Appunti di Algoritmi e Strutture Dati

Luca Facchini
Matricola: 245965

Corso tenuto dal prof. Montresor Alberto
Università degli Studi di Trento

A.A. 2024/2025

Sommario

Appunti del corso di Algoritmi e Strutture Dati tenuto dal prof. Montresor Alberto presso l'Università degli Studi di Trento nell'anno accademico 2024/2025.

Indice

Capitolo 1

Analisi di Algoritmi

1.1 Modelli di calcolo

1.1.1 Definizioni

Complessità

Definizione 1.1. La **complessità** di un algoritmo è definita come la quantità di **tempo** necessaria per eseguirlo in funzione della **dimensione dell'input**.

Le domande spontanee che ci si pone sono dunque:

- Come definire la dimensione dell'input?
- Come misurare il tempo?

Dimensione dell'input

Definizione 1.2. Per definire la **dimensione dell'input** abbiamo due criteri:

Costo Logaritmico Il costo logaritmico è definito come il numero di bit necessari per rappresentare l'input.

Costo Uniforme La taglia dell'input è definita come il numero di elementi da cui è composto.

Ma in molti casi possiamo assumere che tutti gli elementi siano rappresentati dallo stesso numero di bit costante e che coincidono a meno di costante moltiplicativa

Tempo

Definizione 1.3. Un'istruzione si considera elementare se può essere eseguita in tempo "costante" dal processore, dunque un esempio ne è la moltiplicazione o una funzione matematica ad esempio $\cos(d)$, ma una istruzione come il massimo tra due numeri non è elementare.

Modello di calcolo

Definizione 1.4. Un modello di calcolo è definito come la rappresentazione astratta di un calcolatore che rispetta i seguenti criteri:

Astrazione Il modello deve permettere di nascondere i dettagli.

Realismo Il modello deve riflettere una situazione reale.

Potenza Matematica Il modello deve permettere di dimostrare "formalmente" la complessità di un algoritmo.

Esempio di modello di calcolo è la **Macchina di Turing**.

1.1.2 Esempi di Analisi

Tempo di calcolo min()

Sappiamo che ogni istruzione richiede un tempo costante per essere eseguita e che ogni operazione potenzialmente ha una costante diversa dalle altre e che ogni istruzione viene eseguita un numero di volte diversa dalle altre.

```
ITEM min(ITEM[] A, int n)
                                          Costo   # Volte
ITEM min = A[1]                           c1      1
for i = 2 to n do                         c2      n
    if A[i] < min then                    c3      n − 1
        min = A[i]                        c4      n − 1
return min                                c5      1
```

Otteniamo quindi che il tempo di calcolo è:

$$\begin{aligned} T(n) =& c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5 \\ =& (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4) = an + b \end{aligned}$$

Tempo di calcolo di binarySearch()

In questo algoritmo il vettore viene suddiviso in due parti: Parte SX: $\lfloor (n-1)/2 \rfloor$ e Parte DX: $\lfloor n/2 \rfloor$.

```
int binarySearch(ITEM[] A, ITEM v, int i, int j)
                                                          Costo                  # (i > j)  # (i ≤ j)
if i > j then                                             c1                     1          1
    return 0                                              c2                     1          0
else
    int m = ⌊(i + j)/2⌋                                   c3                     0          1
    if A[m] = v then                                      c4                     0          1
        return m                                          c5                     0          0
    else if A[m] < v then                                 c6                     0          1
        return binarySearch(A, v, m + 1, j)               c7 + T(⌊n/2⌋)          0          0/1
    else
        return binarySearch(A, v, i, m − 1)               c7 + T(⌊(n − 1)/2⌋)    0          1/0
```

A questo punto dobbiamo fare delle assunzioni:

- Assumiamo che la n potenza di 2 sia: $n = 2^k$.
- L'elemento cercato non è presente.
- Ad ogni passo andiamo sempre a destra in quanto il numero di elementi da valutare è maggiore: $n/2$.

possiamo ora suddividere il problema in due casistiche:

$$\begin{aligned} i > j \quad (n = 0) \quad T(n) =& c_1 + c_2 \\ i \leq j \quad (n > 0) \quad T(n) =& T(n/2) + c_1 + c_2 + c_3 + c_4 + c_6 + c_7 \\ =& T(n/2) + d \end{aligned}$$

unendo i due casi otteniamo la **Relazione di ricorrenza**:

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(n/2) + d & \text{se } n > 0 \end{cases}$$

ottenuta la relazione generalmente per un numero n di elementi otteniamo che il tempo è dato da:

$$\begin{aligned} T(n) =& T(n/2) + d \\ =& T(n/4) + 2d \\ & \dots \\ =& T(1) + kd \\ =& T(0) + (k+1)d \\ =& kd + (c+d) \qquad = d \log n + e \end{aligned}$$

ottenendo quindi che il tempo di calcolo è $O(\log n)$ di natura logaritmica.

1.1.3 Ordini di Complessità

$f(n)$	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	**Tipo**
$\log n$	3	6	9	13	Logaritmica
$\sqrt{n}$	3	10	31	100	sub-lineare
n	10	100	1000	10000	Lineare
$n \log n$	30	664	9965	132877	log-lineare
n^2	10^2	10^4	10^6	10^8	Quadratica
n^3	10^3	10^6	10^9	10^{12}	Cubica
2^n	1024	10^{30}	10^{301}	10^{3010}	Esponenziale

1.2 Notazione asintotica

1.2.1 Notazioni O, Ω, Θ

Notazione O

Definizione 1.5. Sia $g(n)$ una funzione di costo; indichiamo con $O(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0,\ \exists m \geq 0 : f(n) \leq cg(n),\ \forall n \geq m$$

La seguente notazione si legge: $f(n)$ è "O grande" (big O) di $g(n)$, con un abuso di notazione si scrive $f(n) = O(g(n))$[1] . Inoltre per la precedente definizione possiamo dire che $g(n)$ è un **limite asintotico superiore** per $f(n)$, in quanto dopo qualche valore m la funzione $g(n)$ è sempre maggiore di $f(n)$. Inoltre per questo motivo sappiamo che $f(n)$ cresce al più come $g(n)$.

Notazione Ω

Definizione 1.6. Sia $g(n)$ una funzione di costo; indichiamo con $\Omega(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0,\ \exists m \geq 0 : f(n) \geq cg(n),\ \forall n \geq m$$

La seguente notazione si legge: $f(n)$ è "Omega" di $g(n)$, con un abuso di notazione si scrive $f(n) = \Omega(g(n))$[1]. Inoltre per la precedente definizione possiamo dire che $g(n)$ è un **limite asintotico inferiore** per $f(n)$, in quanto dopo qualche valore m la funzione $g(n)$ è sempre minore di $f(n)$. Inoltre per questo motivo sappiamo che $f(n)$ cresce almeno come $g(n)$.

[1] Questo è un abuso di notazione in quanto $O(g(n))$ è una classe di funzioni e non può essere eguagliata una singola funzione, il simbolo più appropriato sarebbe $f(n) \in O(g(n))$

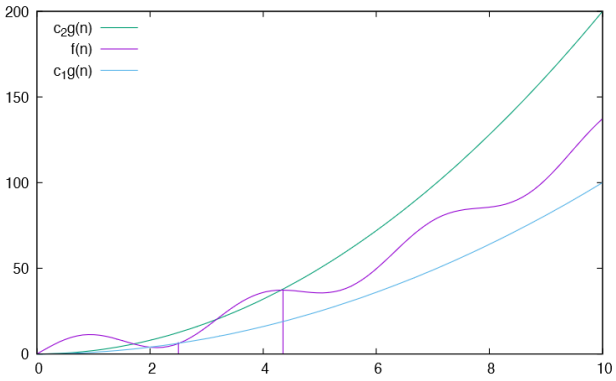
Notazione Θ

Definizione 1.7. Sia $g(n)$ una funzione di costo; indichiamo con $\Theta(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c_1 > 0,\ \exists c_2 > 0,\ \exists m \geq 0 : c_1 g(n) \leq f(n) \leq c_2 g(n),\ \forall n \geq m$$

La seguente notazione si legge: $f(n)$ è "Theta" di $g(n)$, con un abuso di notazione si scrive $f(n) = \Theta(g(n))$[1]. Inoltre per la precedente definizione possiamo dire che $f(n)$ cresce esattamente come $g(n)$, detto ciò $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Esempio grafico



1.2.2 Esempi e Esercizi

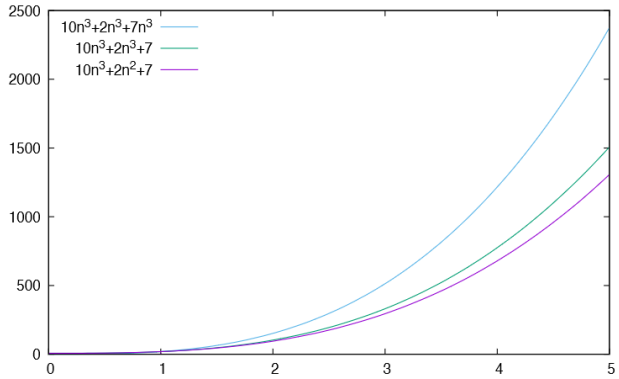
Esempio 1

$$f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} O(n^3)$$

Dobbiamo provare che $\exists c > 0,\ \exists m \geq 0 : f(n) \leq cn^3,\ \forall n \geq m$.

$$\begin{aligned} f(n) =& 10n^3 + 2n^2 + 7 \\ \leq& 10n^3 + 2n^3 + 7 && \forall n \geq 1 \\ \leq& 10n^3 + 2n^3 + n^3 && \forall n \geq \sqrt[3]{7} \\ =& 13n^3 \stackrel{?}{\leq} cn^3 \end{aligned}$$

Che è verificata per qualsiasi $c \geq 13$ e $m \geq \sqrt[3]{7}$, arrotondiamo m ad un intero superiore ottenendo $m = 2$.



1.3 Complessità problemi v/s algoritmi

1.3.1 Moltiplicazione numeri complessi

Per moltiplicare numeri complessi bisogna svolgere la seguente operazione:

$$(a + bi)(c + di) = [ac - bd] + [ad + bc]i$$

dunque dai parametri a, b, c, d otteniamo che il risultato da restituire: $ac - bd$ e $ad + bc$.

Domande Considerando che addizioni e sottrazioni costino $c_1 = 0.01$ e moltiplicazione costi $c_2 = 1$ possiamo chiederci:

- Quanto costa l'algoritmo?
- Si può fare meglio?
- Qual'è il ruolo del modello di calcolo?

Dato che si devono fare almeno 4 moltiplicazioni e 2 somme otteniamo che il costo totale è $4c_2 + 2c_1 = 4.02$.

Ora si può fare meglio? La risposta è no in quanto se si potesse fare meglio si potrebbe fare meglio allora bisognerebbe trovare un algoritmo che esegua meno di 4 moltiplicazioni e 2 somme, ma per fare ciò bisognerebbe cambiare il modello di calcolo.

1.3.2 Sommare numeri binari

Algoritmo elementare della somma - sum()

Ipotizziamo che l'operazione da fare sia la somma di due bit singoli e generare il riporto, assegniamo a questa operazione costo c. Dopo ciò indichiamo con n il numero massimo di bit tra i due numeri da sommare, otteniamo dunque che:

- Richiede di esaminare tutti gli n bit
- Costo totale $cn = O(n)$

Esiste allora un algoritmo più efficiente?

La risposta è no in quanto se esistesse un algoritmo di tale genere allora potremmo cambiare un solo bit di uno dei due numeri e ottenere un risultato diverso senza che l'algoritmo lo vada ad analizzare, il che è impossibile.

1.3.3 Moltiplicare numeri binari

Algoritmo elementare del prodotto - prod()

L'operazione elementare per moltiplicare due numeri binari è la seguente:

							1	0	1	1	1	0	1	$*$
							1	1	0	1	1	1	0	
							0	0	0	0	0	0	0	
						1	0	1	1	1	0	1		
					1	0	1	1	1	0	1			
				1	0	1	1	1	0	1				
			0	0	0	0	0	0	0					
		1	0	1	1	1	0	1						
	1	0	1	1	1	0	1							
1	0	0	1	1	1	1	1	1	1	0	1	1	0	

deduciamo che:

- Dobbiamo accedere a tutti i bit
- Il costo totale è $cn^2 = O(n^2)$ perché dobbiamo fare n somme di n bit.

Soluzione Divide-et-impera

La base del principio **Divide-et-impera** è la seguente:

Divide Dividere il problema in sotto-problemi più piccoli.

Impera risolvi i sotto-problemi in modo ricorsivo.

Combina combina le soluzioni dei sotto-problemi per ottenere la soluzione del problema originale.

La soluzione divide-et-impera per il problema della moltiplicazione binaria è la seguente:

$$\begin{aligned} X =& a \cdot 2^{n/2} + b \\ Y =& c \cdot 2^{n/2} + d \\ XY =& ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd \end{aligned} \qquad \begin{aligned} X =& \overset{a}{\text{Parte SX}} \quad \overset{b}{\text{Parte DX}} \\ Y =& \overset{c}{\text{Parte SX}} \quad \overset{d}{\text{Parte DX}} \end{aligned}$$

Ora possiamo scrivere l'algoritmo: La funzione di costo associata all'algoritmo è:

Algorithm 1 boolean[ ] pdi(boolean[ ] X, boolean[ ] Y, int n)

```
1: if n = 1 then
2:     return X[1] · Y[1]
3: else
4:     spezza X in a e b e Y in c e d
5:     return pdi(a, c, n/2)·2^n + (pdi(a, d, n/2) + pdi(b, c, n/2))·2^{n/2} + pdi(b, d, n/2)
```

$$T(n) = \begin{cases} c_1 & n = 1 \\ 4T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Nota: moltiplicare per 2^n corrisponde a uno shift a sinistra di n posizioni, svolta in tempo lineare.

Ora in quanto abbiamo 4 chiamate ricorsive, assumendo che c_1 sia il tempo per moltiplicare due bit e che c_2 sia il tempo per sommare due numeri binari otteniamo che il tempo di calcolo è $c_2 \cdot 4^i \cdot \frac{n}{2^i} = T(1) \cdot 4^{\log_2 n} = c_1 \cdot n^{\log_2 4} = c_1 \cdot n^2$.

Ma allora è possibile fare meglio? Long story short: si in quanto è stato provato nel 2021 l'esistenza di un algoritmo di complessità $O(n \log n)$.

1.4 Algoritmi di Ordinamento

Introduzione L'obbiettivo di questa sezione è valutare la complessità degli algoritmi in base all'input, in alcuni casi gli algoritmi si comportano differentemente in base all'input se siamo a conoscenza dell'input questa ci consente di scegliere un algoritmo più adeguato alla nostra soluzione.

Come analiziamo gli algoritmi Possiamo analizzare l'efficienza degli algoritmi in base a diversi casi:

Caso Pessimo Questa analisi è la più importante in quanto sappiamo che questa restituisce il limite superiore al tempo di esecuzione qualsiasi sia l'input.

Caso Medio Questa analisi è la più complessa in quanto bisogna definire il "caso medio" e cosa si intende per "medio", ma è utile con una distribuzione uniforme degli input.

Caso Ottimo Utile solo se conosciamo qualcosa sull'input, altrimenti non risulta utile se abbiamo un input arbitrario.

1.4.1 Selection Sort

Algoritmo Selection Sort:

Algorithm 2 selectionSort(Item[ ] A, **int** n)

```
1: for i = 1 to n − 1 do
2:     int min ← min(A, i, n)
3:     A[i] ↔ min(A, i, n)
```

Algoritmo di supporto min: Avendo analizzato il seguente algoritmo notiamo come in ogni caso, ottimo,

Algorithm 3 int min(Item[ ] A, **int** i, **int** n)

```
1: int min ← i
2: for j = i + 1 to n do
3:     if A[j] < A[min] then
4:         min ← j
5: return min
```

medio e pessimo, il "ciclo" esterno della funzione selectionSort() viene eseguito $n-1$ volte, mentre il ciclo interno della funzione min() viene eseguito $n-i$ dove i è il valore dell'iterazione del ciclo esterno, quindi $n-1+n-2+n-3+\ldots+1=\frac{n(n-1)}{2}$ volte, otteniamo quindi che il tempo di calcolo è $O(n^2)$ in quanto questo si può approssimare a $\frac{n^2}{2}$.

$$\sum_{i=1}^{n-1} n-i = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

1.4.2 Insertion Sort

L'algoritmo di insertion sort è efficiente per ordinare piccoli insiemi, il concetto dietro a questo si bassa sull'inserimento dell'elemento preso in analisi al posto giusto.

Algoritmo Insertion Sort:

Algorithm 4 insertionSort(Item[ ] A, **int** n)

```
1: for i = 2 to n do
2:     Item temp ← A[i]
3:     int j ← i
4:     while j > 1 and A[j − 1] > temp do
5:         A[j] ← A[j − 1]
6:         j ← j − 1
7:     A[j] ← temp
```

Il costo di esecuzione non dipende esclusivamente dalla dimensione ma anche dall'ordine degli elementi in ingresso.

Caso Pessimo Il costo dunque nel **caso pessimo** è $O(n^2)$ in quanto vengono eseguiti $n-1$ cicli esterni e $n-1$ cicli interni, ottenendo dunque $(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2}=O(n^2)$.

Caso Medio Nel **caso medio** il costo rimane $O(n^2)$ in quanto il ciclo interni viene eseguito $n/2$ volte, ottenendo dunque $\frac{n(n-1)}{4}=O(n^2)$.

Caso Ottimo Nel **caso ottimo** il costo è $O(n)$ in quanto il ciclo interno non viene mai eseguito.

Questo ci porta a dire che il insertionSort() è un algoritmo di ordinamento utile nei casi in cui l'input è già ordinato o quasi ordinato, nei casi nei quali non conosciamo la natura dell'input è meglio utilizzare un algoritmo di ordinamento differente.

1.4.3 Merge Sort

L'algoritmo di mergeSort() è un algoritmo di ordinamento basato sul principio **divide-et-impera**.

Divide: Spezza virtualmente il vettore di n elementi in sotto-vettori di $n/2$ elementi.

Impera: Chiama mergeSort() ricorsivamente sui due sotto-vettori.

Combina: Unisci (**merge**) i due sotto-vettori ordinati in un unico vettore ordinato.

In input si ha:

- A: vettore di n elementi.
- start, end, mid sono tali che $1 \leq \text{start} < \text{mid} < \text{end} \leq n$.
- I sotto-vettori $A[\text{start}, \ldots, \text{mid}]$ e $A[\text{mid}+1, \ldots, \text{end}]$ sono ordinati.

In output si hanno i due sotto-vettori fusi in un unico sotto-vettore ordinato, tramite un vettore di appoggio B.

Funzione di appoggio Merge:

Algorithm 5 Merge(Item[ ] A, **int** start, **int** end, **int** mid)

```
1:  int i, j, k, h
2:  int i ← start
3:  int j ← mid + 1
4:  int k ← start
5:  while i ≤ mid and j ≤ end do
6:      if A[i] ≤ A[j] then
7:          B[k] ← A[i]
8:          i ← i + 1
9:      else
10:         B[k] ← A[j]
11:         j ← j + 1
12:     k ← k + 1
13: j ← end
14: for h = mid downto i do
15:     A[j] ← A[h]
16:     j ← j − 1
17: for j = start to k − 1 do
18:     A[j] ← B[j]
```

Il costo computazionale di Merge() è $O(n)$, questa è la base del costo computazionale di mergeSort().

Funzione completa mergeSort():

Algorithm 6 mergeSort(Item[ ] A, **int** start, **int** end)

```
1: if start < end then
2:     int mid ← (start + end)/2
3:     mergeSort(A, start, mid)
4:     mergeSort(A, mid + 1, end)
5:     merge(A, start, end, mid)
```

Assumendo per semplificare che $n = 2^k$ dove k è un intero allora l'altezza dell'albero è esattamente $k = \log n$, in questo modo tutti i sotto-vettori hanno dimensione che è potenza di 2. Così facendo il costo computazionale di mergeSort() è:

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + dn & n > 1 \end{cases}$$

dove c è il costo di un'operazione elementare, d è il costo di Merge() e n è il costo di copiare i valori da B a A.

Capitolo 2

Analisi di funzioni

2.1 Notazione asintotica

2.1.1 Definizioni

Si rimanda al sezione 1.2 per le definizioni di O, Ω e Θ.

2.2 Proprietà della notazione asintotica

2.2.1 Regola Generale

Da qui si prende in considerazione la seguente espressione polinomiale:

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0, \quad a_k > 0 \Rightarrow f(n) = \Theta(n^k)$$

Limite Superiore $\exists c > 0, \exists m \geq 0 : f(n) \leq cn^k, \forall n \geq m$

$$\begin{aligned} f(n) =& a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \\ \leq& a_k n^k + |a_{k-1}| \, n^{k-1} + \ldots + |a_1| \, n + |a_0| \\ \leq& a_k n^k + |a_{k-1}| \, n^k + \ldots + |a_1| \, n^k + |a_0| \, n^k \\ =& (a_k + |a_{k-1}| + \ldots + |a_1| + |a_0|) n^k \qquad \forall n \geq 1 \\ \overset{?}{\leq}& cn^k \end{aligned}$$

questa è vera per $c \geq (a_k + |a_{k-1}| + \ldots + |a_1| + |a_0|) > 0$ e per $m = 1$

Limite Inferiore $\exists d > 0, \exists m \geq 0 : f(n) \geq dn^k, \forall n \geq m$

$$\begin{aligned} f(n) =& a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \\ \geq& a_k n^k - |a_{k-1}| \, n^{k-1} - \ldots - |a_1| \, n - |a_0| \\ \geq& a_k n^k - |a_{k-1}| \, n^k - \ldots - |a_1| \, n^k - |a_0| \, n^k \\ =& (a_k - |a_{k-1}| - \ldots - |a_1| - |a_0|) n^k \qquad \forall n \geq 1 \\ \overset{?}{\geq}& dn^k \end{aligned}$$

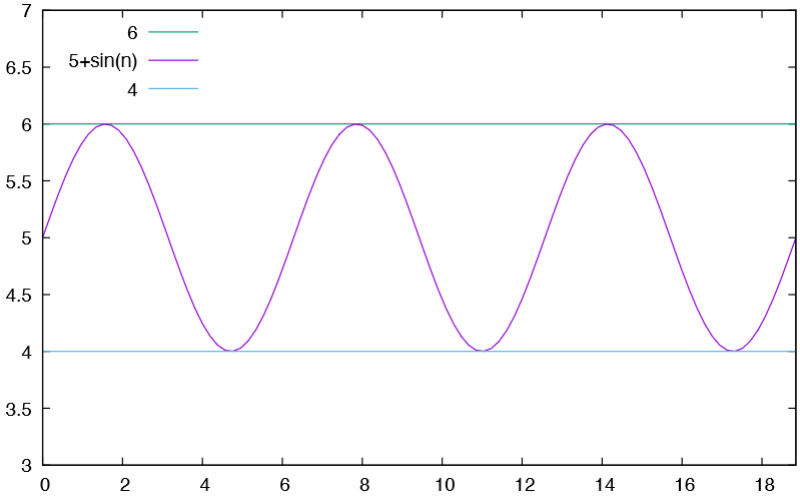
questa è vera se: $d \leq a_k - \frac{|a_{k-1}|}{n} - \ldots - \frac{|a_1|}{n} - \frac{|a_0|}{n} > 0 \Leftrightarrow n > \frac{|a_{k-1}| + \ldots + |a_1| + |a_0|}{a_k}$

Casi Particolari

Complessità di $f(n) = 5$

$$f(n) = 5 \geq c_1 n^0 \Rightarrow c_1 \leq 5$$
$$f(n) = 5 \leq c_2 n^0 \Rightarrow c_2 \geq 5$$
$$\Rightarrow f(n) = \Theta(n^0) = \Theta(1)$$

Complessità di $f(n) = 5+\sin(n)$ La complessità di calcolo di $f(n)$ è $\Theta(1)$, in quanto $\sin(n)$ è una funzione oscillante tra -1 e 1, quindi $5 + \sin(n)$ oscilla tra 4 e 6.



2.2.2 Proprietà delle notazioni

Dualità

$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

Dimostrazione.

$$\begin{aligned} f(n) = O(g(n)) \Leftrightarrow & f(n) \leq cg(n), \forall n \geq m \\ \Leftrightarrow & g(n) \geq \frac{1}{c} f(n), \forall n \geq m \\ \Leftrightarrow & g(n) \geq c' f(n), \forall n \geq m, c' = \frac{1}{c} \\ \Leftrightarrow & g(n) = \Omega(f(n)) \end{aligned}$$

□

Eliminazione di costanti

$$f(n) = O(g(n)) \Leftrightarrow af(n) = O(g(n)), \forall a > 0$$
$$f(n) = \Omega(g(n)) \Leftrightarrow af(n) = \Omega(g(n)), \forall a > 0$$

Dimostrazione.

$$\begin{aligned} f(n) = O(g(n)) \Leftrightarrow & f(n) \leq cg(n), \forall n \geq m \\ \Leftrightarrow & af(n) \leq acg(n), \forall n \geq m, \forall a > 0 \\ \Leftrightarrow & af(n) \leq c' g(n), \forall n \geq m, c' = ac > 0 \\ \Leftrightarrow & af(n) = O(g(n)), \forall a > 0 \end{aligned}$$

□

Sommatoria (sequenza di algoritmi)

$$f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) \Rightarrow f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$$
$$f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) \Rightarrow f_1(n) + f_2(n) = \Omega(\max(g_1(n), g_2(n)))$$

Dimostrazione (Lato O).

$$\begin{aligned} f_1(n) = O(g_1(n)) \wedge f_2(n) = O(g_2(n)) \Rightarrow \\ f_1(n) \leq c_1 g_1(n) \wedge f_2(n) \leq c_2 g_2(n) \Rightarrow \\ f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \Rightarrow \\ f_1(n) + f_2(n) \leq \max\{c_1, c_2\}(2 \cdot \max(g_1(n), g_2(n))) \Rightarrow \\ f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))) \end{aligned}$$

□

Prodotto (cicli annidati)

$$f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) \Rightarrow f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$
$$f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) \Rightarrow f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n))$$

Dimostrazione.

$$\begin{aligned} f_1(n) = O(g_1(n)) \wedge f_2(n) = O(g_2(n)) \Rightarrow \\ f_1(n) \leq c_1 g_1(n) \wedge f_2(n) \leq c_2 g_2(n) \Rightarrow \\ f_1(n) \cdot f_2(n) \leq c_1 c_2 g_1(n) g_2(n) \end{aligned}$$

□

Simmetria

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

Dimostrazione. Grazie alla proprietà della dualità, si ha che:

$$\begin{aligned} f(n) = \Theta(g(n)) \quad &\Rightarrow \quad f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n)) \\ f(n) = \Theta(g(n)) \quad &\Rightarrow \quad f(n) = \Omega(g(n)) \Rightarrow g(n) = O(f(n)) \end{aligned}$$

□

Transitività

$$f(n) = O(g(n)), g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

Dimostrazione.

$$\begin{aligned} f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow \\ f(n) \leq c_1 g(n) \wedge g(n) \leq c_2 h(n) \Rightarrow \\ f(n) \leq c_1 c_2 h(n) \Rightarrow \\ f(n) = O(h(n)) \end{aligned}$$

□

2.2.3 Altre funzioni di costo

Logaritmi v/s funzioni lineari

Proprietà dei Logaritmi Vogliamo provare che $\log(n) = O(n)$. Dimostriamo per induzione che

$$\exists c > 0, \exists m \geq 0 : \log n \leq cn, \forall n \geq m \qquad \text{definizione di } O$$

Dimostrazione.

Caso Base. $(n = 1)$: $\log 1 = 0 \overset{c}{\leq} 0 \cdot \overset{n}{1} = 0$
Ipotesi induttiva: sia $\log k \leq ck, \forall k \leq n$.

Passo Induttivo. Dimostriamo la proprietà per $n + 1$:

$$\begin{aligned} \log(n+1) \leq & \log(n+n) = \log 2n && \forall n \geq 1 \\ = & \log 2 + \log n && \log ab = \log a + \log b \\ = & 1 + \log n && \log 2 = 1 \\ \leq & 1 + cn && \text{per induzione} \\ \overset{?}{\leq} & c(n+1) && \text{Obbiettivo} \end{aligned}$$

$$1 + cn \leq c(n+1) \Leftrightarrow c \geq 1$$

□

2.2.4 Giocando con le espressioni

Es 1 È vero che $\log_a n = \Theta(\log n)$?
Si: $\log_a n = (\log_a 2) \cdot (\log_2 n) = \Theta(\log n)$

Es 2 È vero che $\log n^a = \Theta(\log n)$, per $a > 0$?
Si: $\log n^a = a \log n = \Theta(\log n)$

Es 3 È vero che $2^{n+1} = \Theta(2^n)$?
Si: $2^{n+1} = 2 \cdot 2^n = \Theta(2^n)$

Es 4 È vero che $2^n = \Theta(3^n)$?
Ovviamente $2^n = O(3^n)$
Ma: $3^n = \left(\frac{3}{2} \cdot 2\right)^n = \left(\frac{3}{2}\right)^n \cdot 2^n$: Quindi non esiste $c > 0$ tale per cui $\left(\frac{3}{2}\right)^n \cdot 2^n \leq c2^n$, quindi $2^n \neq O(3^n)$

2.2.5 Classificazione delle funzioni

È possibile definire un ordinamento delle principali classi estendendo le relazioni che abbiamo dimostrato:
Per ogni $r < s, h < k, a < b$:

$$o(1) \subset O(\log^r n) \subset O(\log^s n) \subset O(n^h) \subset O(n^h \log^r n) \subset O(n^h \log^s n) \subset O(n^k) \subset O(a^n) \subset O(b^n)$$

2.3 Ricorrenze

2.3.1 Introduzione

Equazioni di ricorrenza Quando si calcola la complessità di un algoritmo ricorsivo, questa viene espressa tramite un'**equazione di ricorrenza**, ovvero una formula definita in maniera ricorsiva.

MergeSort

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{se } n > 1 \end{cases}$$

Forma Chiusa L'obbiettivo è quello di ottenere, quando possibile, una **formula chiusa** che rappresenti la classe di complessità della funzione.

MergeSort

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & n > 1 \\ \Theta(1) & n \leq 1 \end{cases} \Longrightarrow T(n) = \Theta(n \log n)$$

2.3.2 Metodo dell'albero di ricorsione, o per livelli

Introduzione "Srotoliamo" la ricorrenza in un albero i costi ai vari livelli della ricorsione.

Esempio 1

$$T(n) = \begin{cases} T(n/2) + b & n > 1 \\ c & n \leq 1 \end{cases}$$

È possibile risolvere questa ricorrenza nel modo seguente:

$$\begin{aligned} T(n) =& b + T(n/2) \\ =& b + b + T(n/4) \\ =& b + b + b + T(n/8) \\ =& \dots \\ =& \underbrace{b + b + \dots + b}_{\log n} + T(1) \end{aligned}$$

Assumiamo per semplicità che $n = 2^k$. $T(n) = b \log n + c = \Theta(\log n)$

Esempio 2

$$T(n) = \begin{cases} 4T(n/2) + n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

È possibile risolvere questa ricorrenza nel modo seguente:

$$\begin{aligned} T(n) =& n \cancel{\sum_{j=0}^{\log(n-1)} 2^j} && \underbrace{+4^{\log n}}_{\text{somma degli } T(1)} \\ & \overbrace{n \text{ per prop. log.}} \\ \Rightarrow & \underbrace{n \cdot \frac{\overbrace{2^{\log n}}^{n} - 1}{2 - 1}}_{\substack{\text{usando:} \\ \forall x \neq 1 : \sum_{j=0}^{k} x^j = \frac{x^{k+1}-1}{x-1}}} && +4^{\log n} \\ =& n(n-1) && +4^{\log n} \\ & && \overbrace{\text{cambiamento di base}} \\ =& n^2 - n && \overbrace{+n^2} \\ =& 2n^2 - n = \Theta(n^2) \end{aligned}$$

Esempio 3

$$T(n) = \begin{cases} 4T(n/2) + n^3 & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Da questa equazione notiamo che il primo livello ha costo n^3, il secondo $4\left(\frac{n}{2}\right)^3$ il terzo $4^2\left(\frac{n}{2^2}\right)^3$ e così via. Quindi possiamo scrivere la sommatoria:

$$\begin{aligned}
T(n) =& n^3 + 4\left(\frac{n}{2}\right)^3 + \ldots + 4^{\log n-1}\left(\frac{n}{2^{\log n-1}}\right)^3 && +4^{\log n} \\
=& \sum_{i=0}^{\log n-1} 4^i \left(\frac{n}{2^i}\right)^3 && +4^{\log n} \\
=& n^3 \sum_{i=0}^{\log n-1} \left(\frac{2^{2i}}{2^{3i}}\right) && +4^{\log n} \\
=& n^3 \sum_{i=0}^{\log n-1} \left(2^{2i-3i}\right) && +4^{\log n} \\
=& n^3 \sum_{i=0}^{\log n-1} \left(2^{-1\cdot i}\right) && +4^{\log n} \\
=& n^3 \sum_{i=0}^{\log n-1} \left(\frac{1}{2}\right)^i && \overbrace{+n^2}^{\text{cambiamento di base}} \\
\leq& n^3 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i && +n^2 \\
=& n^3 \cdot \overbrace{\frac{1}{1-\frac{1}{2}}}^{\text{usando: } \forall x, |x|\leq 1: \sum_{i=0}^{\infty} x^i = \frac{1}{1-x}} && +n^2 \\
=& 2n^3 + n^2
\end{aligned}$$

[1]

Abbiamo dunque dimostrato che $T(n) \leq 2n^3 + n^2 = O(n^3)$ però non possiamo, tramite la dimostrazione precedente, dire che $T(n) = \Theta(n^3)$ in quanto siamo passati ad una disequazione. In questo particolare caso d'altra parte possiamo notare che $T(n) \geq n^3$ il che ci porta ad affermare che $T(n) = \Omega(n^3)$ e quindi $T(n) = \Theta(n^3)$.

2.3.3 Metodo di sostituzione

Introduzione È il metodo in cui si cerca di **indovinare** (**guess**) la soluzione e si prova a dimostrarla per **induzione**.

Primo esempio

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

[1] In quanto la sommatoria tende a crescere allora abbiamo potuto sostituire $\log n$ con ∞ e modificare il segno di uguaglianza con quello di $\leq$ in quanto la sommatoria verso l'infinito converge è più grande di quella finita

Notiamo come il costo dei vali livelli sia $n + n/2 + n/4 + \ldots$. Dunque possiamo ipotizzare di poter scrivere:

$$\begin{aligned} T(n) =& n \cdot \sum_{i=0}^{\log n} \left(\frac{1}{2}\right)^i \\ \leq& n \cdot \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i \\ =& n \cdot \underbrace{\frac{1}{1-\frac{1}{2}}}_{\text{usando } \forall x, |x|<1: \sum_{i=0}^{\infty} x^i = \frac{1}{1-x}} \\ =& 2n \end{aligned}$$

[2]

Limite superiore Tentiamo quindi ora di dire che $T(n) = O(n)$:

Caso Base $T(1) = 1 \overset{?}{\leq} 1 \cdot c \Leftrightarrow \forall c \geq 1$

Passo Induttivo Dimostriamo la disequazione per $T(n)$

$$\begin{aligned} T(n) =& T(\lfloor n/2 \rfloor) + n \\ \leq& \overbrace{c\lfloor n/2 \rfloor}^{\text{ipotizzato } O(n)} + n \\ \leq& c \cdot \overbrace{n/2}^{\text{intero inferiore}} + n \\ =& n(c/2 + 1) \\ \overset{?}{\leq}& cn \\ \Leftrightarrow& c/2 + 1 \leq c \Leftrightarrow c \geq 2 \end{aligned}$$

Dunque abbiamo provato che: $T(n) \leq cn$, nel caso base $c \geq 1$ e nel passo induttivo $c \geq 2$. In quanto deve valere per entrambi i casi allora il primo valore utile di c è 2, avendo provato che per $n = 1$ la disequazione vale e che per tutti i successivi valori di n la disequazione vale allora possiamo dire che $T(n) = O(n)$.

Limite Inferiore Tentiamo di dimostrare che $T(n) = \Omega(n)$:

Caso Base Dimostriamo che $T(1) = 1 \overset{?}{\geq} 1 \cdot d \Leftrightarrow \forall d \leq 1$

[2] Abbiamo potuto usare il $\leq$ in quanto la sommatoria in questione all'infinito è sempre maggiore di quella finita e ci stiamo calcolando il costo massimo

Passo Induttivo Dimostriamo la disequazione per $T(n)$:

$$\begin{aligned}
T(n) =& T(\lfloor n/2 \rfloor) + n \\
& \overset{\text{per ipo. induttiva sostituzione}}{\geq} \overbrace{d\lfloor n/2 \rfloor} + n \\
& \overset{\text{intero inferiore}}{\geq} d \cdot \overbrace{\frac{n}{2} - 1} + n \\
=& \left(\frac{d}{2} - \frac{1}{n} + 1\right) n \overset{?}{\geq} dn \\
\Leftrightarrow & \frac{d}{2} - \frac{1}{n} + 1 \geq d \\
\Leftrightarrow & d \leq 2 - \frac{2}{n}
\end{aligned}$$

Abbiamo quindi dimostrato che $T(n) \geq dn$, nel caso base $d \leq 1$ e nel passo induttivo $d \leq 2 - \frac{2}{n}$. In quanto deve valere per entrambi i casi allora il primo valore utile di d è 1, avendo provato che per $n = 1$ la disequazione vale e che per tutti i successivi valori di n la disequazione vale allora possiamo dire che $T(n) = \Omega(n)$.

Conclusione Avendo provato che $T(n) = O(n)$ e $T(n) = \Omega(n)$ e ricordando che se $T(n) = O(n) \wedge T(n) = \Omega(n) \Leftrightarrow T(n) = \Theta(n)$ concludendo che la funzione di costo di $T(n)$ cresce in maniera lineare.

Terzo esempio - Difficoltà matematiche

$$T(n) = \begin{cases} T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 1 & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Limite Superiore Dalla seguente si può notare come il costo di ogni livello sia 1 e che il numero di livelli sia $\log n$. Inoltre la ricorsione viene eseguita su due rami, quindi possiamo scrivere la seguente sommatoria:

$$\begin{aligned}
T(n) =& \sum_{i=0}^{\log n} 2^i \\
=& 1 + 2 + 4 + \ldots + \frac{n}{4} + \frac{n}{2} + n \\
=& O(n)
\end{aligned}$$

Proviamo ora a dimostrare che $T(n) = O(n)$:

Passo Induttivo Ipotizzando che $\forall k < n : T(k) \leq ck$, dimostriamo che $T(n) \leq cn$:

$$\begin{aligned}
T(n) =& T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\
\leq& c\left\lfloor \frac{n}{2} \right\rfloor + c\left\lceil \frac{n}{2} \right\rceil + 1 \\
\leq& cn + 1 \\
\overset{?}{\geq}& cn \Rightarrow 1 \leq 0 \qquad \text{impossibile}
\end{aligned}$$

Sebbene la dimostrazione sia fallita ma l'intuizione ci dice che $T(n) = O(n)$

Proviamo dunque a dimostrarlo per un **ordine inferiore**: $cn + 1 \leq cn$

Passo Induttivo Ipotizzando che $\exists b > 0, \forall k < n : T(k) \leq ck - b$ allora dimostriamo la disequazione per $T(n)$:

$$\begin{aligned} T(n) =& T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ \leq& c\left\lfloor \frac{n}{2} \right\rfloor - b + c\left\lceil \frac{n}{2} \right\rceil - b + 1 \\ =& cn - 2b + 1 \\ \overset{?}{\leq}& cn - b \\ \Rightarrow& \cancel{cn} - 2b + 1 \leq \cancel{cn} - b \Rightarrow b \geq 1 \end{aligned}$$

Dimostriamo il passo base per $b = 1$: $T(1) = 1 \overset{?}{\leq} 1 \cdot c - b \Leftrightarrow \forall c \geq b + 1$

Limite Inferiore Proviamo ora a dimostrare che $T(n) = \Omega(n)$:

Passo Induttivo Ipotizzando che $\forall k < n : T(k) \geq dk$, dimostriamo che $T(n) \geq dn$:

$$\begin{aligned} T(n) =& T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ \geq& d\left\lfloor \frac{n}{2} \right\rfloor + d\left\lceil \frac{n}{2} \right\rceil + 1 \\ =& dn + 1 \\ \overset{?}{\geq}& dn \Rightarrow 1 \geq 0 \end{aligned}$$

Il che è vero $\forall d$ in quanto d è positivo per ipotesi.

Caso base Dimostriamo la disequazione per $T(1)$:

$$T(1) = 1 \geq 1 \cdot d \Leftrightarrow d \leq 1$$

Dunque abbiamo provato che $T(n) = \Omega(n)$

Avendo dimostrato in precedenza che $T(n) = O(n)$ e $T(n) = \Omega(n)$ possiamo concludere che $T(n) = \Theta(n)$ e quindi la funzione di costo cresce linearmente.

2.3.4 Metodo dell'esperto, o delle ricorrenze comuni

Riccorenze comuni Esiste un'ampia classe di ricorrenze che si risolvono facilmente facendo ricorso a qualche teorema, ogni teorema è applicabile ad una particolare classe di ricorrenze.

Ricorrenze lineari con partizione bilanciata

Teorema 2.1. Siano a, b costanti intere tali che $a \geq 1$ e $b \geq 2$, e esistano c, β costanti reali tali che $c > 0$ e $\beta \geq 0$. Sia $T(n)$ data dalla seguente relazione di ricorrenza:

$$T(n) = \begin{cases} aT(n/b) + cn^\beta & n > 1 \\ 1 & n \leq 1 \end{cases} \tag{2.1}$$

Posto: $\alpha = \frac{\log a}{\log b} = \log_b a$ allora:

$$T(n) = \begin{cases} \Theta(n^\alpha) & \alpha > \beta \\ \Theta(n^\alpha \log n) & \alpha = \beta \\ \Theta(n^\beta) & \alpha < \beta \end{cases} \tag{2.2}$$

Assunzioni Assumiamo che n sia una potenza intera di b, ovvero $n = b^k, k = \log_b n$.

Influisce sul risultato?

- Supponendo che l'input abbia dimensione $b^k + 1$
- Estendiamo l'input fino alla dimensione b^{k+1} (**padding**)
- L'input è stato esteso di un fattore b
- Il che non cambia la complessità computazionale

Dimostrazione caso 1

Dimostrazione caso 2 $\alpha = \beta$

Dimostrazione. Ne segue che: $q = b^{\alpha-\beta} = 1$ e dunque la funzione $T(n)$:

$$
\begin{aligned}
T(n) =& dn^\alpha + cb^{k\beta} \sum_{i=0}^{k-1} q^i && && (2.3)\\
=& n^\alpha d + cn^\beta k && q^i = 1^i = 1 && (2.4)\\
=& n^\alpha d + cn^\alpha k && \alpha = \beta && (2.5)\\
=& n^\alpha (d + ck) && && (2.6)\\
=& n^\alpha \left[d + c\frac{\log n}{\log b} \right] && k = \log_b n && (2.7)
\end{aligned}
$$

□

e come conseguenza $T(n) = \Theta(n^\alpha \log n)$