Algoritmi e Strutture Dati

Modelli di calcolo e metodologie di analisi

Domenico Fabio Savo

Notazione asintotica

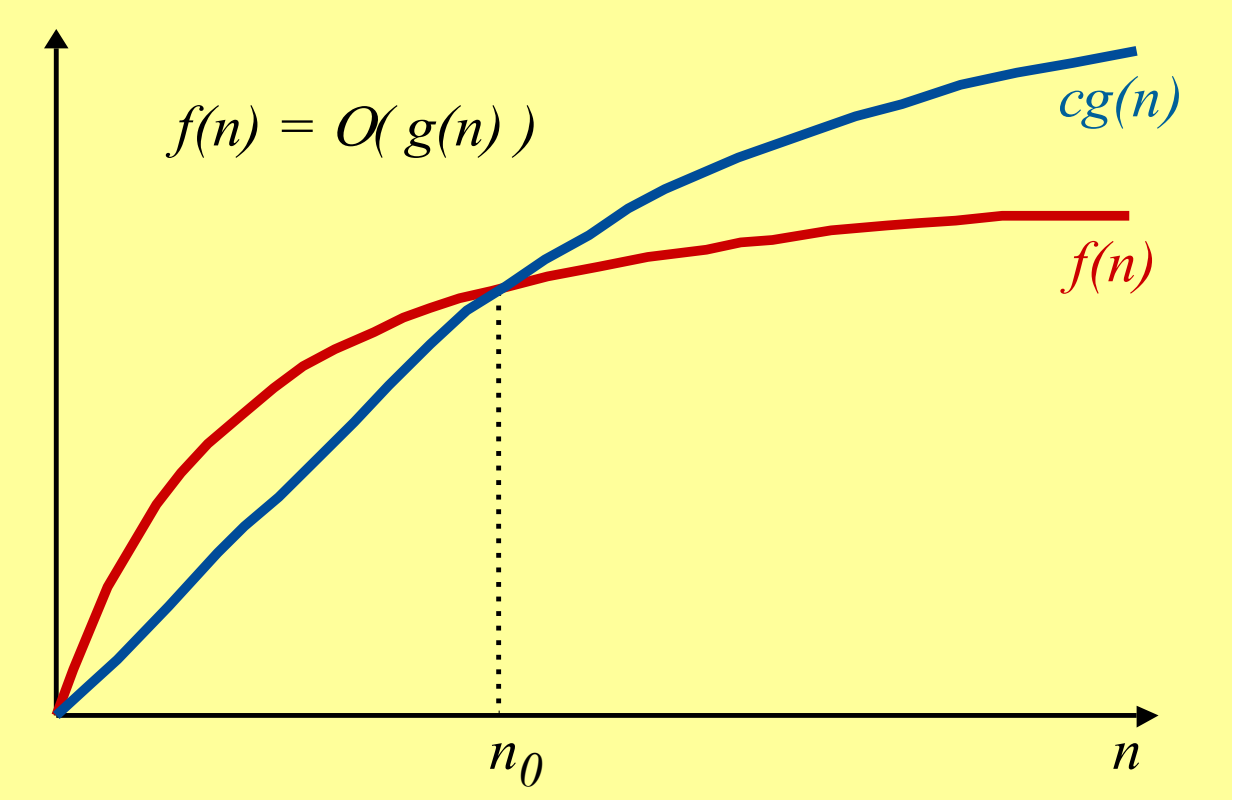
- f(n) = tempo di esecuzione / occupazione di memoria di un algoritmo su input di dimensione n

- La notazione asintotica è un'astrazione utile per descrivere l'ordine di grandezza di f(n) ignorando i dettagli non influenti, come costanti moltiplicative e termini di ordine inferiore

Notazione asintotica O

f(n) = O(g(n)) se $\exists$ due costanti $c>0$ e $n_0 \geq 0$ t.c.

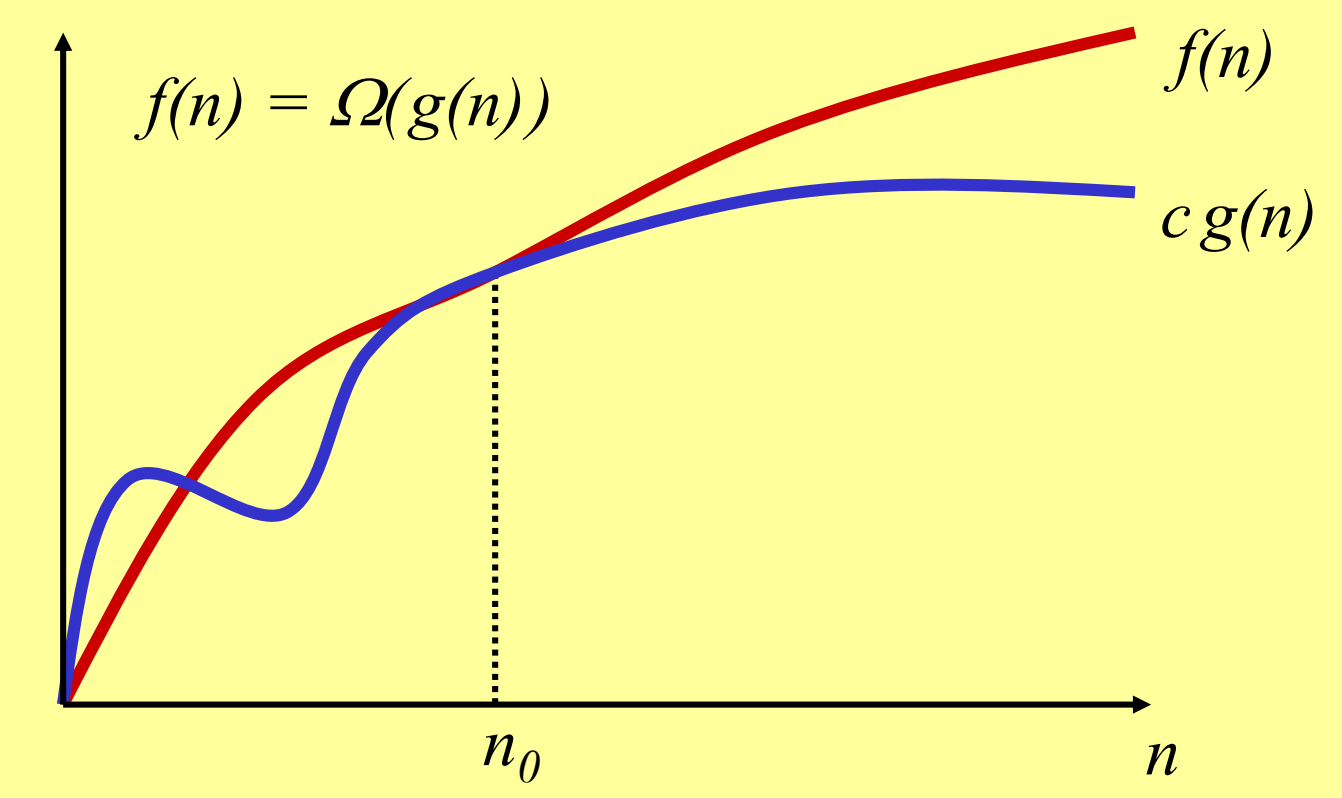
$f(n) \leq c\, g(n)$ per ogni $n \geq n_0$



Notazione asintotica Ω

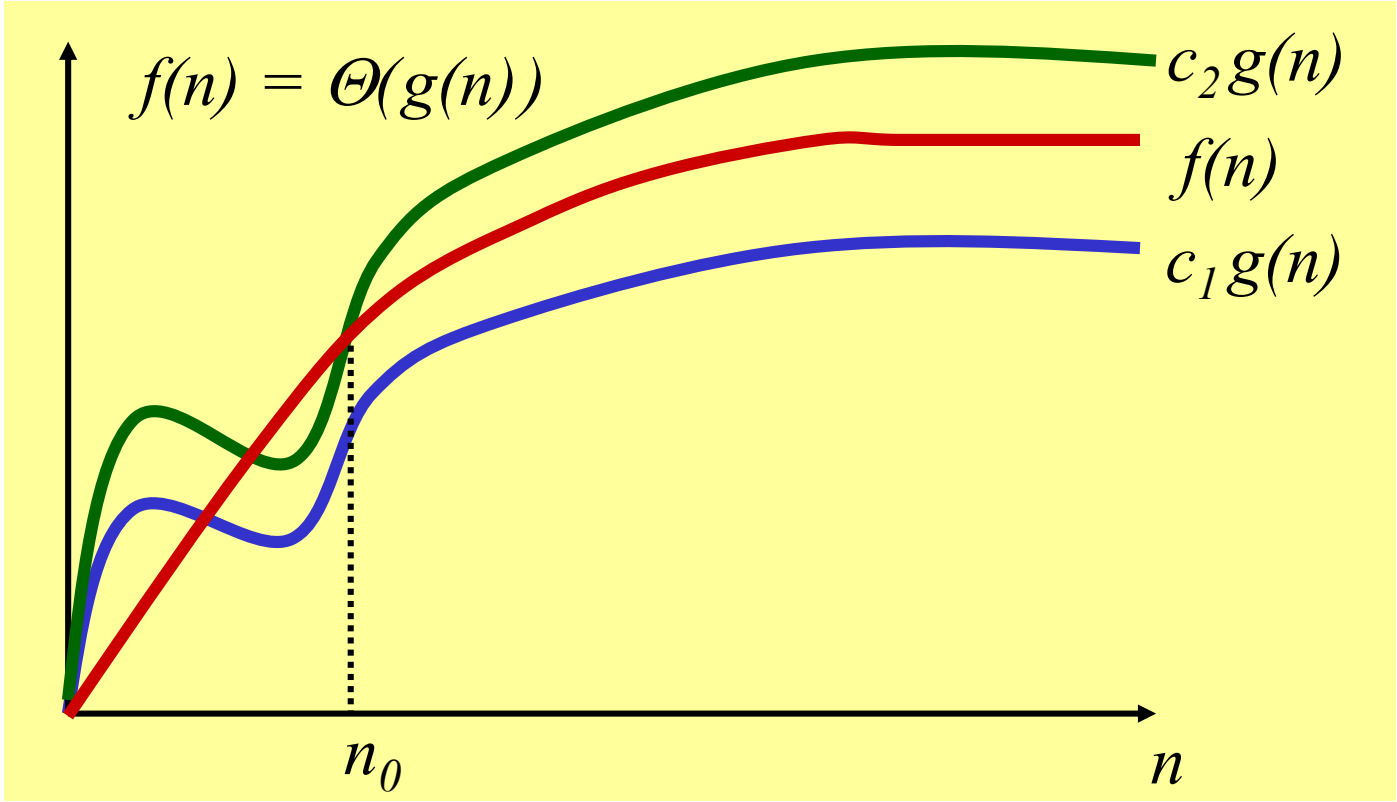
f(n) = Ω(g(n)) se ∃ due costanti c>0 e $n_0 \geq 0$ t.c.

$f(n) \geq c\ g(n)$ per ogni $n \geq n_0$



Notazione asintotica Θ

f(n) = Θ(g(n)) se ∃ tre costanti $c_1, c_2 > 0$ e $n_0 \geq 0$ t.c.
$c_1 g(n) \leq f(n) \leq c_2 g(n)$ per ogni $n \geq n_0$



Notazione asintotica Θ

Proprietà:

Date due funzioni f(n) e g(n), risulta che
f(n) = Θ(g(n)) se e solo se

- f(n) = O(g(n))
- f(n) = Ω(g(n))

Notazione asintotica: esempi

Consideriamo $\mathbf{g(n)=3n^2+10}$

- $\mathbf{g(n)=O(n^2)}$: scegliere c=4 e n_0=10
- $\mathbf{g(n)=\Omega(n^2)}$: scegliere c=1 e n_0=0
- $\mathbf{g(n)=\Theta(n^2)}$: infatti $g(n)= \Theta(f(n))$ se e solo se $g(n)= O(f(n))$ e $g(n)= \Omega(f(n))$
- $\mathbf{g(n)=O(n^3)}$ ma $\mathbf{g(n)\neq\Theta(n^3)}$

Metodi di analisi

Ricerca Sequenziale

Ricerca di un elemento x in una lista $\mathcal{L}$ non ordinata

```
    algoritmo ricercaSequenziale(lista L, elem x) → booleano
1.      for each ( y ∈ L ) do
2.          if ( y = x ) then return trovato
3.      return non trovato
```

Quanti confronti dobbiamo fare per trovare x in $\mathcal{L}$?

- Dipende da dove si trova x (all'inizio? Alla fine? E se non c'è?)
- Vorremmo una risposta che non sia «dipende».

Caso peggiore, migliore e medio

- Misureremo le risorse di calcolo (tempo di esecuzione / occupazione di memoria) usate da un algoritmo in funzione della dimensione n dell'istanza d'ingresso.
- A parità di dimensione, istanze diverse potrebbero richiedere risorse diverse
- Distinguiamo quindi ulteriormente tra analisi nel caso peggiore, migliore e medio

Caso peggiore

- Sia tempo(I) il tempo di esecuzione di un algoritmo sull'istanza I

$$T_{worst}(n) = \max_{\text{istanze I di dimensione n}} \{\text{tempo}(I)\}$$

- Intuitivamente, $\mathbf{T_{worst}(n)}$ è il tempo di esecuzione sulle istanze di ingresso che comportano più lavoro per l'algoritmo
- Da garanzia sulle prestazione

Caso migliore

- Sia tempo(I) il tempo di esecuzione di un algoritmo sull'istanza I

$$T_{best}(n) = \min_{\text{istanze I di dimensione n}} \{\text{tempo}(I)\}$$

- Intuitivamente, $\mathbf{T_{best}(n)}$ è il tempo di esecuzione sulle istanze di ingresso che comportano meno lavoro per l'algoritmo
- Non fornisce molte informazioni….

Caso medio

- Sia $\mathcal{P}(I)$ la probabilità di avere in ingresso un'istanza I

$$T_{avg}(n) = \sum_{\text{istanze I di dimensione n}} \{ \mathcal{P}(I) \text{ tempo}(I) \}$$

- Intuitivamente, $\mathbf{T_{avg}(n)}$ è il tempo di esecuzione nel caso medio (ovvero sulle istanze di ingresso "tipiche" per il problema)
- Richiede conoscenza di una distribuzione statistica dell'input

Analisi della ricerca sequenziale (1/3)

algoritmo `ricercaSequenziale`$(lista\ L,\ elem\ x) \rightarrow\ booleano$

1. **for each** $(\ y \in L\ )$ **do**
2. **if** $(\ y = x\ )$ **then return** trovato
3. **return** non trovato

Caso Migliore:

Nel caso migliore l'algoritmo esegue un solo confronto trovando immediatamente x in prima posizione.

Quindi: $T_{best}(n) = 1$

Analisi della ricerca sequenziale (2/3)

algoritmo `ricercaSequenziale`$(lista\ L,\ elem\ x) \rightarrow\ booleano$

1. **for each** $(\ y \in L\ )$ **do**
2. **if** $(\ y = x\ )$ **then return** trovato
3. **return** non trovato

Caso Peggiore:

Il caso peggiore ha luogo quando x si trova in ultima posizione oppure quando x non è in $\mathcal{L}$.

Quindi: $T_{worst}(n) = n$

Analisi della ricerca sequenziale (3/3)

algoritmo `ricercaSequenziale`$(lista\ L,\ elem\ x) \rightarrow\ booleano$

```
1.    for each ( y ∈ L ) do
2.        if ( y = x ) then return trovato
3.    return non trovato
```

Caso Medio:

Indichiamo con **I_i** il caso in cui x si trova nella posizione **i** di $\mathcal{L}$.
Supponendo che x può occupare ogni posizione con la medesima probabilità, per ogni **i** abbiamo che:
$\mathcal{P}(I_i) = 1/n$ e $tempo(I_i) = i$
Quindi:
$\mathbf{T_{avg}(n)} = \sum_{1 \le i \le n} \{\mathcal{P}(I_i)\ tempo(I)\} = \sum_{1 \le i \le n} (1/n)*i = \mathbf{(n+1)/2}$

Ricerca Binaria

Ricerca di un elemento x in un array $\mathcal{L}$ **ordinato**

```
algoritmo ricercaBinariaIter(array L, elem x) → booleano
1.    a ← 1
2.    b ← lunghezza di L
3.    while ( L[(a + b)/2] ≠ x ) do
4.        i ← (a + b)/2
5.        if ( L[i] > x ) then b ← i − 1
6.        else a ← i + 1
7.        if ( a > b ) then return non trovato
8.    return trovato
```

Confronta x con l'elemento centrale di $\mathcal{L}$ e prosegue nella metà sinistra o destra in base all'esito del confronto

Analisi ricerca binaria

$T_{best}(n) = 1$ l'elemento centrale è uguale a x

$T_{worst}(n) = ??$

$T_{avg}(n) = ??$

Analisi di algoritmi ricorsivi

Ricerca Binaria – versione ricorsiva

L'algoritmo di ricerca binaria può essere riscritto ricorsivamente come:

algoritmo `ricercaBinariaRic`(*elemento x, array L*) → *booleano*

1. n ← lunghezza di L
2. **if** (n = 0) **then return** "elemento non trovato"
3. i ← ⌈n/2⌉
4. **if** (L[i] = x) **return** "trovato"
5. **else if** (L[i] > x) **then return** `ricercaBinariaRic`(x, L[1,i-1])
6. **else return** `ricercaBinariaRic`(x, L[i+1,n])

Come analizzarlo?

Relazioni di ricorrenza

Il tempo di esecuzione può essere descritto tramite una relazione di ricorrenza:

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

simile a `fibonacci6`

Vari metodi per risolvere equazioni di ricorrenza: iterazione, sostituzione, teorema Master...

1. Metodo dell'iterazione

Idea:

Partendo dalla relazione di ricorrenza, andiamo a "srotolare" la ricorsione, ottenendo una sommatoria dipendente solo dalla dimensione n del problema iniziale.

1. Metodo dell'iterazione - esempio

$$T(k) = \begin{cases} c + T(\lceil k-1 \rceil/2) & \text{se } k>1 \\ 1 & \text{se } k=1 \end{cases}$$

Assumiamo che k sia una potenza di 2, per k = n, n/2, n/4 avremo:

$T(n) = c + T(n/2)$ $\quad$ $T(n/2) = c + T(n/4)$ $\quad$ $T(n/4) = c + T(n/8)$

da cui, attraverso sostituzioni successive, otteniamo:

$$T(n) = c + T(n/2) = 2c + T(n/4) = \left(\sum_{j=1\ldots i} c \right) + T(n/2^i) = i\,c + T(n/2^i)$$

Dobbiamo continuare finché $n/2^i = 1$, cioè $i=\log_2 n$.
Quindi:

$$T(n) = c \log n + T(1) = \mathbf{O(\log n)}$$

Analisi ricerca binaria

$T_{best}(n) = 1$ l'elemento centrale è uguale a x

$T_{worst}(n) = O(\log n)$ $x \notin \mathcal{L}$ oppure viene trovato all'ultimo confronto

$T_{avg}(n) = \log(n-1)+1/n$ assumendo che le istanze siano equidistribuite

2. Metodo della sostituzione

Idea:

“Intuire” la soluzione di una relazione di ricorrenza ed utilizzare l’induzione matematica per dimostrare che la soluzione è effettivamente quella che si è intuita.

2. Metodo della sostituzione - esempio

Consideriamo la seguente relazione di ricorrenza:

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

Ipotesi: $T(n) \leq c\, n$ (per una opportuna $c > 0$) quindi

$T(n) = O(n)$

Passo base: $T(1) \leq c\, 1$ (banalmente verificato)

Passo induttivo: $T(n) = T(\lceil n/2 \rceil) + n \leq c \lceil n/2 \rceil + n$

$\leq c(n/2) + n = (c/2 + 1)\, n$

Quindi $T(n) \leq c\, n$ per $c \geq 2$

Divide et Impera (1/2)

Tecnica molto potente e generale per la progettazione di algoritmi per la risoluzione di problemi.

Idea:

dividere i dati di ingresso in sottoinsiemi (*divide*), risolvere il ricorsivamente il problema sui sottoinsiemi, e ricombinare la soluzione dei sottoproblemi per ottenere la soluzione globale (*impera*).

Divide et Impera (2/2)

- Analizziamo la tecnica:
  - Abbiamo un problema di dimensione **n**;
  - Il problema viene suddiviso in **a** sottoproblemi di dimensione **n/b** ciascuno;
  - Suddividere il problema costa **f(n)**;
- La relazione di ricorrenza sarà quindi:

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

- l'algoritmo ricorsivo della ricerca binaria e l'algoritmo `fibonacci6` sono esempi di applicazione della tecnica *Divide et Impera*, con **a = 1, b = 2** e **f(n) = O(1)**.

Teorema Master (1/5)

Permette di analizzare algoritmi basati sulla tecnica del *divide et impera*, quindi con relazione di ricorrenza del tipo:

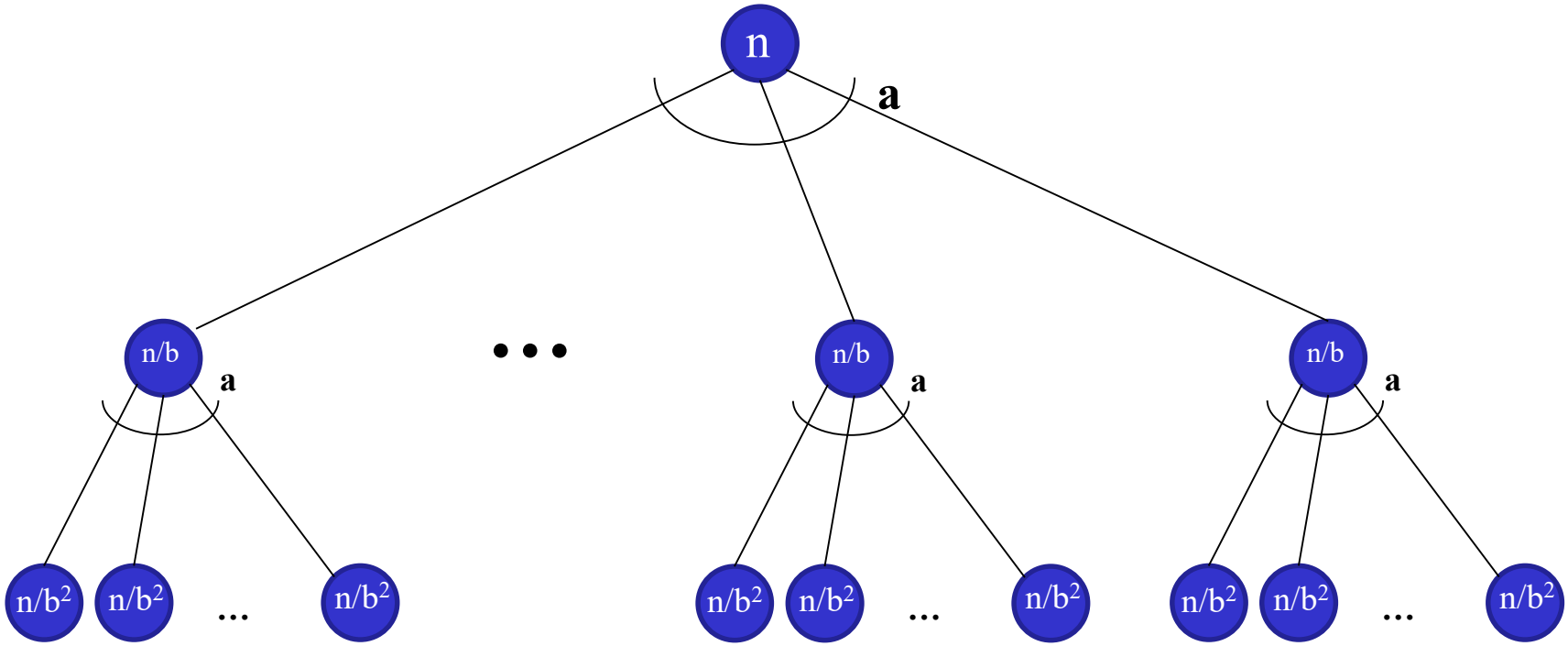
$$T(n) = \begin{cases} a\,T(n/b) + f(n) & \text{se } n>1 \\ 1 & \text{se } n=1 \end{cases}$$

Proviamo a risolvere la relazione…

Teorema Master (2/5)

Assumiamo per semplicità che **n** sia una potenza di **b** e che la ricorsione si fermi quando **n=1**

L'albero di ricorsione e:



Teorema Master (3/5)

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

Osservazione1: i sottoproblemi al livello **i** hanno dimensione $\mathbf{n/b^i}$. Quindi (escluso il tempo pe le chiamate ricorsive) il tempo speso al passo **i** è $\mathbf{f(n/b^i)}$.

Osservazione2: ogni nodo interno ha **a** figli. Quindi sul livello **i** dell'albero di ricorsione avremo $\mathbf{a^i}$ nodi.

Teorema Master (4/5)

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

Osservazione3: all'ultimo livello abbiamo che l'input $\mathbf{n/b^i = 1}$ quindi $\mathbf{n=b^i}$ e quindi $\mathbf{i = \log_b n}$
Segue che l'albero di ricorsione avrà al più altezza $\mathbf{\log_b n}$

Quindi possiamo riscrivere la nostra relazione di ricorrenza come:

$$T(n) = \sum_{i=0}^{log_b n} a^i f(n/bi)$$

Teorema Master (5/5)

La soluzione alla precedente equazione è data dal *teorema fondamentale delle ricorrenze*, anche detto *teorema master*.

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

ha soluzione:

1. $T(n) = \Theta(n^{\log_b a})$ se $f(n)=O(n^{\log_b a - \varepsilon})$ per $\varepsilon>0$

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 $f(n)=\Omega(n^{\log_b a + \varepsilon})$ per $\varepsilon>0$ e $a*f(n/b)\leq c*f(n)$ per $c<1$ e n sufficientemente grande

Teorema Master – Esempio 1

Consideriamo la relazione di ricorrenza della ricerca binaria:

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

Abbiamo:

a = 1, **b = 2** ed **f(n) = O(1)**

Siccome $\Theta(n^{\log_b a}) = \Theta(1)$, siamo nel caso 2 del teorema e quindi:

$$\mathbf{T(n)} = \Theta(n^{\log_b a} \log n) = \mathbf{\Theta(\log n)}.$$

Teorema Master – Esempio 2

Consideriamo la seguente relazione di ricorrenza

$$\mathbf{T(n) = 9T(n/3) + n}$$

Abbiamo:

a = 9, **b = 3** ed **f(n) = n**

Segue che $n^{\log_b a} = n^{\log_3 9} = n^2$ e quindi, dato che $f(n) = n = O(n^{2-\varepsilon})$, ad esempio per $\varepsilon = 1$, siamo nel caso 1 del teorema. Da cui:

$$\mathbf{T(n)} = \Theta(n^{\log_b a}) = \mathbf{\Theta(n^2)}.$$

Riepilogo

- Esprimiamo la quantità di una certa risorsa di calcolo (tempo, spazio) usata da un algoritmo in funzione della dimensione n dell'istanza di ingresso
- La notazione asintotica permette di esprimere la quantità di risorsa usata dall'algoritmo in modo sintetico, ignorando dettagli non influenti
- A parità di dimensione n, la quantità di risorsa usata può essere diversa, da cui la necessità di analizzare il caso peggiore o, se possibile, il caso medio
- La quantità di risorsa usata da algoritmi ricorsivi può essere espressa tramite relazioni di ricorrenza, risolvibili tramite vari metodi generali