell'Heap

47 Procedura MAX-Heapify

47.1 Definizione e Scopo

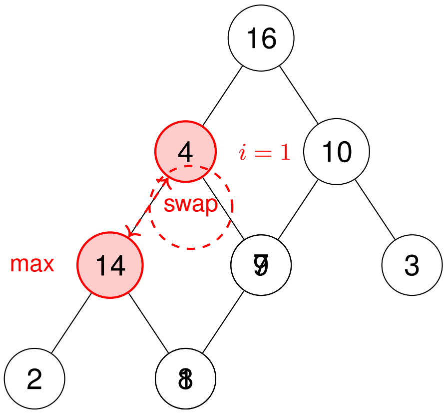
MAX-Heapify

La procedura `MAX-Heapify` è un algoritmo fondamentale utilizzato per ripristinare la proprietà di Max-Heap in un nodo specifico che potrebbe violarla.

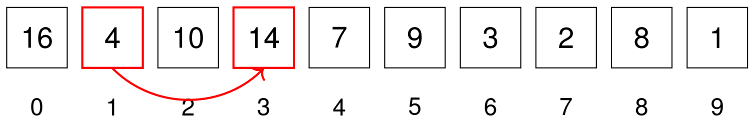
Precondizioni (Ipotesi): Affinché la procedura funzioni correttamente, assumiamo che gli alberi binari radicati in Left(i) e Right(i) siano già dei **Max-Heap**, mentre $A[i]$ potrebbe essere minore dei suoi figli.

47.2 Esempio Grafico

L'indice $i = 1$ (valore 4) viola la proprietà perché è minore del figlio sinistro (14).



Rappresentazione Array:



47.3 Pseudocodice

Algorithm 4 MAX-Heapify(A, i)

Algoritmo

$l \leftarrow$ Left(i)
$r \leftarrow$ Right(i)
$max \leftarrow i$
▷ Controlla se il figlio sinistro esiste ed è maggiore del corrente massimo
If $l < A.heap\_size$ **and** $A[l] > A[max]$ **then**
    $max \leftarrow l$
end If
▷ Controlla se il figlio destro esiste ed è maggiore del corrente massimo
If $r < A.heap\_size$ **and** $A[r] > A[max]$ **then**
    $max \leftarrow r$
end If
▷ Se il massimo non è la radice i, scambia e ricorri
If $max \neq i$ **then**
    swap $A[i] \leftrightarrow A[max]$
    MAX-HEAPIFY(A, max)
end If

47.4 Analisi della Complessità

Costo Temporale

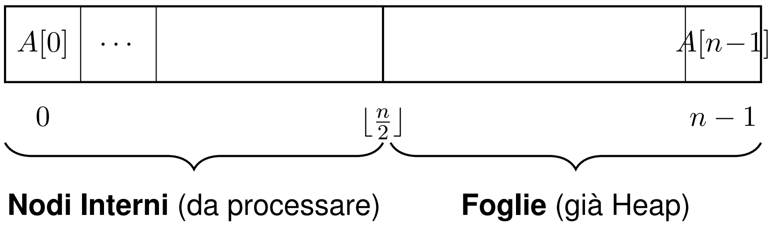
Il costo è proporzionale all'altezza del nodo i, poiché nel caso peggiore il valore scende fino alle foglie.

$$T(n) = O(h) = O(\log n)$$

48 Costruzione dell'Heap (Build-Max-Heap)

48.1 Strategia Bottom-Up

La procedura trasforma un array disordinato in un Max-Heap chiamando `Max-Heapify` a ritroso, dai nodi interni fino alla radice. Le foglie (da $\lfloor n/2 \rfloor$ a $n-1$) sono già heap validi.



48.2 Pseudocodice

Algorithm 5 Build-Max-Heap(A)

Algoritmo

$A.heap\_size \leftarrow A.length$
For $i \leftarrow \lfloor \frac{A.length}{2} \rfloor - 1$ **downto** 0 **do**
  MAX-HEAPIFY(A, i)
end For

49 Analisi della Complessità di Build-Max-Heap

49.1 Analisi Accurata

Il costo totale non è $O(n \log n)$, ma **lineare** $O(n)$. Il costo totale $T(n)$ è la somma dei costi per ogni nodo, che dipendono dall'altezza h.

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} (\text{nodi di altezza } h) \times O(h)$$

Sostituendo il numero massimo di nodi $\lceil n/2^{h+1} \rceil$:

$$T(n) \leq \frac{c \cdot n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}$$

La serie $\sum \frac{h}{2^h}$ converge a 2. Pertanto:

$$T(n) \leq \frac{c \cdot n}{2} \cdot 2 = O(n)$$

Correttezza

49.2 Invariante di ciclo

All'inizio dell'iterazione del ciclo for...

50 Analisi del Costo in Tempo

Limite superiore: $n/2$ chiamate di MAX-HEAPIFY...

Parte XII

Lezione 29/1 - 3/2: Pile e Code

51 Pile e Code: Insiemi Dinamici

Sono insiemi dinamici in cui l'elemento rimosso dall'operazione di cancellazione, o inserito dall'operazione di inserimento, è **PREDETERMINATO**.

Organizzazione Logica

- **PILA (Stack)**: Politica **LIFO** (Last In First Out).
- **CODA (Queue)**: Politica **FIFO** (First In First Out).

Implementazione: ARRAY o LISTE.

52 Pile (Stacks)

Le operazioni possibili (Query e Modifica) sono:

- `ISEMPTY`: dice se la pila è vuota.
- `TOP`: lettura dell'elemento in cima alla pila (immutata).
- `PUSH`: inserimento (in cima).
- `POP`: cancellazione (dalla cima).

52.1 Implementazione su Array

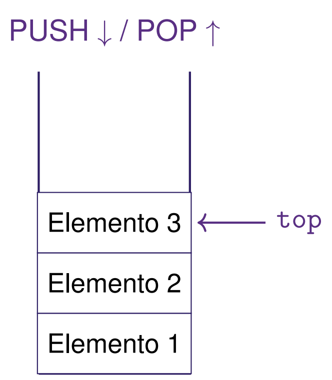
Algoritmo

```
Procedure IsEmpty(PILA, top)
    If top < 1 then Return TRUE
    ElseReturn FALSE
    end If                                   ▷ Complessità Costante Θ(1)
end Procedure

Procedure Top(PILA, top)
    If IsEmpty(PILA, top) then Return error
    ElseReturn PILA[top]
    end If
end Procedure

Procedure Push(PILA, top, x)
    top ← top + 1
    If top > PILA.length then Return error
    Else
        PILA[top] ← x
    end If
end Procedure

Procedure Pop(PILA, top)
    If IsEmpty(PILA, top) then Return error
    Else
        x ← PILA[top]
        top ← top − 1 Return x
    end If                                   ▷ Complessità Costante Θ(1)
end Procedure
```

Gestione Stack (Array)

Tutte le operazioni lavorano sull'indice `top` in tempo costante $\Theta(1)$.

- **Push**: Prima incrementa, poi scrive (Prefix).
- **Pop**: Legge, poi decrementa (Postfix logico).



Stack (LIFO)

52.2 Implementazione su Lista

Algoritmo

```
Procedure IsEmpty(topEl)
    If topEl == nil then Return TRUE
    ElseReturn FALSE
    end If
end Procedure

Procedure Top(topEl)
    If IsEmpty(topEl) then Return error
    ElseReturn topEl.key
    end If
end Procedure

Procedure Push(topEl, x)
    x.next ← topEl
    topEl ← x
end Procedure

Procedure Pop(topEl)
    If IsEmpty(topEl) then Return error
    end If
    VAL ← topEl.key
    topEl ← topEl.next Return VAL
end Procedure
```

Gestione Stack (Lista)

Le operazioni avvengono sempre sulla **testa** della lista (`topEl`):

- **Push**: `x.next = topEl; topEl = x` (Inserimento in testa).
- **Pop**: `topEl = topEl.next` (Rimozione in testa).

Complessità: Sempre Costante.

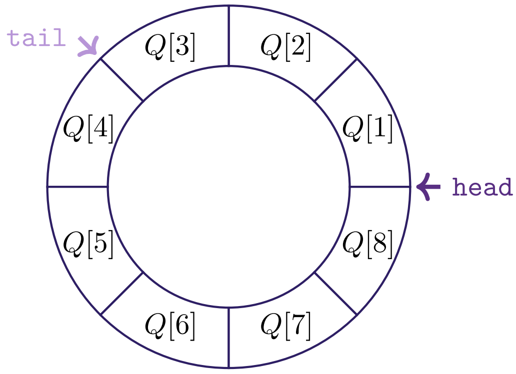
53 Code (Queues)

53.1 Possibili Query e Operazioni

- `ISFULL` (ARRAY ONLY).
- `ISEMPTY`: dice se la coda è vuota.
- `FIRST`: lettura dell'elemento in testa alla coda (immutata).
- `ENQUEUE`: inserimento.
- `DEQUEUE`: cancellazione.

53.2 Implementazione su Array (Gestione Circolare)

- `head`: indice dell'elemento in testa.
- `tail`: indice della locazione in cui inserire il prossimo elemento.



Algoritmo

Function IsEmpty(A, head, tail) **Return** $(head == tail)$
end Function

Function IsFull(A, head, tail) **Return** $(head == tail + 1)$ ▷ N.B. array circolare
end Function

Function First(A, head, tail)
    If IsEmpty(A, head, tail) **then Return error**
    ElseReturn $A[head]$
    end If
end Function

Procedure Enqueue(A, head, tail, x)
    If IsFull(A, head, tail) **then Return error**
    end If
    $A[tail] \leftarrow x$
    $tail \leftarrow (tail + 1)\% A.length$ ▷ $\Theta(1)$
end Procedure

Procedure Dequeue(A, head, tail)
    If IsEmpty(A, head, tail) **then Return error**
    end If
    $x \leftarrow A[head]$
    $head \leftarrow (head + 1)\% A.length$ **Return** x ▷ $\Theta(1)$
end Procedure

Coda Circolare

L'array "si morde la coda" grazie all'operatore modulo:

$$next\_idx = (curr\_idx + 1)\% capacity$$

Questo evita di dover shiftare gli elementi dopo una Dequeue.

53.3 Implementazione su Lista

Algoritmo

```
Function IsEmpty(head) Return (head == nil)
end Function

Function First(head)
    If IsEmpty(head) then Return error
    ElseReturn head.key
    end If
end Function

Procedure Enqueue(head, tail, x)
    If IsEmpty(head) then
        head ← x
    Else
        tail.next ← x
    end If
    tail ← x
    x.next ← nil
end Procedure

Procedure Dequeue(head, tail)
    If IsEmpty(head) then Return error
    end If
    VAL ← head.key
    If head == tail then
        tail ← nil
    end If
    head ← head.next Return VAL                ▷ Θ(1)
end Procedure
```

Coda su Lista

- **Enqueue**: Avviene su `tail`. Necessario aggiornare il puntatore `next` della vecchia coda e spostare `tail`.
- **Dequeue**: Avviene su `head`. Caso speciale: se la coda diventa vuota, bisogna mettere a NULL anche `tail`.

Entrambe Complessità Costante.

Parte XIII

Lezione 1/12: Heapsort e Code di Priorità

54 Build-Max-Heap

Algoritmo

```
Procedure BUILD-MAX-HEAP(A, n)
    A.hs ← n
    For i = ⌊n/2⌋ − 1 downto 0 do
        MAX-HEAPIFY(A, i)
    end For
end Procedure
```

Logica di Costruzione (Bottom-Up)

Partendo dall'ultimo nodo interno ($\lfloor n/2 \rfloor - 1$) fino alla radice, chiamiamo `Max-Heapify`. Le foglie sono già heap validi, quindi non serve processarle.

54.1 Analisi di Complessità

- **Limite Superiore**: $n/2$ chiamate di `Max-Heapify` (costo $O(\log n)$) $\to T(n) = O(n \log n)$.
- **Limite Stretto (Corretto)**: $T(n) = O(n)$.

54.2 Correttezza

Invariante: All'inizio di ogni iterazione del ciclo for, ogni nodo $i+1, i+2, \dots, n-1$ è radice di un max-heap.

55 Heapsort

Algoritmo

```
Procedure HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    For i = n − 1 downto 1 do
        scambia A[0] con A[i]
        A.hs ← A.hs − 1
        MAX-HEAPIFY(A, 0)
    end For
end Procedure
```

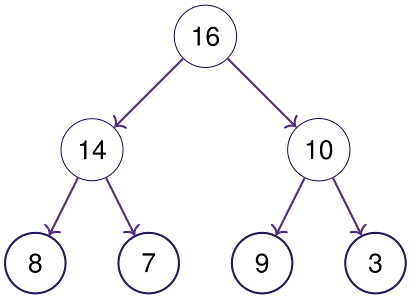
Fasi dell'Heapsort

1. **Costruzione**: Si trasforma l'array in un Max-Heap.
2. **Estrazione**: Si scambia la radice (massimo) con l'ultimo elemento e si riduce la dimensione dell'heap.
3. **Ripristino**: Si chiama `Max-Heapify` sulla nuova radice per far "affondare" il valore scambiato.

Costo Totale: $T(n) = O(n \log n)$.

55.1 Esempio Grafico (Heap)

Esempio su Array: $A = [16, 14, 10, 8, 7, 9, 3]$.



Max-Heap:
Ogni padre $\geq$ figli.
Radice = Max Assoluto.

56 Code di Priorità

Mantiene un insieme di elementi con chiavi (key, priorità).

56.1 Operazioni

- `Insert(S, x)`: $S = S \cup \{x\}$.
- `Heap-Max(A)`: restituisce l'elemento massimo, $O(1)$.
- `Heap-Extract-Max(A)`: rimuove e restituisce il massimo, $O(\log n)$.
- `Heap-Increase-Key(A, i, k)`: aumenta il valore della chiave del nodo i a k, $O(\log n)$.
- `Max-Heap-Insert(A, key)`: inserisce una nuova chiave, $O(\log n)$.

Parte XIV

Lezione 17: Alberi Binari: Bilanciamento e Proprietà Avanzate

57 Schema Generale di Ricorsione su Alberi

Quando si affrontano problemi su alberi binari che richiedono di calcolare una proprietà o un valore basato sulla struttura dell'albero, si utilizza spesso uno schema ricorsivo standard basato sulla proprietà di **decomponibilità** del problema.

Schema di Risoluzione Decomponibile

L'algoritmo segue questi passi fondamentali:

1. **Caso Base:** Si gestisce l'albero vuoto (o foglia), restituendo un valore neutro o specifico per la chiusura della ricorrenza (es. 0, null, true).
2. **Passo Induttivo (Divide):** Si effettuano le chiamate ricorsive sui sottoalberi sinistro (`u.left`) e destro (`u.right`).
3. **Combinazione (Conquer):** Si combinano i risultati ottenuti dai sottoalberi con le informazioni del nodo corrente (`u`) per ottenere il risultato finale.
4. **Restituzione:** Si restituisce l'esito al chiamante (caso generale).

57.1 Implementazione Generica e Complessità

Di seguito lo pseudocodice generico per una funzione `DECOMP(u)` che opera su un nodo `u`.

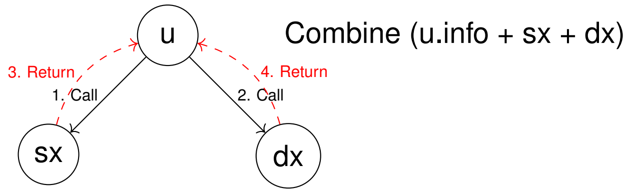
Pseudocodice: DECOMP(u)

```
Funzione DECOMP(u)
    // 1. Caso Base
    IF (u == NULL) THEN
        RETURN ValoreBase;

    // 2. Passo Induttivo
    RisSx = DECOMP(u.left);  // Ricorsione a sinistra
    RisDx = DECOMP(u.right); // Ricorsione a destra

    // 3. Combinazione
    RETURN RICOMBINA(RisSx, RisDx, u.info);
```

Analisi dello Schema Ricorsivo

Questo "template" è universale per problemi decomponibili:

- **Discesa (Pre-order)**: Si scende fino alle foglie (null).
- **Risalita (Post-order)**: I risultati parziali (`RisSx`, `RisDx`) tornano dai figli e vengono combinati nel nodo corrente.



Complessità Computazionale. Poiché l'algoritmo visita ogni nodo esattamente una volta (come una visita post-order), la complessità temporale è lineare rispetto al numero di nodi n:

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

—

58 Alberi Binari Completamente Bilanciati (ABCB)

Un concetto fondamentale è la distinzione tra un albero completo e uno completamente bilanciato.

Definizioni Tassonomiche

- **Albero Binario Completo:** Un albero binario è *completo* se ogni nodo ha esattamente 0 o 2 figli (nessun nodo ha grado 1).
- **Albero Binario Completamente Bilanciato (ABCB):** È un albero binario che è **sia completo**, sia ha tutte le **foglie allo stesso livello**.

58.1 Proprietà Matematiche degli ABCB

Dato un ABCB di altezza h (dove l'altezza è il numero di archi dalla radice alla foglia più profonda, oppure definita in base ai nodi come h_{nodi}), valgono le seguenti relazioni notevoli:

- **Numero di foglie:** 2^h.
- **Numero di nodi interni:** $2^h - 1$.
- **Numero totale di nodi (n):** Somma di foglie e interni:

$$n = 2^h + (2^h - 1) = 2^{h+1} - 1$$

- **Relazione Altezza-Nodi:** Invertendo la formula:

$$n + 1 = 2^{h+1} \implies \log_2(n+1) = h + 1 \implies h = \log_2(n+1) - 1$$

  Pertanto, l'altezza è logaritmica: $h(n) = \Theta(\log n)$.

58.2 Algoritmo di Verifica ABCB

Vogliamo scrivere un algoritmo che restituisca `TRUE` se un albero è ABCB, `FALSE` altrimenti.

58.2.1 Approccio 1: Naive (Inefficiente)

Un primo approccio controlla ricorsivamente se i sottoalberi sono ABCB e se hanno la stessa altezza.

$$\text{Check}(u) = \text{ABCB}(u.left) \wedge \text{ABCB}(u.right) \wedge (H(u.left) == H(u.right))$$

Questo approccio è inefficiente perché ricalcola l'altezza $H(u)$ ripetutamente per ogni nodo.

58.2.2 Approccio 2: Ottimizzato (Lineare)

Per ottenere una complessità $\Theta(n)$, dobbiamo calcolare il bilanciamento e l'altezza in un'unica visita (bottom-up). La funzione restituisce una coppia di valori `<bool bilanciato, int altezza>`.

Algoritmo: ABCB2(u) $\rightarrow< bool, int >$

```
ABCB2(u):
    // Caso Base: albero vuoto è bilanciato, altezza -1
    IF (u == NULL) THEN
        RETURN <TRUE, -1>;

    // Chiamate Ricorsive
    <bil_s, alt_s> = ABCB2(u.left);
    <bil_d, alt_d> = ABCB2(u.right);

    // Logica di Combinazione:
    // 1. Sottoalberi devono essere bilanciati (bil_s && bil_d)
    // 2. Altezze devono essere uguali (alt_s == alt_d)
    is_balanced = bil_s AND bil_d AND (alt_s == alt_d);

    // Calcolo nuova altezza corrente
    current_height = 1 + max(alt_s, alt_d);

    RETURN <is_balanced, current_height>;
```

Strategia Bottom-Up $O(n)$

Invece di ricalcolare l'altezza separatamente ($O(n^2)$), la funzione restituisce **due valori**: 1. Il booleano di bilanciamento (AND logico dei figli). 2. L'altezza corrente (aggiornata in risalita). Questo permette di visitare ogni nodo una sola volta.

Complessità: Lineare, $\Theta(n)$, poiché esegue un attraversamento post-order costante per nodo.

—

59 Nodi Cardine

Un problema algoritmico interessante riguarda l'identificazione di nodi che soddisfano una specifica proprietà geometrica all'interno dell'albero.

Definizione: Nodo Cardine

Un nodo u di un albero binario si dice **CARDINE** se e solo se la sua profondità (P_u) è uguale alla sua altezza (h_u).

$$u \text{ è Cardine} \iff P_u = h_u$$

Nota: P_u è la distanza dalla radice, h_u è la distanza dalla foglia più profonda nel suo sottoalbero.

59.1 Strategia Risolutiva

Per verificare questa condizione in modo efficiente ($\Theta(n)$), dobbiamo:

1. Passare la profondità p **scendendo** nell'albero (parametro in input).
2. Calcolare l'altezza **risalendo** dall'albero (valore di ritorno).
3. Verificare la condizione $p == altezza$ nel post-order.

Algoritmo: CARDINE(u, p)

```
// Input: u (nodo corrente), p (profondità corrente, inizialmente 0)
// Output: altezza del sottoalbero radicato in u

INT CARDINE(Node u, int p)
    // 1. Caso Base
    IF (u == NULL) THEN
        RETURN -1;

    // 2. Discesa ricorsiva (aumento profondità)
    int alt_s = CARDINE(u.left, p + 1);
    int alt_d = CARDINE(u.right, p + 1);

    // 3. Calcolo Altezza (fase di risalita)
    int \texttt{my\_alt} = max(alt_s, alt_d) + 1;

    // 4. Verifica proprietà Cardine
    IF (p == \texttt{my\_alt}) THEN
        PRINT(u.key);

    RETURN \texttt{my\_alt};
```

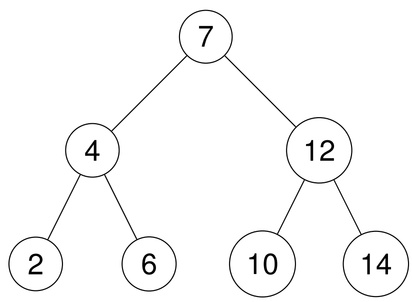
Doppio Flusso Informativo

Qui combiniamo due direzioni:

- **Input (Top-down)**: La profondità `p` viene passata dal padre al figlio incrementandola.
- **Output (Bottom-up)**: L'altezza `my_alt` viene calcolata dalle foglie risalendo.
- La verifica `p == my_alt` avviene nel momento di incontro (post-order).

59.2 Esempio di Traccia (Trace)

Consideriamo l'albero tracciato nei sorgenti.



Analisi per alcuni nodi (supponendo struttura bilanciata dall'esempio):

- **Nodo 7 (Radice):** Profondità $P = 0$, Altezza $h = 2$. $(0 \neq 2) \rightarrow$ No.
- **Nodo 4:** Profondità $P = 1$, Altezza $h = 1$. $(1 = 1) \rightarrow$ **CARDINE**.
- **Nodo 12:** Profondità $P = 1$, Altezza $h = 1$. $(1 = 1) \rightarrow$ **CARDINE**.
- **Foglie (2, 6, 10, 14):** Profondità $P = 2$, Altezza $h = 0$. $(2 \neq 0) \rightarrow$ No.

—

60 Esercizi per Casa

60.1 Nodi Centrali

Problema: Progettare un algoritmo che, dato un Albero Binario, stampi le chiavi dei suoi **Nodi Centrali**.

Definizione: Nodo Centrale

Un nodo u è detto **CENTRALE** se la dimensione del sottoalbero di cui è radice (numero di nodi) è uguale alla somma delle chiavi dei nodi che appartengono al percorso dalla radice al nodo stesso.

$$Size(u) = \sum_{v \in \text{Path}(root \to u)} v.key$$

Suggerimento per la soluzione: L'algoritmo deve combinare due flussi di informazioni, similmente all'esercizio sui Nodi Cardine: 1. **Top-down (Parametro):** La somma delle chiavi dalla radice al padre + chiave corrente. 2. **Bottom-up (Return):** La dimensione del sottoalbero (1 + size(sx) + size(dx)). 3. **Verifica:** Nel passo di post-order, confrontare i due valori.

Abbozzo Soluzione: CENTRALE(u, somma_cammino)

```
CENTRALE(u, somma_corrente):
    IF u == NULL RETURN 0

    nuova_somma = somma_corrente + u.key

    size_sx = CENTRALE(u.left, nuova_somma)
    size_dx = CENTRALE(u.right, nuova_somma)

    my_size = 1 + size_sx + size_dx

    IF (my_size == nuova_somma) PRINT u.key

    RETURN my_size
```

Logica Nodi Centrali

Simile ai nodi cardine, ma accumulando la somma delle chiavi:

- `somma_corrente` scende (Pre-order).
- `my_size` (dimensione sottoalbero) risale (Post-order).

Parte XV

Lezione 18: Strutture Dati Lineari

61 Liste (Linked Lists)

Le liste rappresentano una struttura dati fondamentale per la gestione di collezioni dinamiche di elementi.

Definizione e Struttura

Una lista è una sequenza di nodi dove ogni elemento è **virtualmente concatenato** ma non necessariamente contiguo in memoria. Non è necessario conoscere la dimensione a priori (numero di elementi necessari).
La struttura di base di un nodo comprende:

- Un campo dati (informazione).
- Un campo `NEXT` che punta all'elemento successivo.

La lista è accessibile tramite un puntatore alla testa (`HEAD`).

61.1 Tipologie e Vantaggi

- **Liste Semplici:** Collegamento unidirezionale (`NEXT`).
- **Liste Doppie:** Ogni nodo possiede due puntatori: uno al successivo (`NEXT`) e uno al precedente (`PREC`).

Vantaggi rispetto agli Array: Le liste sono molto comode per effettuare inserimenti ed eliminazioni in posizioni intermedie, operazioni che in un array risulterebbero scomode (richiedendo lo shift degli elementi).

HEAD → | Dati | Next | → | Dati | Next | → | Dati | $\emptyset$ |

—

62 Pile (Stack)

La Pila è un insieme dinamico gestito con politica **LIFO (Last In First Out)**: si entra e si esce solo dalla "cima".

Operazioni Fondamentali

Una pila supporta le seguenti operazioni, tutte con complessità $\Theta(1)$:

- **PUSH:** Aggiunge un elemento in cima alla pila.
- **POP:** Rimuove l'elemento in cima alla pila.
- **QUERY:**
  - `ISEMPTY`: Verifica se la pila è vuota.
  - `TOP`: Restituisce il valore dell'elemento in cima senza rimuoverlo.

62.1 Implementazione con Array

Struttura: PILA $= \langle A, \text{TOP} \rangle$.

- `TOP`: indice che varia tra 0 e $n-1$. Vale -1 se la pila è vuota.
- **Svantaggio:** Bisogna definire a priori la dimensione massima ($A.length$).

Algoritmi Pila (Array)

ISEMPTY(PILA)

```
IF (TOP < 0) THEN RETURN TRUE
ELSE RETURN FALSE
```

TOP(PILA)

```
IF ISEMPTY(PILA) THEN error
ELSE RETURN A[TOP]
```

PUSH(PILA, X)

```
TOP = TOP + 1
IF (TOP >= A.length) THEN error (Overflow)
A[TOP] = X
```

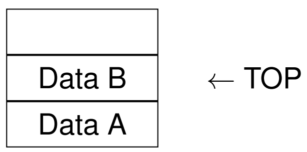
POP(PILA)

```
IF ISEMPTY(PILA) THEN error (Underflow)
ELSE
    X = A[TOP]
    TOP = TOP - 1
    RETURN X
```

Dettagli Implementativi (Stack Array)

1. **TOP**: Mantiene l'indice dell'ultimo elemento inserito.
2. **PUSH**: Incrementa prima `TOP`, poi scrive. Se `TOP` supera la capacità, si ha *Overflow*.
3. **POP**: Legge l'elemento a `TOP` e poi decrementa. Se `TOP` diventa negativo, lo stack è vuoto (Underflow).

Stack (Array)



62.2 Implementazione con Lista

Struttura gestita tramite un puntatore `TOPER` che indica la testa della lista (cima della pila). Non ha limiti di dimensione fissa.

Algoritmi Pila (Lista)

ISEMPTY(TOPER)

```
IF (TOPER == NULL) THEN RETURN TRUE
ELSE RETURN FALSE
```

PUSH(TOPER, X) (X è il nuovo nodo)

```
X.next = TOPER
TOPER = X
```

POP(TOPER)

```
IF ISEMPTY(TOPER) THEN error
val = TOPER.key
TOPER = TOPER.next
RETURN val
```

Dettagli Implementativi (Stack Lista)

- Non esiste limite di dimensione (se non la memoria).
- **PUSH**: Inserimento in testa ($O(1)$). Il nuovo nodo punta alla vecchia testa.
- **POP**: Rimozione in testa ($O(1)$). Si aggiorna `TOPER` al successivo.

—

63 Code (Queue)

La Coda è un insieme dinamico gestito con politica **FIFO (First In First Out)**: si entra solo dalla coda (`tail`) e si esce solo dalla testa (`head`).

63.1 Implementazione con Array Circolare

Struttura: CODA $= \langle A, \text{head}, \text{tail} \rangle$. L'array è trattato come circolare usando l'aritmetica modulare.

Gestione Indici Circolari

Gli indici avanzano modulo la lunghezza dell'array ($A.length$). Esempio calcolo indici:

$$(6+1) \pmod 7 = 0$$

$$(3+1) \pmod 7 = 4$$

Algoritmi Coda (Array Circolare) - $\Theta(1)$

ISEMPTY(CODA)

```
RETURN (head == tail)
```

FIRST(CODA)

```
IF ISEMPTY(CODA) THEN error
ELSE RETURN A[head]
```

ENQUEUE(CODA, X)

```
// Controllo Overflow (Coda piena se tail+1 tocca head)
IF (head == (tail + 1) % A.length) THEN error
A[tail] = X
tail = (tail + 1) % A.length
```

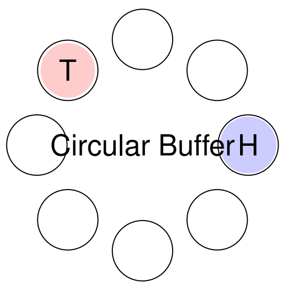
DEQUEUE(CODA)

```
IF ISEMPTY(CODA) THEN error
ELSE
    X = A[head]
    head = (head + 1) % A.length
    RETURN X
```

Logica Coda Circolare

L'uso del **modulo** (%) permette di riutilizzare gli spazi liberi all'inizio dell'array quando la coda avanza.

- `tail`: Punta alla prima cella *libera*.
- `head`: Punta al primo elemento valido.
- **Coda Piena**: Se avanzando `tail` si raggiunge `head` ($tail + 1 == head$).



63.2 Implementazione con Lista

Manteniamo due puntatori: `head` (per estrazione) e `tail` (per inserimento).

Algoritmi Coda (Lista)

ISEMPTY(head, tail)

```
IF (head == NULL) THEN RETURN TRUE
ELSE RETURN FALSE
```

ENQUEUE(head, tail, X)

```
IF ISEMPTY(head, tail) THEN
    head = X
ELSE
    tail.next = X
tail = X  // Aggiornamento della coda (implicito ma necessario)
```

DEQUEUE(head, tail)

```
IF ISEMPTY(head, tail) THEN error
ELSE
    val = head.key
    IF (head == tail) THEN tail = NULL // Caso svuotamento coda
    head = head.next
    RETURN val
```

Gestione coda con Lista

Manteniamo due puntatori per garantire operazioni $O(1)$:

- **ENQUEUE**: Inserisce in coda (`tail`). Aggiorna `tail.next` e poi `tail`.
- **DEQUEUE**: Rimuove dalla testa (`head`). Se la lista si svuota, bisogna aggiornare anche `tail` a NULL.

Parte XVI
Lezione 19: Alberi Binari (Approfondimenti)

64 Il Nodo Centrale

Questa sezione analizza un problema specifico sugli Alberi Binari (AB).

Definizione: Nodo Centrale

Dato un Albero Binario (AB) T, un nodo u non vuoto è detto **CENTRALE** se:

La dimensione del sottoalbero di cui u è radice (numero di nodi)
È PARI A
La somma delle chiavi dei nodi che appartengono al percorso dalla radice dell'albero al nodo u stesso.

64.1 Algoritmo Risolutivo

Il problema richiede di progettare un algoritmo che stampi le chiavi di tutti i nodi centrali.

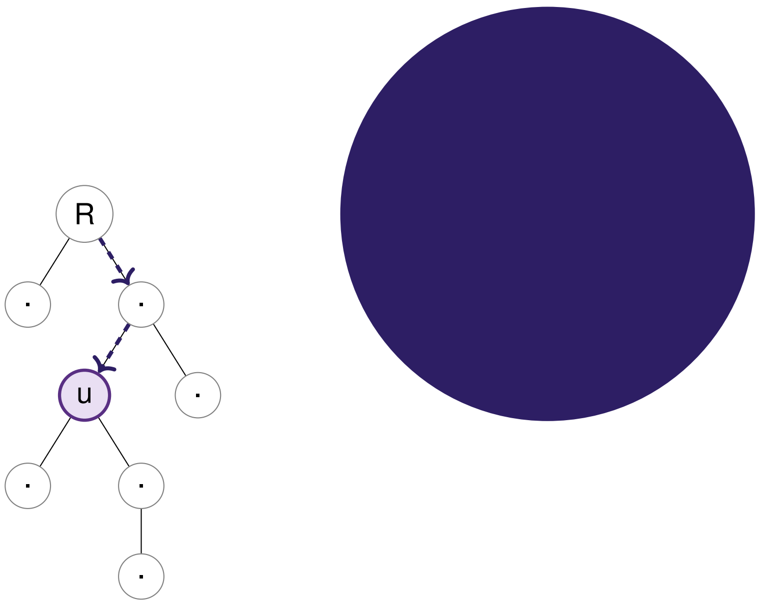


Figura 2: Rappresentazione logica del Nodo Centrale

Pseudocodice: CENTRALI(u, SUM)

Idea: Usiamo una visita *posticipata* (post-order). La funzione restituisce la dimensione del sottoalbero al chiamante, ma internamente verifica la condizione di centralità usando il parametro accumulatore SUM.

Algoritmo

Function CENTRALI(u, SUM)
    If $u ==$ nil **then** ▷ Caso Base: Albero vuoto
        Return 0
    end If
    Visita Ricorsiva (Posticipata)
    $dim_{sx} \leftarrow$ CENTRALI($u.left, SUM + u.key$)
    $dim_{dx} \leftarrow$ CENTRALI($u.right, SUM + u.key$)
    Calcolo Dimensione Locale
    $dim_u \leftarrow dim_{sx} + dim_{dx} + 1$
    Verifica Condizione
    If $dim_u == (SUM + u.key)$ **then**
        PRINT(u.key) ▷ Stampa il nodo centrale
    end If
    Return dim_u ▷ Restituisce la dimensione al padre
end Function

65 Visite degli Alberi

Analisi delle visite (Anticipata, Simmetrica, Posticipata) su un albero specifico tratto dagli appunti.

65.1 Albero di Esempio e Tracce

Ricostruzione dell'albero basata sulle tracce delle visite presenti nel manoscritto (Pag. 4).

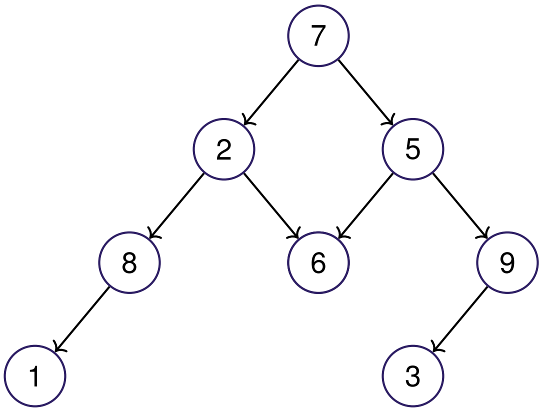


Figura 3: **Struttura dell'Albero**: Radice 7, Figli 2 e 5.

1. **Visita Anticipata (Pre-Order)**
   Ordine: Radice $\rightarrow$ Sinistra $\rightarrow$ Destra.
   Sequenza: $7, 2, 8, 1, 4, 5, 6, 9, 3$
2. **Visita Simmetrica (In-Order)**
   Ordine: Sinistra $\rightarrow$ Radice $\rightarrow$ Destra.
   Sequenza: $1, 8, 2, 4, 7, 6, 5, 3, 9$
3. **Visita Posticipata (Post-Order)**
   Ordine: Sinistra $\rightarrow$ Destra $\rightarrow$ Radice.
   Sequenza: $1, 8, 4, 2, 6, 3, 9, 5, 7$

66 Conteggio Foglie

Algoritmo ricorsivo per contare le foglie di un albero binario.

Pseudocodice: CONTAFOGLIE(u)

Algoritmo

```
Function ContaFoglie(u)
    If u == nil then                                    ▷ Albero vuoto
        Return 0
    end If
                                ▷ È una foglia se entrambi i figli sono nil
    If (u.left == nil) ∧ (u.right == nil) then
        Return 1
    end If
                            ▷ Passo Ricorsivo: somma foglie sx + foglie dx
    Return ContaFoglie(u.left) + ContaFoglie(u.right)
end Function
```

Analisi Complessità

$$T(n) = O(n)$$

L'algoritmo visita ogni nodo esattamente una volta. La relazione è $n = l + r$ (nodi totali = foglie + nodi interni).

67 Verifica Albero Completo

L'algoritmo seguente verifica se un albero è "Completo" secondo la definizione data negli appunti (ogni nodo deve avere 0 o 2 figli). In letteratura questo è spesso noto come *Albero Strettamente Binario*.

Pseudocodice: COMPLETO(u)

Restituisce `TRUE` se la proprietà è verificata, `FALSE` altrimenti.

Algoritmo

Function COMPLETO(u)
  If $u ==$ nil **then** ▷ Vuoto è completo
    Return true
  end If
▷ Se foglia, ok
  If $(u.left ==$ nil$) \wedge (u.right ==$ nil$)$ **then**
    Return true
  end If
▷ Controllo figli spaiati (XOR logico)
  If $(u.left ==$ nil$) \neq (u.right ==$ nil$)$ **then**
    Return false ▷ Ha un solo figlio
  end If
  Return COMPLETO(u.left) $\wedge$ COMPLETO(u.right)
end Function

68 Esercizi (Vecchio Compitino)

68.1 Problema: Chiave Doppia del Padre

Testo: Scrivere un algoritmo che restituisca il numero delle foglie la cui chiave è il doppio di quella del genitore $(u.key == 2 \cdot u.p.key)$.

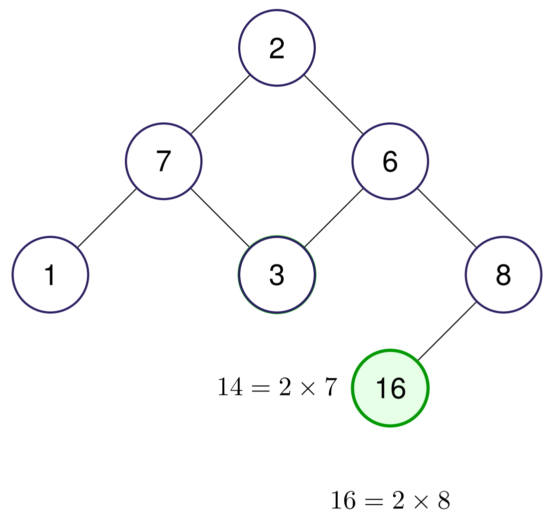


Figura 4: Esempio: Nodi foglia (evidenziati) che soddisfano la condizione.

Soluzione: CONTA(u)

Algoritmo

```
Function CONTA(u)
    If u == nil then
        Return 0
    end If
    If (u.left == nil) ∧ (u.right == nil) then        ▷ Se è foglia
        If (u.p ≠ nil) ∧ (u.key == 2 ∗ u.p.key) then
            Return 1
        Else
            Return 0
        end If
    end If
    Return CONTA(u.left) + CONTA(u.right)
end Function
```

68.2 Problema: Stampa Chiave e Profondità

Progettare un algoritmo che stampi chiave e profondità di ciascun nodo. Si suggerisce una visita anticipata passando la profondità come parametro incrementale: ‘Stampa(u.left, depth+1)‘.

Parte XVII

Errori Comuni e Note

69 Errori Comuni da Evitare nell'Analisi

Questa sezione riassume gli errori più frequenti che si possono commettere nell'analisi della complessità e nell'uso della notazione asintotica.

69.1 Errore nel Calcolo della Complessità Totale

Uno degli errori più comuni riguarda la combinazione delle complessità quando diverse fasi di un algoritmo vengono eseguite in sequenza.

69.1.1 Moltiplicare Invece di Sommare

L'errore comune consiste nel **moltiplicare** le complessità delle diverse fasi, anziché **sommarle**, quando queste sono eseguite in sequenza.

Errore di Moltiplicazione

Si consideri un algoritmo composto da tre fasi eseguite consecutivamente:

1. Fase 1: $O(n^2)$
2. Fase 2: $O(n \log n)$
3. Fase 3: $O(n)$

Calcolo Errato: Complessità totale $\neq O(n^2) \cdot O(n \log n) \cdot O(n) = O(n^4 \log n)$.

Regola Fondamentale

Quando le operazioni sono eseguite **in sequenza** (l'una dopo l'altra), il tempo totale è la somma dei tempi. La complessità finale è data dal termine dominante (quello con l'ordine di grandezza maggiore):

$$T_{totale}(n) = T_1(n) + T_2(n) + T_3(n)$$

Calcolo Corretto:

$$O(n^2) + O(n \log n) + O(n) = \mathbf{O(n^2)}$$

La moltiplicazione delle complessità si applica unicamente quando le operazioni sono **annidate** (es. un ciclo iterativo interno a un altro ciclo).

69.2 Imprecisioni Terminologiche sulle Strutture Dati

69.2.1 Definire un Array "Disordinato"

È un errore usare l'aggettivo "disordinato" per descrivere lo stato di una struttura dati (come un array). La caratteristica dell'ordine è una proprietà booleana.

Nota

Non si deve dire che l'array è "disordinato". Si deve dire che l'array **non è ordinato**.

69.3 Errori di Definizione sulle Notazioni Asintotiche ($\mathrm{O}, \Theta, \Omega$)

Le notazioni asintotiche (O-grande, Theta, Omega) non definiscono un'uguaglianza tra funzioni, ma una **relazione di limitazione** del tasso di crescita asintotico.

69.3.1 Confondere l'Appartenenza con l'Eguaglianza

Nota

È un errore affermare che Θ = equazione (es. $\Theta = n^2$). La notazione non è un'equazione in senso stretto e non rappresenta una singola funzione.

La scrittura $f(n) = O(g(n))$ non indica un'uguaglianza, ma significa che la funzione $f(n)$ **appartiene all'insieme** delle funzioni che crescono al più come $g(n)$ (a meno di una costante per n sufficientemente grande).

Sintesi Notazioni

Sia $g(n)$ una funzione di riferimento.

- $\mathbf{O(g(n))}$ (**O-grande**): Indica il **limite superiore** (Worst Case). Una funzione $f(n)$ è $O(g(n))$ se $f(n)$ cresce al più velocemente di $g(n)$.
- $\mathbf{\Omega(g(n))}$ (**Omega**): Indica il **limite inferiore** (Best Case). Una funzione $f(n)$ è $\Omega(g(n))$ se $f(n)$ cresce almeno tanto velocemente quanto $g(n)$.
- $\mathbf{\Theta(g(n))}$ (**Theta**): Indica l'**ordine esatto** (Average Case o limite sia superiore che inferiore). Una funzione $f(n)$ è $\Theta(g(n))$ se $f(n)$ cresce esattamente con lo stesso tasso di $g(n)$.

69.3.2 Errore nella Spiegazione dei Simboli

È un errore comune spiegare in modo confuso o invertito le limitazioni date dalle tre notazioni.

Osservazione

Quando si spiega $f(n) = O(g(n))$, non significa che $f(n)$ sia limitata da $g(n)$ nel senso di una frazione con un risultato specifico. Significa che esiste una costante positiva c e un n_0 tali che:

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

La definizione richiede che $f(n)$ sia **asintoticamente dominata** da $g(n)$, a meno di un fattore costante.