Cheatsheet di Algoritmi e Strutture Dati

Giacomo Scampini

15 luglio 2025

1 Complessità

1.1 Notazioni di Complessità Asintotica in Elenco

- $f(n) = O(g(n))$ - **O grande** - Limite asintotico superiore
- $f(n) = \Omega(g(n))$ - **Omega grande** - Limite asintotico inferiore
- $f(n) = \Theta(g(n))$ - **Theta grande** - Limite asintotico sia superiore che inferiore

1.2 Confronto Tramite Limiti

Dato il limite $L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$:

- Se $L = 0$, allora $\Theta(f(n)) < \Theta(g(n))$.
- Se $L = c$ (con $c \neq 0, \infty$), allora $\Theta(f(n)) = \Theta(g(n))$.
- Se $L = \infty$, allora $\Theta(f(n)) > \Theta(g(n))$.

1.3 Gerarchia Fondamentale degli Ordini di Grandezza

Per costanti $k, h \in \mathbb{R}^+$ e $a > 1$:

$$\Theta(1) < \Theta((\log n)^k) < \Theta(n^h) < \Theta(a^n) < \Theta(n!) < \Theta(n^n)$$

1.4 Classi di Complessità Comuni

- $\mathcal{O}(1)$: Costante (es. accesso a un elemento di un array)
- $\mathcal{O}(\log n)$: Logaritmica (es. ricerca binaria)
- $\mathcal{O}(n)$: Lineare (es. scansione di una lista)
- $\mathcal{O}(n \log n)$: Lineare-logaritmica (es. merge sort, heapsort)
- $\mathcal{O}(n^2)$: Quadratica (es. bubble sort, selection sort)
- $\mathcal{O}(2^n)$: Esponenziale (es. problemi risolti con la forza bruta)
- $\mathcal{O}(n!)$: Fattoriale (es. problema del commesso viaggiatore con forza bruta)

2 Automi e TM

2.1 Complessità degli Automi

- **DFSA (Automa a Stati Finiti Deterministico)**
  - Complessità Temporale: $T_A(n) = \Theta(n)$
  - Complessità Spaziale: $S_A(n) = \Theta(1)$
- **DPDA (Automa a Pila Deterministico)**
  - Complessità Temporale: $T_A(n) = \Theta(n)$
  - Complessità Spaziale: $\Theta(0) \leq \Theta(S_A(n)) \leq \Theta(n)$
- **k-DTM (Macchina di Turing Deterministica a k-nastri)**
  - Complessità Temporale: Nessun limite generale. Per calcolarla devi immaginare il funzionamento della macchina.
  - Complessità Spaziale: $\Theta(S_M(n)) \leq \Theta(T_M(n))$
- **SDTM (Macchina di Turing Deterministica a nastro singolo)**
  - Complessità Temporale: Nessun limite generale. Per calcolarla devi immaginare il funzionamento della macchina.
  - Complessità Spaziale: $S_M(n) = \Omega(n)$, ciò significa che la complessità spaziale dev'essere almeno lineare, questo perché il nastro di input coincide con il nastro di memoria.

TIP: per il calcolo della complessità spaziale, ricordati di considerare il caso peggiore. Il caso peggiore può anche essere per una stringa che non viene accettata, ovvero $x \notin L$.

2.2 Contatori (Implementati su DTM)

- **Complessità Spaziale**: per contare fino a m, sono necessari $\Theta(\log m)$ simboli. Se m dipende dalla lunghezza dell'input n, la complessità spaziale diventa $S_M(n) = \Theta(\log n)$.
- **Complessità Temporale** (per eseguire n incrementi/decrementi):
  - $T(n) = \Theta(n)$: se ad ogni modifica vengono visitate solo le cifre necessarie (userai questo in sede d'esame).
  - $T(n) = \Theta(n \log n)$: se ad ogni modifica si visitano tutte le cifre del contatore.

3 RAM

3.1 Complessità delle RAM

3.1.1 Criteri di Costo

- **Costo Costante:** Ogni istruzione ha costo 1. Ogni cella di memoria ha costo 1, indipendentemente dal valore contenuto.
- **Costo Logaritmico:** Il costo di un'operazione e dello spazio occupato dipende dalla dimensione (logaritmo) dei valori numerici coinvolti.

$$l(x) := \begin{cases} \lfloor \log_2 x \rfloor + 1 & \text{se } x \neq 0 \\ 1 & \text{se } x = 0 \end{cases} \quad \text{N.B.: } l(x) = \Theta(\log x)$$

- **Quando sceglierli:** I due criteri sono equivalenti se la dimensione degli operandi è limitata da una costante. Se i numeri possono diventare arbitrariamente grandi, il criterio logaritmico è più realistico.

3.1.2 Calcolo del Costo Logaritmico (caso semplificato)

Sotto l'ipotesi di usare un numero costante di celle di memoria:

- **Costo Spaziale:** Lo spazio totale è la somma della "lunghezza" (logaritmo) di tutti i numeri più grandi salvati in ogni cella di memoria utilizzata. E' sempre $\Theta(\log i)$, dove i è il numero che viene calcolato in quell'istante.
- **Gestione di un intero i** (es. `LOAD, STORE, READ, WRITE, JZ`)
  - Costo Temporale: $\Theta(\log i)$
- **Operazioni Aritmetiche** (su operandi n_1, n_2)
  - Addizione (+), Sottrazione (-): $\Theta(\log n_1 + \log n_2)$
  - Moltiplicazione (*), Divisione (/): $\Theta(\log n_1 \cdot \log n_2)$

Come si calcola in generale il costo temporale in caso di costo algoritmico? In generale si prende l'operazione nell'ultimo istante, che può essere magari una somma o una moltiplicazione, e con una sommatoria si somma tutto. Usi l'approssimazione di Stirling per calcolare la complessità.

4 Equazioni di Ricorrenza

4.1 Algoritmo Divide et Impera

Un algoritmo che segue la strategia *divide et impera* si articola in tre fasi:

- **Dividi:** Il problema è scomposto in sottoproblemi più semplici della stessa forma.
- **Impera:** I sottoproblemi vengono risolti ricorsivamente.
- **Combina:** Le soluzioni dei sottoproblemi sono combinate per ottenere la soluzione del problema originale.

La sua complessità temporale $T(n)$ è descritta da un'equazione di ricorrenza, tipicamente nella forma $T(n) = a \cdot T(n/b) + f(n)$.

4.2 Metodi di Risoluzione delle Ricorrenze

4.2.1 Risoluzione Diretta Esplicita

Consiste nello sviluppare iterativamente la ricorrenza fino a individuare un modello generale per esprimerne l'ordine di grandezza.

4.2.2 Metodo di Sostituzione

Consiste nel formulare un'ipotesi per la soluzione e nel verificarla rigorosamente tramite dimostrazione per induzione.

4.2.3 Metodo dell'Albero di Ricorsione

È una tecnica visuale per sviluppare le chiamate ricorsive e sommarne i costi livello per livello. È utile per formulare un'ipotesi di soluzione, da verificare poi con il metodo di sostituzione. Esempio di albero:

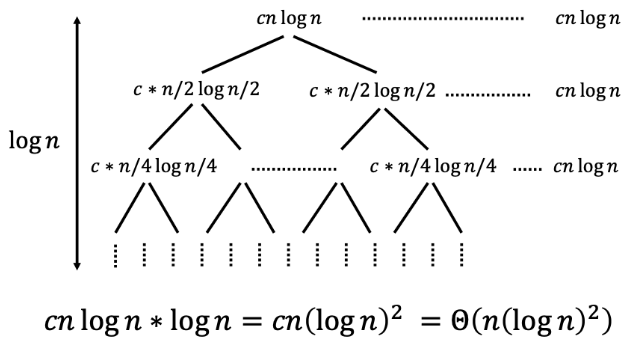

Figura 1: Esempio di un albero di ricorsione.

4.2.4 Metodo per Ricorrenze Lineari

Si applica a ricorrenze della forma $T(n) = \sum_{i=1}^{h} a_i T(n-i) + cn^k$. Posto $a := \sum a_i$, la soluzione (come limite superiore) è:

- $T(n) = O(n^{k+1})$ se $a = 1$.
- $T(n) = O(n^k a^n)$ se $a \geq 2$.

4.2.5 Teorema Master

Fornisce una soluzione "pronta" per ricorrenze della forma $T(n) = a \cdot T(n/b) + f(n)$ (con $a \geq 1, b > 1$). Si confronta $f(n)$ con $n^{\log_b a}$:

- **Caso 1:** Se $f(n) = O(n^{\log_b a - \epsilon})$ per qualche $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$.
- **Caso 2:** Se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \log n)$.
- **Caso 3:** Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche $\epsilon > 0$ e se $f(n)$ soddisfa una condizione di regolarità, allora $T(n) = \Theta(f(n))$.

Corollario del Teorema Master (Caso Polinomiale)

Una versione semplificata del teorema si applica quando $f(n)$ è un polinomio della forma $\Theta(n^k)$. Data la ricorrenza $T(n) = a \cdot T(n/b) + \Theta(n^k)$:

- **Caso 1:** Se $k < \log_b a$, allora $T(n) = \Theta(n^{\log_b a})$.
- **Caso 2:** Se $k = \log_b a$, allora $T(n) = \Theta(n^k \log n)$.
- **Caso 3:** Se $k > \log_b a$, allora $T(n) = \Theta(n^k)$.

5 Pseudocodice

5.1 Sintassi di base

- **Riga di commento**

```
//...
```

- **Assegnamento**

```
i := j
```

- **Operazioni**

```
+, -, *, /, %
```

- **Confronto di interi**

```
>, <, >=, <=, =, !=
```

- **Lettura dell'input**

```
x := read()
```

- **Restituzione in output**

```
return x
```

5.2 Istruzioni comuni

- **If-else**

```
if condizione
    istruzioni
else
    istruzioni
```

- **Cicli**

```
while condizione
    istruzioni

for i := n_1 to n_2
    istruzioni
```

5.3 Oggetti e variabili

- I dati composti sono organizzati come oggetti. Gli oggetti hanno attributi (campi):
  - `x.attr` è il valore dell'attributo `attr` dell'oggetto `x`.
- Gli array sono oggetti, dotati dell'attributo `length`.
  - $A[j]$ è l'elemento di indice j dell'array A.
  - $A[i..j]$ è il sottoarray di A dall'i-esimo al j-esimo elemento.
- Una variabile che corrisponde ad un oggetto è un puntatore all'oggetto.
  - Dopo le istruzioni $y := x, x.attr := 3$ si ha $y.attr = x.attr = 3$.
- Un puntatore che non fa riferimento ad alcun oggetto ha valore NIL.
- Usa l'istruzione $ALLOCATE(varname, length)$ per creare un nuovo array.

6 Algoritmi di Ordinamento

6.1 Algoritmi comuni

Algoritmo	Complessità temporale	Complessità spaziale
Insertion sort	$\Theta(n^2)$	$\Theta(1)$
Merge sort	$\Theta(n \log n)$	$\Theta(n)$
Heapsort	$\Theta(n \log n)$	$\Theta(1)$
Quicksort	$\Theta(n^2)$	$\Theta(1)$
Counting sort	$\Theta(n + k)$	$\Theta(k)$

6.2 Come Riconoscere la Complessità Logaritmica

Un algoritmo ha complessità logaritmica quando il problema si riduce in modo esponenziale a ogni passo. I principali indizi nel codice sono:

- **La variabile del ciclo moltiplica o divide.**
  - L'aggiornamento non è un'addizione/sottrazione (es. `i := i + 1`).
  - L'aggiornamento è una moltiplicazione o divisione per una costante $c > 1$ (es. `i := i * 2` oppure `i := i / 2`).
  - La variabile "salta" verso il valore finale, invece di "camminare". Questo richiede $\Theta(\log n)$ passi.
- **Lo spazio del problema viene ridotto di una frazione costante.**
  - L'algoritmo scarta una porzione significativa dei dati a ogni iterazione (es. metà, un terzo, etc.).
  - L'esempio classico è la Ricerca Binaria, che dimezza lo spazio di ricerca a ogni passo.
  - La ricorrenza associata è spesso nella forma $T(n) = T(n/b) + O(1)$, la cui soluzione è $\Theta(\log n)$.
- **Caso speciale ($\log \log n$): la variabile esegue un "super-salto".**
  - La variabile di controllo viene elevata a una potenza, tipicamente al quadrato (es. `i := i * i`).
  - La crescita è doppiamente esponenziale, portando a una complessità ancora minore di $\Theta(\log \log n)$.

7 Strutture Dati

7.1 Vettori (Array)

- `A.length` = lunghezza dell'array.
- `A[i]` = accesso a elemento `i` dell'array.
- `A[i..j]` = sottoarray da `i` a `j`.
- `n` = dimensione dell'array che è uguale a `A.length`.

7.2 Matrici

- `M.height` = numero di righe.
- `M.width` = numero di colonne.
- `M[i][j]` = accesso a riga i colonna j.
- `n` = dimensione dell'input che è uguale al numero di elementi, ovvero $M.\texttt{height} \times M.\texttt{width}$.
- $n := M.\texttt{size}$ per una matrice quadrata, dove `size` è il numero di righe (o colonne).

7.3 Liste Concatenate

- `L.head` = puntatore alla testa della lista.
- `x_f.next = NIL`, dove `x_f` è l'ultimo elemento della lista.

7.3.1 Liste Singolarmente Concatenate

- `x.key` = dato contenuto nell'elemento x.
- `x.next` = puntatore all'elemento successivo.

7.3.2 Liste Doppiamente Concatenate

- `x.key` = dato contenuto nell'elemento x.
- `x.next` = puntatore all'elemento successivo.
- `x.prev` = puntatore all'elemento precedente.
- `L.head.prev = NIL`.

7.3.3 Liste Doppiamente Concatenate Circolari

- Utilizzano un nodo speciale detto **sentinella** (`L.nil`) al posto di `L.head`.
- `L.nil.key = NIL`.
- `L.nil.next` punta alla testa della lista.
- `L.nil.prev` punta alla coda della lista.
- La lista è "circolare": la `prev` della testa e la `next` della coda puntano a `L.nil`.
- Se la lista è vuota, `L.nil` punta a se stesso.

7.4 Tabelle Hash

- `T` = array di `m` celle (slot) che memorizza i dati.
- `h(k)` = funzione hash che mappa una chiave `k` a un indice della tabella.
- $\alpha = n/m$ = fattore di carico, definito come rapporto tra elementi e slot.

7.4.1 Scelta della dimensione m

- Per il **metodo della divisione** ($h(k) = k \pmod m$): scegliere `m` come un **numero primo** non troppo vicino a una potenza di 2.
- Per l'**indirizzamento aperto** ($h(k, i) = (h_1(k) + i \cdot h_2(k)) \pmod m$): se $h_2(k)$ è sempre dispari, scegliere `m` come una **potenza di 2**.

7.4.2 Risoluzione delle Collisioni

- **Concatenamento (Chaining)**: Ogni cella della tabella `T[j]` punta a una lista concatenata di tutti gli elementi la cui chiave ha valore hash `j`. Le operazioni di inserimento, cancellazione e ricerca operano sulla lista corrispondente.
- **Indirizzamento Aperto (Open Addressing)**: Tutti gli elementi sono memorizzati nella tabella stessa. Per inserire un elemento, si esamina (ispeziona) una sequenza di slot fino a trovarne uno vuoto.
  - **Ispezione Lineare**: La sequenza di ispezione è data da $h(k,i) = (h'(k)+i) \pmod m$ per $i = 0, 1, \ldots, m-1$.
  - **Ispezione Quadratica**: La sequenza di ispezione è data da $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \pmod m$ per $i = 0, 1, \ldots, m-1$, con c_1, c_2 costanti ausiliarie.
  - **Double Hashing**: La sequenza di ispezione è data da $h(k,i) = (h_1(k) + i \cdot h_2(k)) \pmod m$ per $i = 0, 1, \ldots, m-1$, dove h_1 e h_2 sono funzioni hash ausiliarie.

7.4.3 Complessità (Hashing Uniforme)

- Numero medio di tentativi per accesso (indirizzamento aperto): $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
- Valore atteso del numero di collisioni (concatenamento): $E[Y] = \frac{n(n-1)}{2m}$

7.5 Alberi

7.5.1 Alberi Binari di Ricerca (BST)

- `T.root` = puntatore alla radice dell'albero.
- `x.key` = chiave del nodo `x`.
- `x.p` = puntatore al nodo padre.
- `x.left` = puntatore al figlio sinistro.
- `x.right` = puntatore al figlio destro.
- `x.leftsize` = (opzionale) dimensione del sottoalbero sinistro del nodo.

7.5.2 Alberi di Ricerca Generici (GST)

- `T.root` = puntatore alla radice dell'albero.
- `x.key` = chiave del nodo `x`.
- `x.p` = puntatore al nodo padre.
- `x.fs` = puntatore al figlio più a sinistra (first son).
- `x.lb` = puntatore al fratello a sinistra (left brother).
- `x.rb` = puntatore al fratello a destra (right brother).

7.5.3 Alberi Rosso-Neri (RBT)

- Un RBT è un BST con attributi e proprietà aggiuntive.
- `T.nil` = nodo speciale **sentinella** che sostituisce i puntatori a `NIL`. Il suo colore è sempre `BLACK`.
- `x.color` = attributo di ogni nodo che può essere `RED` o `BLACK`.
- `bh(x)` = **altezza nera** del nodo, ovvero il numero di nodi neri in ogni cammino da `x` (escluso) a `T.nil` (incluso).

Proprietà RB

- Ogni nodo è **rosso o nero**.
- La **radice è nera** (`T.root.color = BLACK`).
- Ogni foglia (il nodo sentinella `T.nil`) è **nera**.
- Se un nodo è rosso, allora entrambi i suoi **figli sono neri**.
- Per ogni nodo, tutti i cammini semplici da quel nodo alle foglie discendenti contengono lo **stesso numero di nodi neri**.

7.6 Grafi

8 Algoritmi BST GST RBT

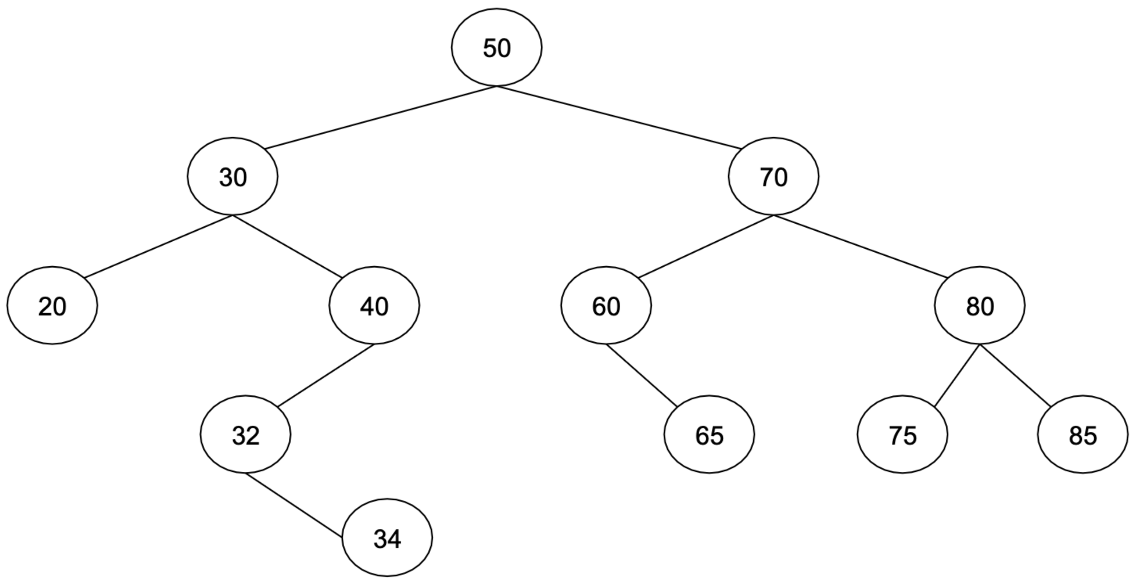

Figura 2: BST di esempio.

- **Attraversamento In-Ordine (In-order Traversal)**, complessità: $\Theta(n)$
  - **Descrizione:** Questo algoritmo visita un albero binario di ricerca (BST) processando prima il sottoalbero sinistro, poi la radice e infine il sottoalbero destro. Il risultato è la stampa delle chiavi dei nodi in ordine non decrescente.
  - **Esempio:** Dato un albero con le chiavi disposte come nell'esempio del documento, la sequenza di output dell'attraversamento in-ordine è:

    `20, 30, 32, 34, 40, 50, 60, 65, 70, 75, 80, 85`

- **Attraversamento Pre-Ordine (Pre-order Traversal)**, complessità: $\Theta(n)$
  - **Descrizione:** L'attraversamento anticipato (o pre-ordine) visita prima la radice, poi ricorsivamente il sottoalbero sinistro e infine ricorsivamente il sottoalbero destro.
  - **Esempio:** Utilizzando lo stesso albero di riferimento, l'output dell'attraversamento pre-ordine è:

    `50, 30, 20, 40, 32, 34, 70, 60, 65, 80, 75, 85`

- **Attraversamento Post-Ordine (Post-order Traversal)**, complessità: $\Theta(n)$
  - **Descrizione:** L'attraversamento posticipato (o post-ordine) visita ricorsivamente prima il sottoalbero sinistro, poi il sottoalbero destro e infine la radice.
  - **Esempio:** Per lo stesso albero, la sequenza di output generata è:

    `20, 34, 32, 40, 30, 65, 60, 75, 85, 80, 70, 50`

- **Attraversamento per Livelli (Level-order Traversal)**, complessità: $\Theta(n)$
  - **Descrizione:** Questo algoritmo visita i nodi dell'albero livello per livello, da sinistra a destra, partendo dalla radice. Si avvale di una coda per tenere traccia dei nodi da visitare: la radice viene inserita in coda, e poi, in un ciclo, il nodo in testa alla coda viene rimosso, la sua chiave stampata e i suoi figli (se esistenti) vengono aggiunti alla coda.
  - **Esempio:** Per l'albero di riferimento, l'output dell'attraversamento per livelli sarebbe:

    `50, 30, 70, 20, 40, 60, 80, 32, 65, 75, 85, 34`

9 Progetta Strutture Dati

9.0.1 Complessità Liste Concatenate

	Unsorted, Singly linked	Sorted, Singly linked	Unsorted, Doubly linked	Sorted, Doubly linked
SEARCH (L,k)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT (L,x)	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
DELETE (L,x)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
SUCCESSOR (L,x)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
PREDECESSOR (L,x)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
MAXIMUM (L)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
MINIMUM (L)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$