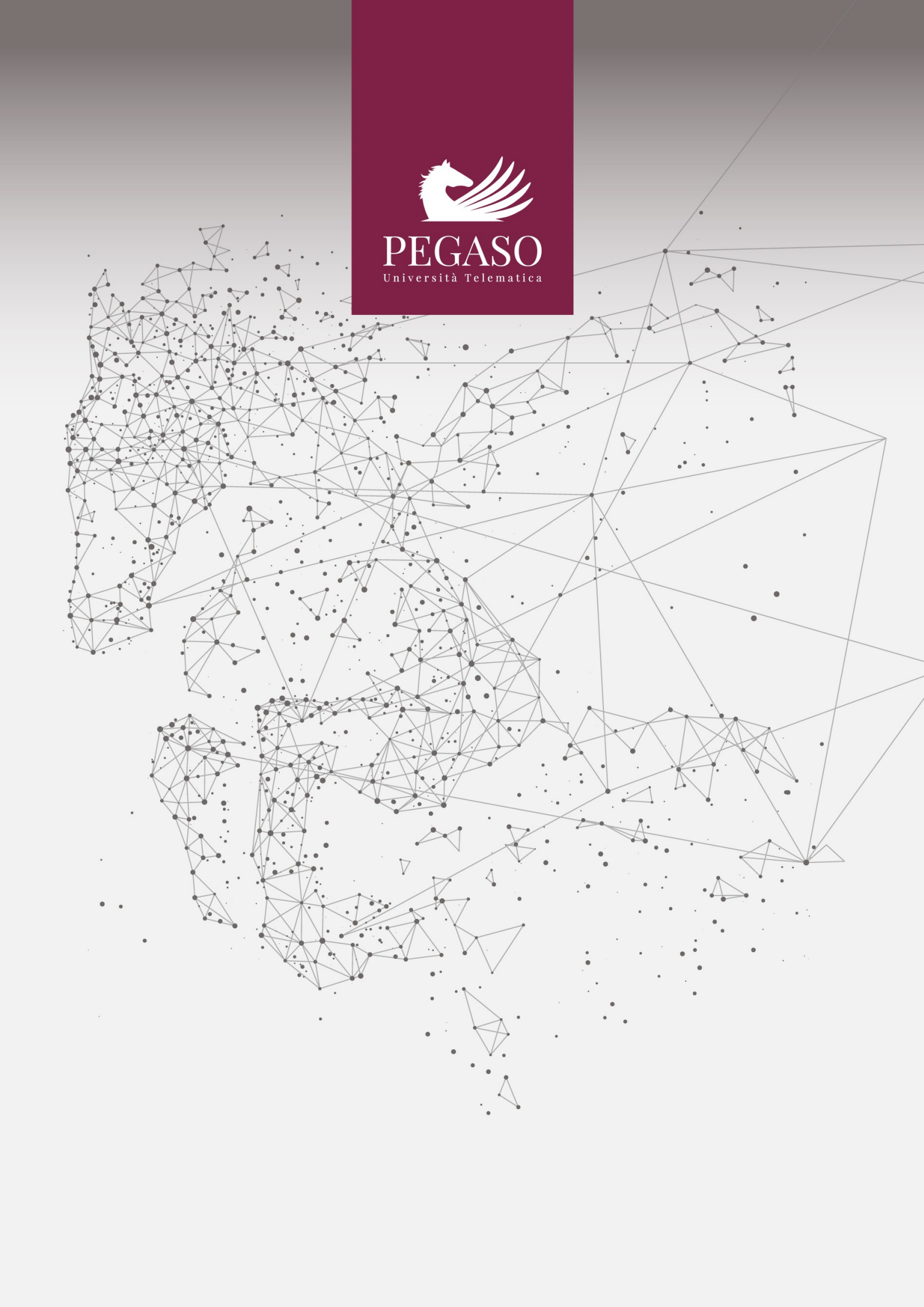
PEGASO
Università Telematica

Indice

1. NOTAZIONE ASINTOTICA 3
2. CALCOLO DELLA COMPLESSITÀ DEGLI ALGORITMI RICORSIVI.....6
3. METODI DI RISOLUZIONE DELLE RICORRENZE.....11
BIBLIOGRAFIA 14

1. Notazione asintotica

Gli algoritmi che risolvono uno stesso problema (decidibile) vengono confrontati sulla base della loro efficienza, cioè, in base al loro tempo d'esecuzione; tale tempo di esecuzione è e espresso come una funzione della dimensione dei dati di ingresso.

La dimensione n dei dati di ingresso dipende dallo specifico problema e può essere rappresentata dal numero stesso di dati di ingresso oppure dalla quantità di memoria necessaria per contenere tali dati.

Tuttavia, la valutazione del tempo di esecuzione deve essere indipendente dalla tecnologia dell'esecutore e dunque non può essere misurato in una unità temprale come i "secondi" o il tempo di CPU. Per questo motivo si usa come unità di misura il numero di passi base compiuti durante l'esecuzione dell'algoritmo; un passo base può essere:

- l'esecuzione di un'istruzione di assegnamento (priva di chiamate di funzioni);
- la valutazione di un'espressione (priva di chiamate di funzioni) contenuta in un'istruzione di selezione;
- la valutazione di un'espressione (priva di chiamate di funzioni) contenuta in un'istruzione di ripetizione.

È utile a questo punto valutare il tempo d'esecuzione in tre diversi casi:

- **Caso pessimo**: determinato dall'istanza dei dati di ingresso che massimizza il tempo d'esecuzione e quindi fornisce un limite superiore alla quantità di risorse computazionali necessarie all'algoritmo.
- **Caso ottimo**: determinato dall'istanza dei dati di ingresso che minimizza il tempo d'esecuzione e quindi fornisce un limite inferiore alla quantità di risorse computazionali necessarie all'algoritmo.
- **Caso medio**: determinato dalla somma dei tempi d'esecuzione di tutte le istanze dei dati di ingresso, con ogni addendo moltiplicato per la probabilità di occorrenza della relativa istanza dei dati di ingresso.

Si introduce il concetto di "complessità asintotica" (e dunque di "notazione asintotica" in ordine di grandezza) dal momento che il confronto degli algoritmi che risolvono lo stesso problema decidibile si riduce al confronto di funzioni (che possono relazionarsi in modi diversi per istanze diverse dei dati di ingresso): tale notazione ci consente dunque di comparare il tasso di crescita (cioè, il comportamento asintotico) di una funzione nei confronti di un'altra.

Esistono tre notazioni asintotiche:

- **Notazione asintotica O** (notazione O grande): limite superiore asintotico.
- **Notazione asintotica Ω** (notazione Omega): limite inferiore asintotico.

- **Notazione asintotica θ** (notazione Theta): limite asintotico stretto.

  La **notazione asintotica** O è il limite superiore asintotico.

  Date due funzioni $f(n)$ e $g(n)$, si dice che $f(n) = O(g(n))$ se esiste un valore $c > 0$ tale che $0 \leq f(n) \leq c \cdot g(n)$ per ogni $n \geq n_0$.

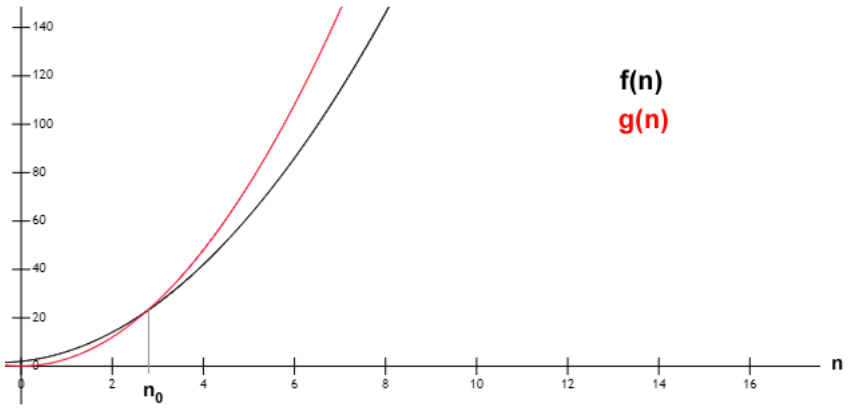

Figura 1 *- Notazione asintotica O*

La **notazione asintotica** Ω è il limite asintotico inferiore.

Date due funzioni $f(n)$ e $g(n)$, si dice che $f(n) = \Omega(g(n))$ se esiste un valore $c > 0$ tale che $0 \leq c \cdot g(n) \leq f(n)$ per ogni $n \geq n_0$.

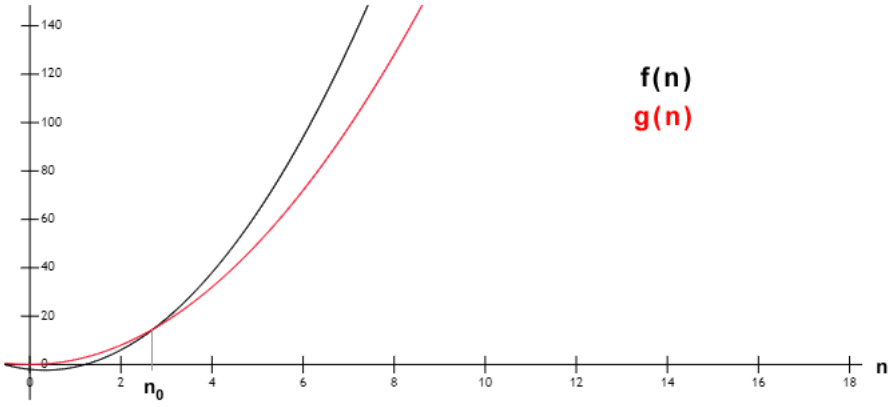

Figura 2 *- Notazione asintotica Omega*

La **notazione asintotica** $\boldsymbol{\theta}$ è il limite asintotico stretto.

Date due funzioni $f(n)$ e $g(n)$, si dice che $f(n) = \theta(g(n))$ se esistono due valori $c_1 > 0$ e $c_2 > 0$ tali che $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ per ogni $n \geq n_0$.

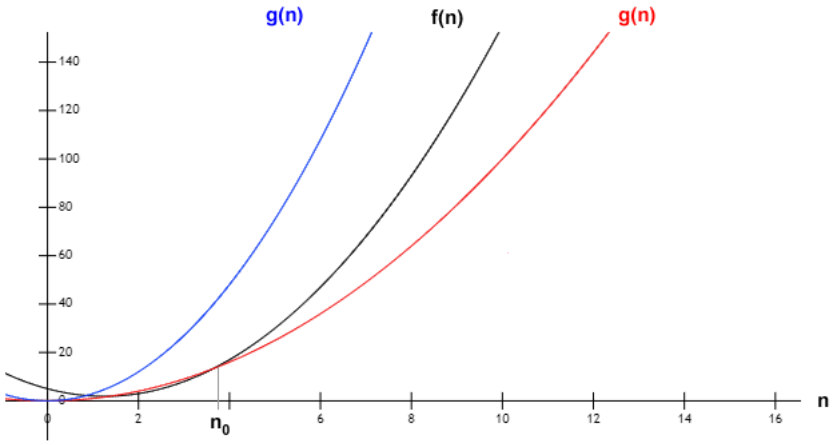

Figura 3 *- Notazione asintotica Theta*

Dal momento che le funzioni tempo d'esecuzione dei vari algoritmi che risolvono lo stesso problema decidibile vengono confrontate considerando il loro andamento al crescere della dimensione n dei dati di ingresso, all'interno delle funzioni, si può intuitivamente comprendere il perché si possano ignorare le costanti moltiplicative e i termini non dominanti al crescere di n.

2. Calcolo della complessità degli algoritmi ricorsivi

Il tempo d'esecuzione di un algoritmo ricorsivo non può essere definito attraverso le regole analizzate nel caso di algoritmi non ricorsivi; va seguito un "procedimento induttivo" attraverso una relazione di "ricorrenza".

Quando un algoritmo contiene una chiamata ricorsiva a sé stesso, il suo tempo di esecuzione spesso può essere descritto con una equazione di ricorrenza che esprime il tempo di esecuzione totale di un problema di dimensione n in funzione del tempo di esecuzione per input più piccoli. Successivamente, è possibile utilizzare strumenti matematici per risolvere l'equazione di ricorrenza e stabilire i limiti delle prestazioni dell'algoritmo.

Consideriamo un algoritmo che utilizza la tecnica del Divide et Impetra

Una ricorrenza per il tempo di esecuzione di un algoritmo divide et impera si basa sui tre passi del paradigma di base:

- Divide.
- Impera.
- Combina.

Supponiamo che $T(n)$ sia il tempo di esecuzione di un problema di dimensione n. Se la dimensione del problema è sufficientemente piccola, per esempio $n \leq c$ per qualche costante c, la soluzione semplice richiede un tempo costante → $\Theta(1)$.

Supponiamo che la suddivisione del problema generi a sotto-problemi e che la dimensione di ciascun sotto-problema sia $1/b$ della dimensione del problema originale (nell'algoritmo mergesort, i valori di a e b sono entrambi pari a 2, ma in generale tali valori possono essere differenti tra loro).

Indichiamo con:

- $D(n)$: tempo per dividere il problema in sotto-problemi
- $C(n)$: tempo per combinare le soluzioni dei sotto-problemi nella soluzione del problema originale

La ricorrenza che si ottiene è la seguente:

$$T(n) = \begin{cases} \Theta(1), & n < c \\ aT\,(n/b) + D(n) + C(n), & negli\ altri\ casi \end{cases}$$

Al fine di semplificare i calcoli, supponiamo che la dimensione del problema originale sia una potenza di 2. Ogni passo **divide** genera due sotto-sequenze di dimensione pari a $n/2$ (di fatto sto assumendo che $a = b = 2$).

Immaginiamo di lavorare su un vettore di n elementi (uno scenario simile a quello di una procedura di ordinamento come il merge sort).

L'algoritmo, se applicato a un solo elemento impiega un tempo costante.

Con $n > 1$ elementi, suddividiamo il tempo di esecuzione nel modo seguente.

Divide: questo passo calcola semplicemente il centro dell'array con un tempo costante: $D(n) = \Theta(1)$.

Impera: risolviamo in modo ricorsivo i due sotto-problemi, ciascuno di dimensione $n/2$; il contributo al tempo di esecuzione è: $2T(n/2)$.

Combina: la procedura MERGE con un array di n elementi richiede un tempo $\Theta(n)$, quindi $C(n) = \Theta(n)$.

Abbiamo dunque che:

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

Possiamo anche riscrivere tale ricorrenza nel seguente modo:

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

Cerchiamo di capire "intuitivamente" il perché $T(n)$ è $\Theta(n \log_2 n)$.

Da un punto di vista grafico, $2T\left(\frac{n}{2}\right) + cn$ può essere rappresentato come segue:

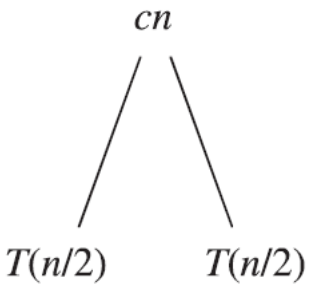

Figura 4 *- prima espansione di T(n)*

La radice dell'albero rappresenta la parte cn che non è altro che il costo del "combina" mentre le 2 foglie sono i costi delle ricorrenze (la cui somma è $2T\left(\frac{n}{2}\right)$)

Tale processo può proseguire…

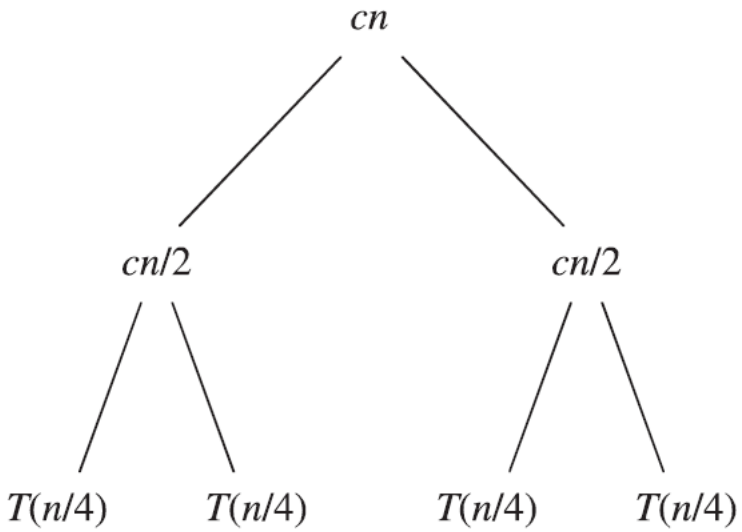

Figura 5 *- seconda espansione di T(n)*

…finché le dimensioni dei problemi si riducono a 1, ciascuno con un costo c.

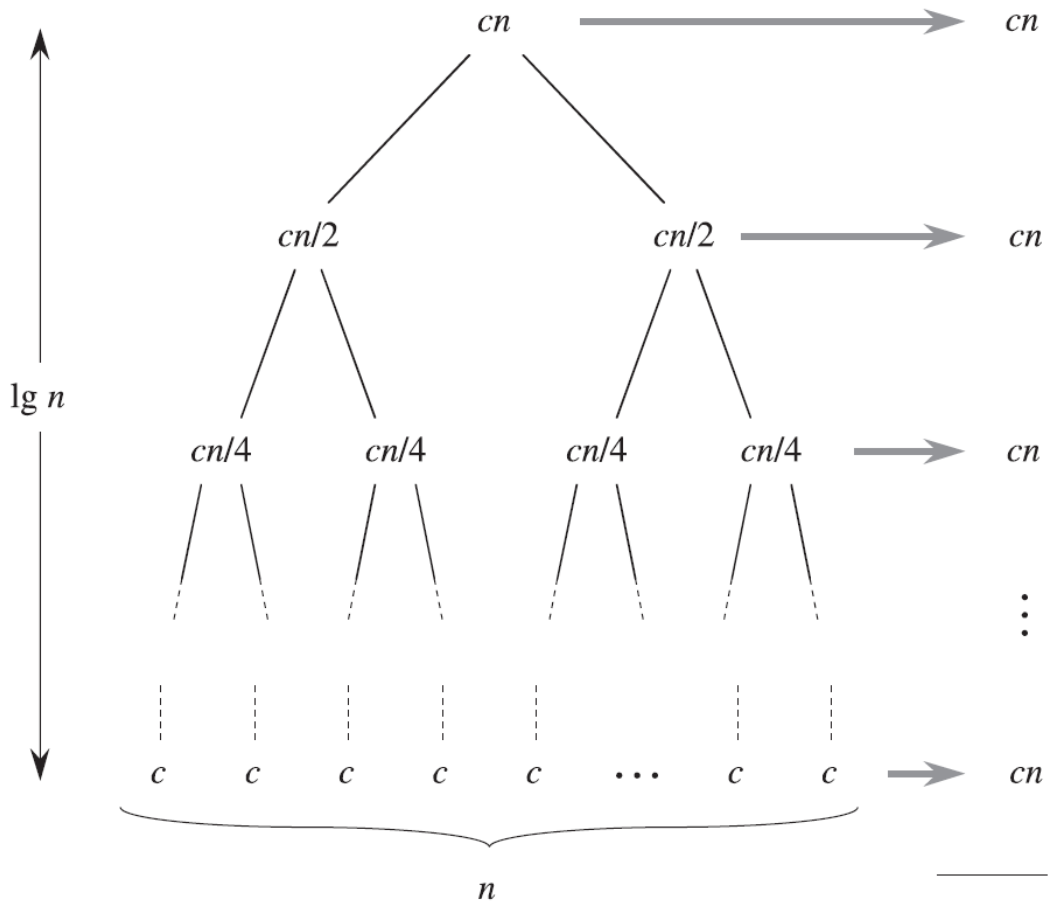

Figura 6 *- espansione di T(n)*

Sommiamo i costi per ogni livello dell'albero:

- il primo livello ha un costo totale cn
- il secondo livello ha un costo totale $c(n/2) + c(n/2) = cn$
- il terzo livello ha un costo totale $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$
- […]

Il livello i sotto il primo ha 2^i nodi, ciascuno dei quali ha un costo $c(n/2^i)$, quindi l'i-esimo livello sotto il primo ha un costo totale $2^i c\left(\frac{n}{2^i}\right) = cn$.

A livello più basso ci sono n nodi, ciascuno con un costo c, per un costo totale cn.

Il numero totale di livelli dell'albero di ricorsione nella Figura 6 è $\log_2 n + 1$.

Questo può essere dimostrato con un ragionamento induttivo informale:

- Il caso base si verifica quando $n = 1$; essendo $\log_2 1$=0 → $\log_2 n + 1 = 1$ (ottenendo così il numero corretto di livelli)

- Supponiamo come ipotesi induttiva, che il numero di livelli di un albero di ricorsione per 2^i nodi sia $\log_2 2^i + 1 = i + 1$. Poichè stiamo supponendo che la dimensione dell'input originale sia una potenza di 2, la successiva dimensione da considerare è 2^{i+1}. Un albero con 2^{i+1} nodi ha un livello in più di un albero con 2^i nodi → il numero totale di livelli è $(i + 1) + 1 = \log_2 2^{i+1} + 1$.

Per calcolare il costo totale sommiamo i costi di tutti i livelli ci sono $\log_2 n + 1$ livelli, ciascuno di costo cn, per un costo totale di $cn(\log_2 n + 1) = cn\log_2 n + cn = \Theta(n\log_2 n)$.

3. Metodi di risoluzione delle ricorrenze

Vi sono 3 metodi di risoluzione delle ricorrenze:

1. **Nel metodo di sostituzione**: ipotizziamo un limite e poi usiamo l'induzione matematica per dimostrare che la nostra ipotesi è corretta.
2. **Il metodo dell'albero di ricorsione**: converte la ricorrenza in un albero i cui nodi rappresentano i costi ai vari livelli della ricorsione; per risolvere la ricorrenza, si adottano delle tecniche che limitano le sommatorie.
3. **Il metodo dell'esperto**: fornisce i limiti sulle ricorrenze nella forma $T(n) = aT(n/b) + f(n)$ dove a ≥ 1, b > 1 ed $f(n)$ `e una funzione data. Questo metodo richiede la memorizzazione di tre casi, ma fatto questo, è facile determinare i limiti asintotici per molte ricorrenze semplici.

Metodo di sostituzione

Il metodo di sostituzione per risolvere le ricorrenze richiede due steps:

1. Si ipotizza la forma della soluzione (purtroppo, non esiste un metodo generale per formulare l'ipotesi)
2. Si usa l'induzione matematica per trovare le costanti e dimostrare che la soluzione funziona.

Metodo dell'albero di ricorsione

Un albero di ricorsione è un ottimo metodo per ottenere una buona ipotesi, che poi viene verificata con il metodo di sostituzione.

In un albero di ricorsione ogni nodo rappresenta il costo di un singolo sotto-problema: si sommano i costi all'interno di ogni livello dell'albero per ottenere un insieme di costi per livello e successivamente si sommano tutti i costi per livello per determinare il costo totale di tutti i livelli della ricorsione.

Gli alberi di ricorsione sono particolarmente utili quando la ricorrenza descrive il tempo di esecuzione di un algoritmo divide et impera.

Metodo dell'esperto

Il metodo dell'esperto è usato per risolvere le ricorrenze della forma

$$T(n) = a\,T(n/b) + f(n)$$

dove $a \geq 1$ e $b > 1$ sono costanti e $f(n)$ è una funzione asintoticamente positiva.

Tale ricorrenza descrive il tempo di esecuzione di un algoritmo che divide un problema di dimensione n in a sottoproblemi, ciascuno di dimensione n/b, dove a e b sono costanti positive. I sottoproblemi vengono risolti in modo ricorsivo, ciascuno nel tempo $T(n/b)$. Il costo per dividere il problema e combinare i risultati dei sotto-problemi è descritto dalla funzione $f(n) = D(n) + C(n)$

Il metodo dell'esperto si basa sul Teorema dell'esperto:

Date le costanti $a \geq 1$ e $b > 1$ ed $f(n)$; sia $T(n)$ una funzione definita sugli interi non negativi dalla ricorrenza $T(n) = a\,T(n/b) + f(n)$, allora $T(n)$ può essere asintoticamente limitata nei seguenti modi:

- Se $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ per qualche $\epsilon > 0$ allora $T(n) = \theta\left(n^{\log_b a}\right)$
- Se $f(n) = \theta\left(n^{\log_b a}\right)$ per qualche $\epsilon > 0$ allora $T(n) = \theta\left(n^{\log_b a} \log_2 n\right)$
- Se $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ per qualche $\epsilon > 0$ e se $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ e per ogni n, allora $T(n) = \theta(f(n))$

In ciascuno dei tre casi, confrontiamo la funzione $f(n)$ con la $n^{\log_b a}$. Intuitivamente, la soluzione della ricorrenza è determinata dalla più grande delle due funzioni.

- Se, come nel caso 1, la funzione $n^{\log_b a}$ è la più grande, allora la soluzione è $T(n) = \theta\left(n^{\log_b a}\right)$; da notare che $f(n)$ deve essere polinomialmente più piccola di $n^{\log_b a}$, cioè deve essere asintoticamente più piccola di $n^{\log_b a}$ per un fattore n^{ϵ} per qualche $\epsilon > 0$
- Se, come nel caso 3, la funzione $f(n)$ è la più grande, allora la soluzione è $T(n) = \theta(f(n))$; da notare che $f(n)$ deve essere polinomialmente più grande e soddisfare la condizione di regolarità $af(n/b) \leq cf\,(n)$.
- Se, come nel caso 2, le due funzioni hanno la stessa dimensione, moltiplichiamo per un fattore logaritmico e la soluzione è $T(n) = \theta\left(n^{\log_b a} \log_2 n\right) = \theta(f(n) \log_2 n)$

Applicazione del metodo dell'esperto

Esempio 1

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

Essendo $a = 9$, $b = 3$ ed $f(n) = n \rightarrow n^{\log_b a} = n^{\log_3 9} = \theta(n^2)$

Dal momento che $f(n) = n = O(n^{\log_3 9-\epsilon})$ dove $\epsilon = 1$, possiamo applicare il caso 1 e concludere che $T(n) = \theta(n^2)$

Esempio 2

Consideriamo $T(n) = T(2n/3) + 1$

Essendo $a = 1, b = 3/2$ ed $f(n) = 1 \rightarrow n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Dal momento che $f(n) = 1 = \theta(n^{\log_b a}) = \theta(1)$, possiamo applicare il caso 2 e concludere che $T(n) = \theta(\log_2 n)$

Esempio 3

Consideriamo $3T\left(\frac{n}{4}\right) + n\log_2 n$

Essendo $a = 3, b = 4$ ed $f(n) = n\log_2 n \rightarrow n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$

Dal momento che $f(n) = \Omega(n^{\log_4 3+\epsilon})$ dove $\epsilon \approx 0.2$, possiamo applicare il caso 3

Per n sufficientemente grande, $af(n/b) = 3(n/4)\ \log_2(n/4) \leq (3/4)n\ \log_2 n = cf(n)$ per $c = 3/4 \rightarrow T(n) = \theta(\log_2 n)$

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley
- Sedgewick: Algoritmi in C, Pearson, 2015
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione