

Ricorsione e equazioni di ricorrenza

Prima di cominciare notiamo di nuovo la differenza del calcolo della complessità quando la calcoliamo come linee di codice eseguite ed invece come operazioni del modello RAM.

Problema: Ricerca di un elemento in una lista non ordinata:

```
#Torna la posizione dell'elemento cercato, altrimenti -1

def RicercaSequenziale(Lista, elemento):
    n = len(Lista)
    i = 0
    for i in range(n):
        if Lista[i] == elemento:
            return i
    return -1
```

Calcolo della complessità:

Consideriamo $T(n)$ = numero elementi acceduti (linea 4) su un array di dimensione n
Di conseguenza avremo che:

$T_{worst}(n) = n$ quando $x \notin Lista$ oppure è in ultima posizione.

$T_{avg}(n) = \frac{n+1}{2}$ assumendo che $x \in L$ e che si trovi con la stessa probabilità in una posizione.

Se adesso decidessimo di svolgere questo calcolo sul **modello RAM otterremo dei risultati differenti**, allora consideriamo $T(n)$ = numero operazioni RAM su un array di dimensione n

$T_{worst}(n) = \Theta(n)$ quando $x \notin Lista$ oppure è in ultima posizione.

$T_{avg}(n) = \Theta(n)$ assumendo che $x \in L$ e che si trovi con la stessa probabilità in una posizione.

La differenza è che nel modello RAM **otteniamo una soluzione leggermente meno dettagliata, ma con un livello di semplicità maggiore.**

Problema: Ricerca di un elemento in una lista ordinata:

Questa è una variante del problema precedente che si può risolvere tramite l'algoritmo della Ricerca Binaria.

```
def RicercaBinaria(Lista, elemento, i, j):
    if i > j:
        return -1
    m = (i+j)//2
    if Lista[m] == elemento:
        return m
    if Lista[m] > elemento:
        return RicercaBinaria(Lista, elemento, i, m-1)
    else:
        return RicercaBinaria(Lista, elemento, m+1, j)

Lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
elemento = 5
print(RicercaBinaria(Lista, elemento, 0, len(Lista)-1))
```

i e j indicano la porzione di L in cui vogliamo cercare x , man mano che l'algoritmo va avanti la dimensione di ricerca diminuisce finché non troviamo il valore cercato.

Adesso utilizzando il modello RAM otteniamo che:

$$T(n) = T(n/2) + O(1) \rightarrow T(n) = O(\log(n))$$

Ma come possiamo analizzare gli algoritmi ricorsivi? Semplice, esistono delle tecniche per risolvere le equazioni di ricorrenza:

- Iterazione
- Albero della ricorsione
- Sostituzione
- Teorema Master
- Cambiamento di variabile

Equazioni di ricorrenza

Le equazioni di ricorrenza sono tutte quelle equazioni che richiamano se stesse.

La complessità computazionale di un algoritmo ricorsivo può essere descritto tramite una equazione di ricorrenza.

 **Esempi:**

$$\text{Fibonacci2} \rightarrow T(n) = T(n-1) + T(n-2) + O(1)$$

$$\text{Algoritmo di ricerca binaria} \rightarrow T(n) = T(n/2) + O(1)$$

$$\text{Algoritmo pesate numero quattro} \rightarrow T(n) = T(n/3) + O(1)$$

La funzione $T(n)$ **richiama se stessa, ma su un'istanza più piccola.**

Inoltre ogni funzione dovrà avere un **Caso Base** che si può rappresentare come:

$$T(\text{costante}) = \text{cost}$$

Solitamente $T(1) = 1$

Metodo dell'iterazione

L'idea di questo metodo è di **'srotolare'** la ricorsione **ottenendo una sommatoria** dipendente solo dalla dimensione di n del problema iniziale.

 **Esempio 1:** $T(n) = c + T(n/2)$

Per definizione possiamo trovare $T(n/2)$, cioè la funzione chiamerà nuovamente sé stessa, ma su un'istanza più piccola di metà:


$$T(n/2) = c + T(n/4)$$

Ma allora possiamo trovare tutti i valori di $T(n)$ fino ad un valore i :

$$T(n) = c + T(n/2) = 2c + T(n/4) = 3c + T(n/8) = \dots = ic + T(n/2^i)$$

Adesso cerchiamo il valore i che rende la funzione un caso base, cioè per $i = \log_2(n)$:

$$T(n) = c * \log_2(n) + T(1) = \Theta(\log(n))$$

 **Esempio 2:** $T(n) = 2T(n-1) + 1$

$$T(n) = 2T(n-1) + 1 = 4T(n-2) + 2 + 1 = 8T(n-3) + 4 + 2 + 1 = 16T(n-4) + 8 + 4 + 2 + 1$$

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

Adesso dobbiamo trovare il valore i che renda $T(n-i) = T(1)$, cioè $i = n-1$

Quindi sostituiamo:

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

Il problema principale con questa tecnica è che ci sono alcune funzioni ricorsive troppo complicate è per questo motivo non funziona sempre l'idea di srotolare.

⚠ Sommatoria:

Per convincerci dell'esempio sopra dobbiamo tenere di conto della seguente uguaglianza:

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

Quindi:

$$\sum_{j=0}^{n-2} 2^j = 2^{n-1} - 1$$

Notiamo che il valore sopra alla sommatoria deve essere aumentato di uno è posto come esponente del due.

Tecnica dell'albero della ricorsione

L'idea di questa tecnica è quella di disegnare un albero con tutte le chiamate ricorsive indicando la dimensione di ogni nodo, stimando il tempo complessivo sommando il tempo speso in ogni nodo.

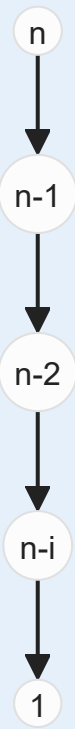
Due suggerimenti che potrebbero tornare utili:

- Se il tempo speso da ogni nodo è costante, $T(n)$ è proporzionale al numero di nodi.
- A volte conviene analizzare l'albero per livelli, quindi analizzare il tempo speso su ogni livello e stimare il numero di livelli.

✍ Esempio 1:

$T(n) = T(n-1) + 1$ e $T(1) = 1$, in questo caso $T(n-1)$ sarà la struttura dell'albero mentre 1 è il costo di ogni nodo.

Costruiamo l'albero:



Quanto costa ogni nodo? **Uno**, Quanto nodi ha l'albero? **n**

E' facile capire che il costo sarà $T(n) = \Theta(n)$.

Esempio 2:

$$T(n) = T(n - 1) + n \text{ e } T(1) = 1$$

La situazione qui è un po' più complicata perché se è vero che l'albero è lo stesso di quello dell'esempio di sopra, adesso il costo di ogni nodo è n

Quindi ogni nodo costa al più n , ma non tutti come per esempio il caso base.

L'albero ha n nodi, possiamo **trovare un upper bound**, cioè una funzione che la nostra complessità non riuscirà mai a superare (Praticamente un tetto).

$T(n) = O(n^2)$ sarà il nostro upper bound.

Ma vale anche $T(n) = \Theta(n^2)$?

Per capirlo dobbiamo trovare un **lower bound**, per trovarlo basta considerare che i nodi che vanno da n a $n - i$, che sono metà ($n/2$) dei nodi costeranno al più $n/2$.

Quindi $T(n) \geq n/2 * n/2 = n^2/4$, cioè $T(n) = \Omega(n^2)$

Allora per definizione $T(n) = \Theta(n^2)$

Quindi l'idea generale quando si crea un albero della ricorsione è quella di tenere conto di un **lower bound** e di un **upper bound** in modo tale da trovare di preciso la complessità.

⚠ Albero binario completo:

E' un tipo di albero particolare in cui tutte le foglie hanno la stessa altezza e tutti i nodi interni hanno esattamente due figli.

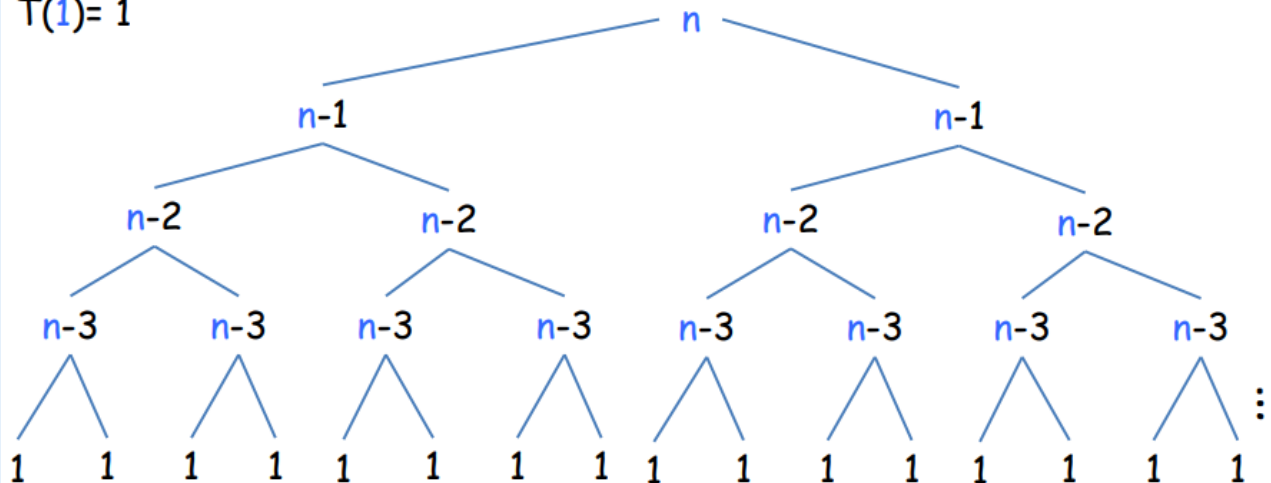
Potrebbe tornare utile anche calcolare la complessità di ogni livello dell'albero specialmente nelle situazioni in cui le foglie hanno altezze diverse, cioè che alcuni rami finiscono prima degli altri, in questi casi conviene trovare un **lower bound sul ramo che finisce prima** e un **upper bound su quello che finisce per ultimo**.

✎ Altri esempi solo della costruzione degli alberi e non della complessità:

$$T(n) = 2T(n-1) + n$$

$$T(n) = 2T(n-1) + n$$

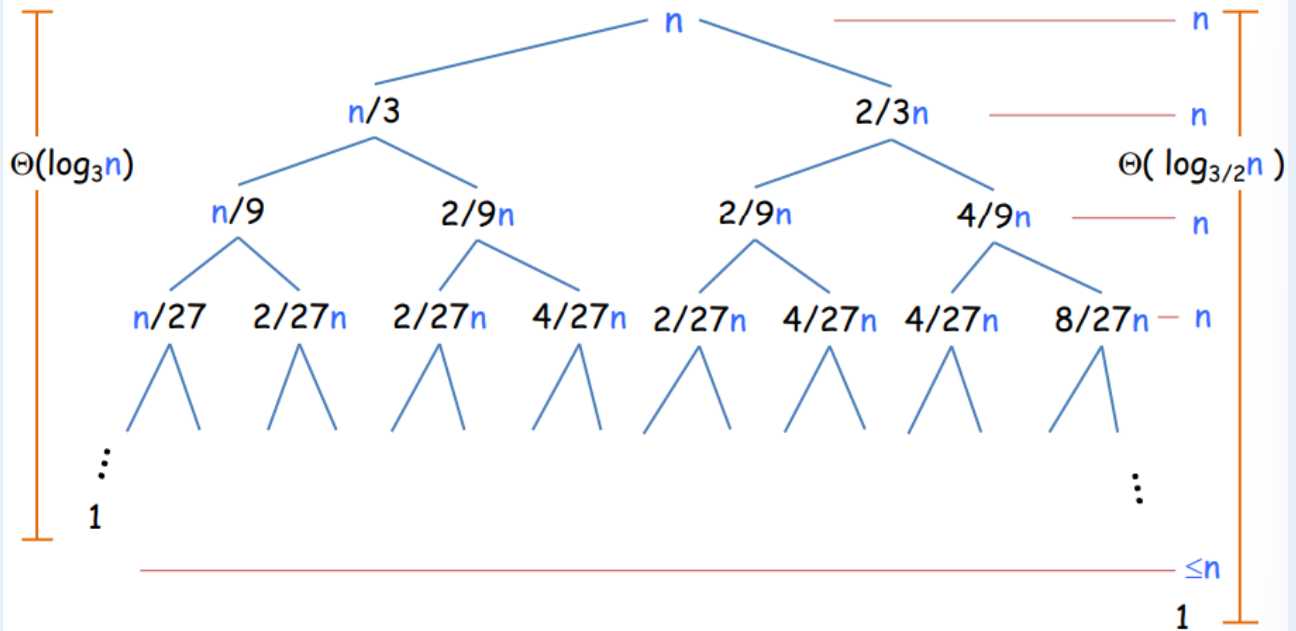
$$T(1) = 1$$



$$T(n) = T(n/3) + T(2/3 * n) + n$$

$$T(n) = T(n/3) + T(2/3 n) + n$$

$$T(1) = 1$$



In questo caso bisogna calcolare il costo di ogni livello n e moltiplicarlo per l'altezza del ramo più piccolo per avere un lower bound ($\Omega(n \log(n))$), la stessa cosa, ma al contrario per l'upper bound ($O(n \log(n))$) di conseguenza la complessità è $\Theta(n \log(n))$.

Metodo della sostituzione

Consiste nel cercare di intuire la soluzione, dimostrarla per induzione per poi risolvere la disequazione rimasta rispetto alle costanti.

Esempio:

$$T(n) = 4T(n/2) + n \text{ con } T(1) = 1$$

Ipotizziamo ad occhio che $T(n) = O(n^2)$

Di conseguenza la nostra ipotesi induttiva è $T(n) \leq cn^2$, quindi:

$$T(n) \leq 4c\left(\frac{n}{2}\right)^2 + n = cn^2 + n \leq cn^2$$

Adesso questa equazione sopra è impossibile da risolvere per c .

⚠️ Ciò non significa che $T(n) \neq O(n^2)$, infatti potrebbe essere che la nostra ipotesi induttiva è troppo 'debole' e quindi dobbiamo dimostrarla in maniera 'più forte.'

Proviamo ad usare questa come ipotesi induttiva:


$$T(n) \leq c_1 n^2 - c_2 n$$

Quindi:

$$T(n) \leq 4(c_1(\frac{n}{2})^2 - c_2(\frac{n}{2})^2) = c_1 n^2 - 2c_2 n + n \leq c_1 n^2 - c_2 n$$

Adesso l'ultima parte la possiamo riscrivere come:

$$c_1 n^2 - c_2 n - (c_2 n - n) \leq c_1 n^2 - c_2 n$$

 La parte nella parentesi la chiamiamo **residuo**, le altre due parti invece sono identiche, quindi per vedere per quali valori di c la disequazione ci basta porre il residuo maggiore o uguale a zero.

$$c_2 n - n \geq 0 \rightarrow c_2 \geq 1$$

Ora non ci resta che fare il caso base:

$$T(1) = 1 \leq c_1 - c_2 \rightarrow c_1 = 2 \text{ e } c_2 = 1$$

Abbiamo trovato i valori delle c adesso li possiamo sostituire all'ipotesi induttiva ormai dimostrata:

$$T(n) \leq 2n^2 - n \rightarrow T(n) = O(n^2)$$

Tecnica del divide et impera e Teorema Master

Gli algoritmi basati sulla tecnica del **divide et impera** funzionano in questo modo:

- Abbiamo un problema di dimensione n è lo **dividiamo** in a sottoproblemi di dimensione n/b .
- Risolviamo i sottoproblemi **ricorsivamente**.
- **Ricombiniamo le soluzioni**, $f(n)$ è il tempo che serve per dividere e ricombinare le istanze di dimensione n .

La relazione di ricorrenza data è:

$$\begin{cases} aT(n/b) + f(n) & \text{se } n > 1 \\ \Theta(1) & \text{se } n = 1 \end{cases}$$

 **Esempi:**

Fibonacci 6: $a = 1, b = 2, f(n) = O(1)$

Algoritmo ottimo di pesatura: $a = 1, b = 3, f(n) = O(1)$

Teorema Master: La relazione di ricorrenza

$$\begin{cases} aT(n/b) + f(n) & \text{se } n > 1 \\ \Theta(1) & \text{se } n = 1 \end{cases}$$

ha soluzione:

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

Esempi:

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

Stiamo nel **secondo caso del teorema**:

$$a = 2, b = 2, f(n) = n = \Theta(n^{\log_2 2}) \rightarrow T(n) = \Theta(n \log(n))$$

2) $T(n) = c + 3T(n/9)$:

Stiamo nel **primo caso del teorema**:

$$a = 3, b = 9, f(n) = c = O(n^{\log_9 2 - \epsilon}) \rightarrow T(n) = \Theta(\sqrt[3]{n})$$

Possiamo scegliere per esempio: $\epsilon = 0.1$

3) $T(n) = n + 3T(n/9)$:

Stiamo nel **terzo caso**:

$$3(n/9) \leq cn \text{ per } c = 1/3 \rightarrow T(n) = \Theta(n)$$

4) $T(n) = n \log n + 2T(n/2)$:

$$a = 2, b = 2, f(n) = \omega(n^{\log_2(2)})$$

$$\text{MA } f(n) \neq \Omega(n^{\log_2 2 + \epsilon}) \quad \forall \epsilon > 0$$

Quindi **non si può applicare il teorema Master**.

Cambiamento di variabile

Facciamo semplicemente il **cambio di una variabile su un valore difficile da calcolare**:

Esempio: $T(n) = T(\sqrt{n}) + O(1)$ con $T(1) = 1$

$$T(n) = T(n^{1/2}) + O(1)$$

Se sostituiamo n con $2^x \rightarrow x = \log_2(n)$ otteniamo:

$$T(2^x) = T(2^{x/2}) + O(1)$$

Se ora sostituiamo $R(x) = T(2^x)$, otteniamo:

$$R(x) = R(x/2) + O(1) \rightarrow R(x) = O(\log(x))$$

Adesso bisogna solo fare le sostituzioni all'indietro:

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