Il paradigma "divide et impera"

La tecnica algoritmica del "divide et impera" consiste nel

- decomporre il problema in un piccolo numero di sotto-problemi, ciascuno dei quali è dello stesso tipo del problema originale ma è definito su un insieme di dati più piccolo rispetto a quello iniziale;
- risolvere **ricorsivamente** ciascun sotto-problema fino a che non si arriva a risolvere sotto-problemi di taglia così piccola da poter essere risolti direttamente (senza effettuare ulteriori chiamate ricorsive);
- combinare le soluzioni dei sotto-problemi al fine di ottenere una soluzione al problema di partenza.

Ordinamento per fusione: MergeSort

L'algoritmo MergeSort ordina in modo non decrescente una sequenza di numeri. L'idea dell'algoritmo è descritta di seguito.

- Se la sequenza contiene due o più elementi, la sequenza viene suddiviso in due parti ciascuna delle quali contiene circa la metà degli elementi
- Le due sottosequenze vengono ordinate ricorsivamente.
- Una volta ordinate, le due sottosequenze vengono fuse in un'unica sequenza ordinata.

MergeSort

Vediamo lo pseudocodice dell'algoritmo MergeSort che ordina un array. L'algoritmo riceve in input un array e due interi che delimitano la parte di array che si desidera ordinare. Inizialmente invochiamo MergeSort con *sinistra* uguale a 0 e *destra* uguale al numero di elementi dell'array -1.

```
MergeSort( a, sinistra, destra ):
  IF (sinistra < destra) {
    centro = (sinistra+destra)/2;
    MergeSort( a, sinistra, centro );
    MergeSort( a, centro+1, destra );
    Merge( a, sinistra, centro, destra );
  }
```

Per calcolare il tempo di esecuzione $T(n)$ dobbiamo tener conto del

- tempo per decomporre il problema in due sottoproblemi : O(1) in quanto occorre solo calcolare il centro
- tempo per eseguire le due chiamate ricorsive: $T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$
- tempo per fondere le due sequenze: ?

L'algoritmo Merge

- Possiamo fondere due sequenze ordinate $A =< a_0, \ldots, a_{n-1} >$ e $B =< b_0, \ldots, b_{m-1} >$ in modo da formare un'unica sequenza ordinata in tempo lineare in $n + m$.
- L'idea dell'algoritmo è il seguente:
  1. Scandiamo gli elementi delle due sequenze da sinistra verso destra utilizzando l'indice i per A e l'indice j per B.
  2. Fino a che $i \leq n$ e $j \leq m$, confrontiamo a_i con b_j. Se a_i è minore o uguale di b_j, a_i viene inserito alla fine della sequenza output e i viene incrementato di 1. Se a_i è maggiore di b_j, b_j viene inserito alla fine della sequenza output e j viene incrementato di 1.
  3. Al termine del ciclo precedente se $i \leq n$ trasferiamo uno dopo l'altro gli elementi $a_i, \ldots, a_n$ alla fine della sequenza output; se $j \leq m$ trasferiamo uno dopo l'altro gli elementi $b_j, \ldots, b_m$ alla fine della sequenza output.

L'algoritmo Fusione

- Ogni volta che eseguiamo un confronto tra un elemento di A ed uno di B, viene incrementato uno tra i due indici i e j. Di consequenza l'algoritmo effettua al più $n + m$ confronti.
- Sia $k \leq n + m$ il numero totale di confronti effettuati dall'algoritmo. Al termine di questi confronti, la sequenza output conterrà k elementi e in una delle due sequenze ci saranno $n + m - k$ elementi che dovranno essere trasferiti nella sequenza output.
- Il tempo totale per fondere le due sequenze ordinate è quindi lineare in $k + (n + m - k) = n + m$.

MERGE: ALGORITMO MERGE

Vediamo lo pseudocodice dell'algoritmo Merge che fonde due segmenti adiacenti di un array.

- Il primo segmento parte dalla locazione di indice sx e finisce nella locazione di indice cx
- il secondo segmento parte dalla locazione di indice $cx + 1$ e finisce nella locazione di indice dx

```
Merge( a, sx, cx, dx ):
  i = sx; j = cx+1; k = 0;
  WHILE ((i <= cx) && (j <= dx)) {
    IF (a[i] <= a[j]) {
      b[k] = a[i]; i = i+1;
    } ELSE {
      b[k] = a[j]; j = j+1;
    }
    k = k+1;
  }
  FOR ( ; i <= cx; i = i+1, k = k+1)
    b[k] = a[i];
  FOR ( ; j <= dx; j = j+1, k = k+1)
    b[k] = a[j];
  FOR (i = sx; i <= dx; i = i+1)
    a[i] = b[i-sx];
```

Analisi dell'algoritmo MergeSort

Ora che sappiamo qual è il tempo di esecuzione dell'algoritmo Merge possiamo completare l'analisi dell'algoritmo MergeSort. Indichiamo con $T(n)$ il suo tempo di esecuzione per un array input di n elementi. Il tempo $T(n)$ è dato da

- tempo per decomporre il problema in due sottoproblemi : $\Theta(1)$,
- tempo per eseguire le due chiamate ricorsive: $T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$,
- tempo per fondere le due sequenze: $cn = \Theta(n)$.

Si ha quindi $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + cn + c' = O(?)$.

Relazioni di ricorrenza

- Quando un algoritmo contiene una o più chiamate ricorsive a sé stesso, il suo tempo di esecuzione può essere spesso descritto da una *relazione di ricorrenza*.
- Una relazione di ricorrenza consiste in un'uguaglianza o in una disuguaglianza che descrive una funzione in termini dei suoi valori su input più piccoli.
- Esempio:

$$f(n) = \begin{cases} a & \text{se } n \leq 2 \\ 2f(n/3) + 4n & \text{altrimenti} \end{cases}$$

Relazioni di ricorrenza

- Vediamo come si scrive la relazione di ricorrenza che descrive il tempo di esecuzione $T(n)$ di un algorimo basato sulla tecnica del divide et impera per un input di dimensione n.

- Se la dimensione n del problema è minore di una certa costante c, l'algoritmo risolve direttamente il problema (senza effettuare chiamate ricorsive)
$$T(n) \leq c_0, \text{ per una certa costante } c_0 \ .$$

- Per $n > c$, il problema viene suddiviso in sottoproblemi: supponiamo che il problema venga suddiviso in α sottoproblemi, ognuno di dimensione n/β
- L'algoritmo viene invocato ricorsivamente per risolvere ciascuno di questi α sottoproblemi
- Le α soluzioni per questi sottoproblemi vengono ricombinate per ottenere la soluzione al problema originario.

Relazioni di ricorrenza

- Supponiamo che l'algoritmo impieghi al più tempo $d(n)$ per suddividere il problema di partenza in α sottoproblemi.
- Supponiamo che l'algoritmo impieghi al più tempo tempo $r(n)$ per ricombinare le soluzioni degli α sottoproblemi.
- Il tempo di esecuzione $T(n)$ per $n > c$ può essere descritto dalla relazione:

$$T(n) \leq \alpha T(n/\beta) + d(n) + r(n)$$

- Quindi possiamo scrivere la relazione di ricorrenza:

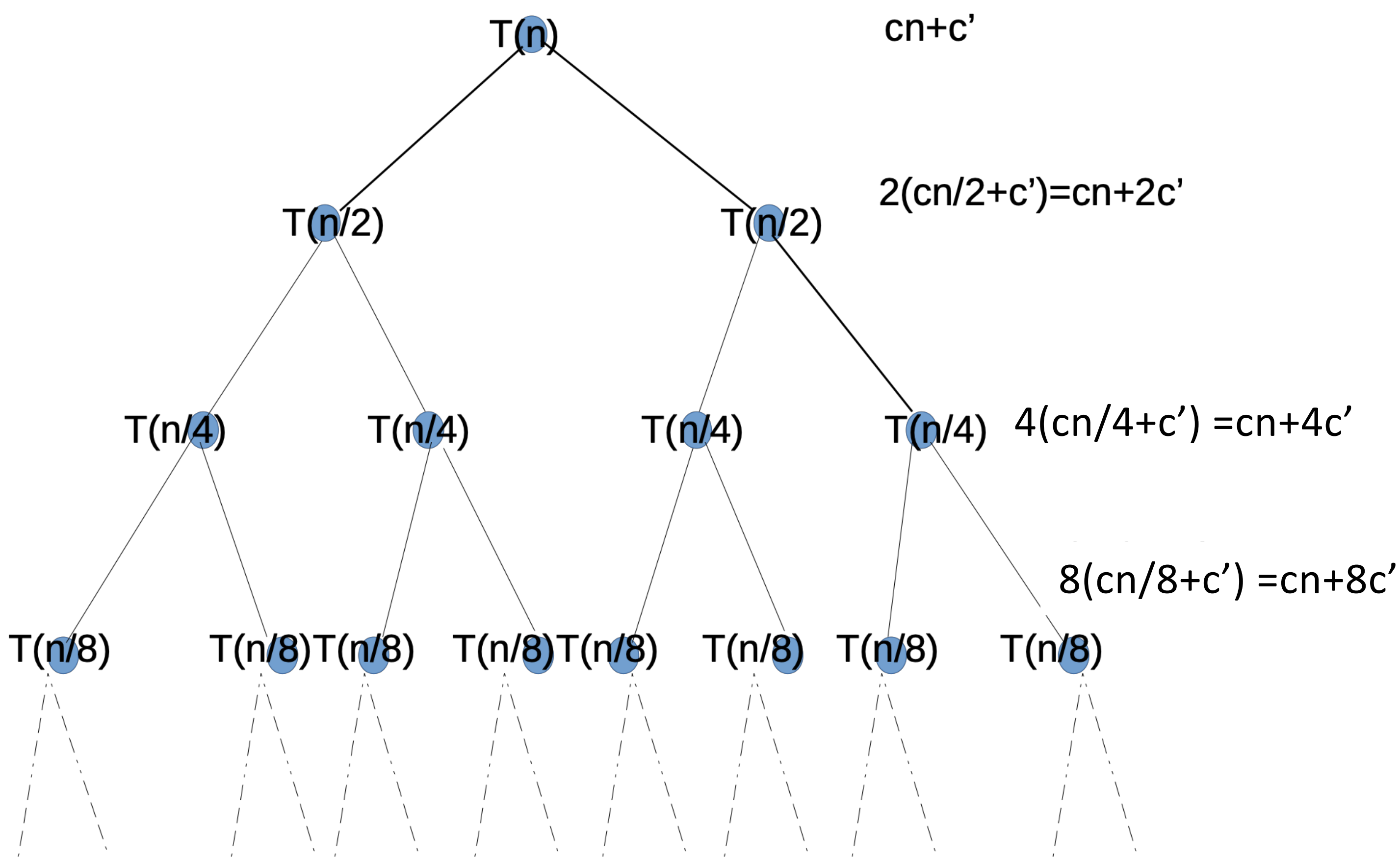
$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq c \\ \alpha T(n/\beta) + d(n) + r(n) & \text{altrimenti} \end{cases}$$

Tempo di esecuzione di MergeSort

- L'algoritmo MergeSort decompone il problema in due sottoproblemi di dimensione $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$ rispettivamente e impiega tempo costante per la decomposizione (deve semplicemente computare l'indice centrale in modo da individuare la fine e l'inizio dei due segmenti da ordinare) e tempo lineare per ricombinare le soluzioni dei sue sottoproblemi (deve fondere i due segmenti ordinati).
- Nell'analisi per semplicità assumiamo che n sia una potenza di 2 in modo che ogni chiamata ricorsiva divida il segmento su cui opera in due segmenti di uguale grandezza.
- Quindi

$$T(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ 2T(n/2) + cn + c' & \text{altrimenti} \end{cases}$$

TEMPO DI ESECUZIONE DI MERGESORT



- $\log_2 n+1$ livelli: nodi di profondità 0, nodi di profondità 1, ..., nodi di profondità log n
- Il costo totale associato al livello dei nodi di profondità $i \leq \log_2 n$ è $cn+2^i c'$
- L'ultimo livello contiene n foglie ciascuna delle quali rappresenta il tempo per risolvere il problema su un input di dimensione 1. In totale il lavoro richiesto da queste n chiamate ricorsive è $c_0 n$
- sommando su tutti i livelli $\sum_{i=0}^{\log_2 n-1}(cn+2^i c')+c_0 n = cn\log_2 n+(2^{\log_2 n}-1)c'+c_0 n = cn\log_2 n+c'n-c'+c_0 n$

→ T(n)=Θ(nlogn)

Tempo di esecuzione di MergeSort

- Dimostriamo con il metodo iterativo che il tempo di esecuzione è $\Theta(n \log n)$.
- Iteriamo la ricorrenza

$$
\begin{aligned}
T(n) &= c' + cn + 2T(n/2) = c' + cn + 2(c' + cn/2 + 2T(n/4)) \\
&= (1+2)c' + 2cn + 4T(n/4) = (1+2)c' + 2cn + 4(c' + cn/4 + 2T(n/8)) \\
&= (1+2+4)c' + 3cn + 8T(n/8) \\
\ldots &= (1+2+4+\ldots+2^{i-1})c' + icn + 2^i T\left(\frac{n}{2^i}\right) \\
&= (2^i - 1)c' + icn + 2^i T\left(\frac{n}{2^i}\right)
\end{aligned}
$$

- Quante volte dobbiamo iterare la ricorrenza per raggiungere il caso base?
- Ogni volta che applichiamo la ricorrenza il valore dell'argomento di T viene dimezzato per cui l'i-esima volta che applichiamo la ricorrenza l'argomento della funzione T diventa $\frac{n}{2^i}$. Raggiungiamo il caso base quando $\frac{n}{2^i} \leq 1$ e cioè non appena $2^i \geq n$. Ne consegue che ci fermiamo dopo che abbiamo applicato la ricorrenza $\log n$ volte.
- Dopo aver applicato la ricorrenza $\log n$ volte si ha

$$T(n) = c'(2^{\log n} - 1) + cn \log n + 2^{\log n} T(1) = c'n - c' + cn \log n + nc_0.$$

- Abbiamo dimostrato che $T(n) = \Theta(n \log n)$

Ricerca binaria: versione ricorsiva

```
RicercaBinariaRicorsiva( a,k,sinistra,destra ):
  IF (sinistra > destra) {
    RETURN -1;
  }
  c = (sinistra+destra)/2;
  IF (k == a[c]) {
    RETURN c;
  }
  IF (sinistra==destra) {
    RETURN -1;
  }
  IF (k <a[c]) {
    RETURN RicercaBinariaRicorsiva( a,k,sinistra,c-1 );
  } ELSE {
    RETURN RicercaBinariaRicorsiva( a,k,c+1,destra );
  }
```

Paradigma divide et impera

1. **Caso base**: Il segmento in cui stiamo effettuando la ricerca contiene al più un elemento oppure abbiamo trovato l'elemento al centro del segmento
2. **Decomposizione**: per decomporre occorre calcolare l'indice centrale c e vedere se k è minore o maggiore di $a[c]$
3. **Ricorsione e ricombinazione**: di fatto non occorre nessun lavoro di ricombinazione

Analisi mediante relazione di ricorrenza

- Se il segmento all'interno del quale stiamo cercando contiene al più un elemento oppure l'elemento cercato è quello centrale, allora l'algoritmo esegue un numero costante di operazioni $\leq c_0$.
- Altrimenti, il tempo richiesto è pari a una costante c più il tempo richiesto dalla ricerca dell'elemento in un segmento di dimensione al più pari alla metà di quello attuale.
  Il tempo totale di esecuzione $T(n)$ su un array di n elementi verifica la relazione di ricorrenza:

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \text{ oppure } k \text{ è l'elemento centrale} \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

- Applicando iterativamente la ricorrenza si ha

$$T(n) \leq T(n/2) + c \leq T(n/4) + c + c \leq \ldots \leq T(\frac{n}{2^i}) + ci$$

- Per $i = \log n$ abbiamo

$$T(n) \leq T(1) + c \log n \leq c_o + c \log n = O(\log n).$$

Analisi mediante relazione di ricorrenza

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \text{ oppure } k \text{ è l'elemento centrale} \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

- Risolviamo la relazione di ricorrenza con il metodo della sostituzione.
- Intuizione ci suggerisce che $T(n) = O(\log n)$. Dimostriamo questo limite con l'induzione. Dimostremo che $T(n) \leq c' \log n$ per una certa costante $c' > 0$ e per ogni $n \geq 2$.
- Base dell'induzione: per $n = 2$ si ha tempo minore o uguale di $T(1) + c = c_0 + c$ per cui basta scegliere $c' \geq c + c_0$.
- Passo induttivo: Supponiamo che per $2, \ldots, n-1$ il limite superiore sia verificato. Si ha quindi che $T(n/2) \leq c' \log(n/2)$.Di conseguenza

$$T(n) \leq T(n/2) + c \leq c' \log(n/2) + c = c' \log n - c' + c$$

- Affinché risulti $T(n) \leq c' \log n$ basta scegliere $c' \geq c$.
- Abbiamo quindi dimostrato che $T(n) \leq c'n$ per ogni $n \geq 2$ e $c' = \max\{c + c_0, c\} = c + c_0$.

Paradigma della ricerca binaria

Viene usato in diverse situazioni: per esempio, indovinare un numero positivo x con domande del tipo "$x \leq b$?", per un certo b

1. Chiedi se il numero intero x è $\leq 2^i$ per $i = 1, 2, \ldots$
2. Fermati non appena la risposta è *sì*.
3. Sia h l'indice in corrispondenza del quale otteniamo *sì* come risposta. Ovviamente si ha che $2^{h-1} < x \leq 2^h$ e di conseguenza $\log x \leq h < \log x + 1$
4. Effettua ricerca binaria nell'intervallo $[2^{h-1} + 1, 2^h]$
5. Intervallo contiene 2^{h-1} interi per cui ricerca binaria nell'intervallo richiede tempo $O(\log 2^{h-1}) = O(h) = O(\log x)$
6. In totale $O(\log x)$: $h = \lceil \log x \rceil$ domande fatte per individuare l'intervallo $[2^{h-1} + 1, 2^h]$ e h-1 domande per cercare x in $[2^{h-1} + 1, 2^h]$.

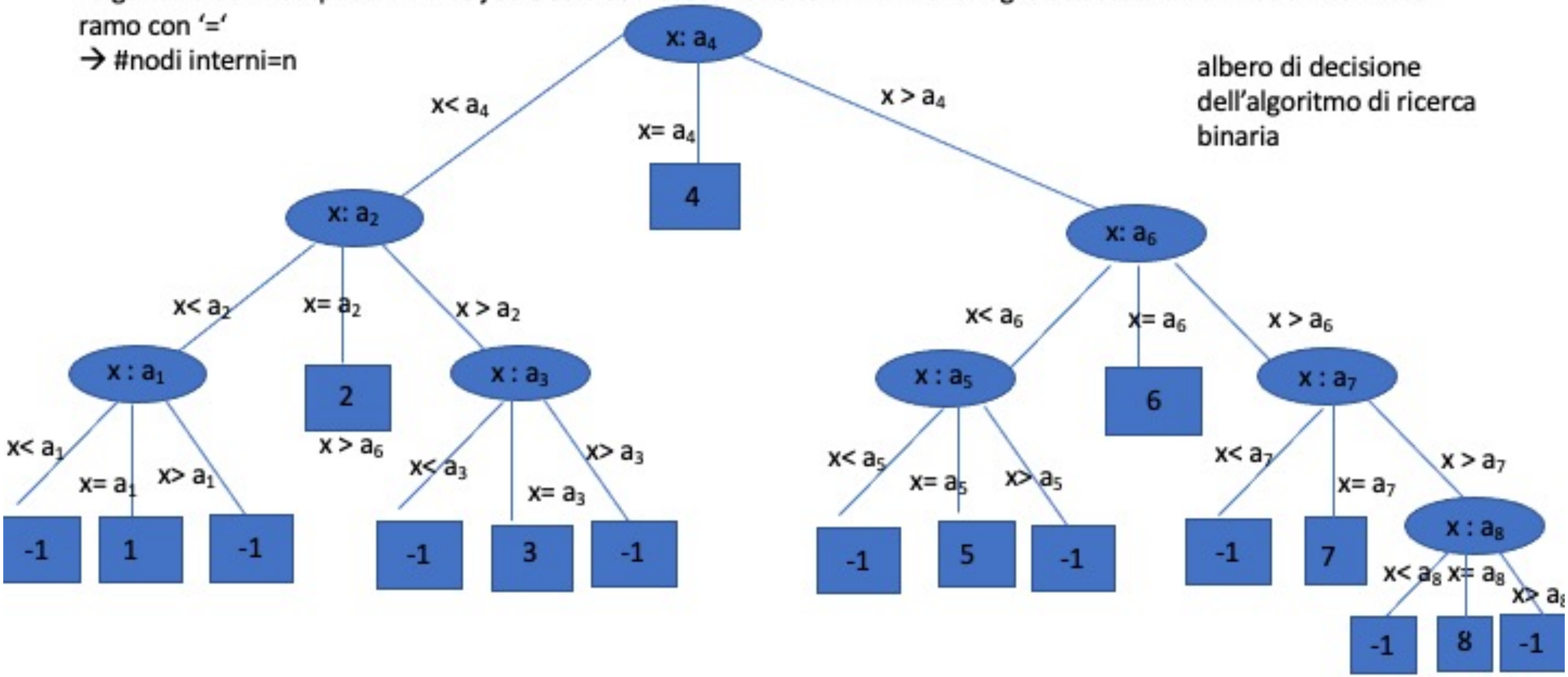
array ordinato A= <a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 >

albero di decisione descrive tutte le possibili sequenze di confronti che potrebbe fare l'algoritmo.

nell'albero di decisione i nodi interni rappresentano i confronti, mentre le foglie rappresentano gli output prodotti. → altezza albero=numero confronti nel caso pessimo

l'algoritmo dà in output un indice j se e solo se l'esecuzione termina in una foglia attaccata ad un nodo interno da ramo con '='

→ #nodi interni=n

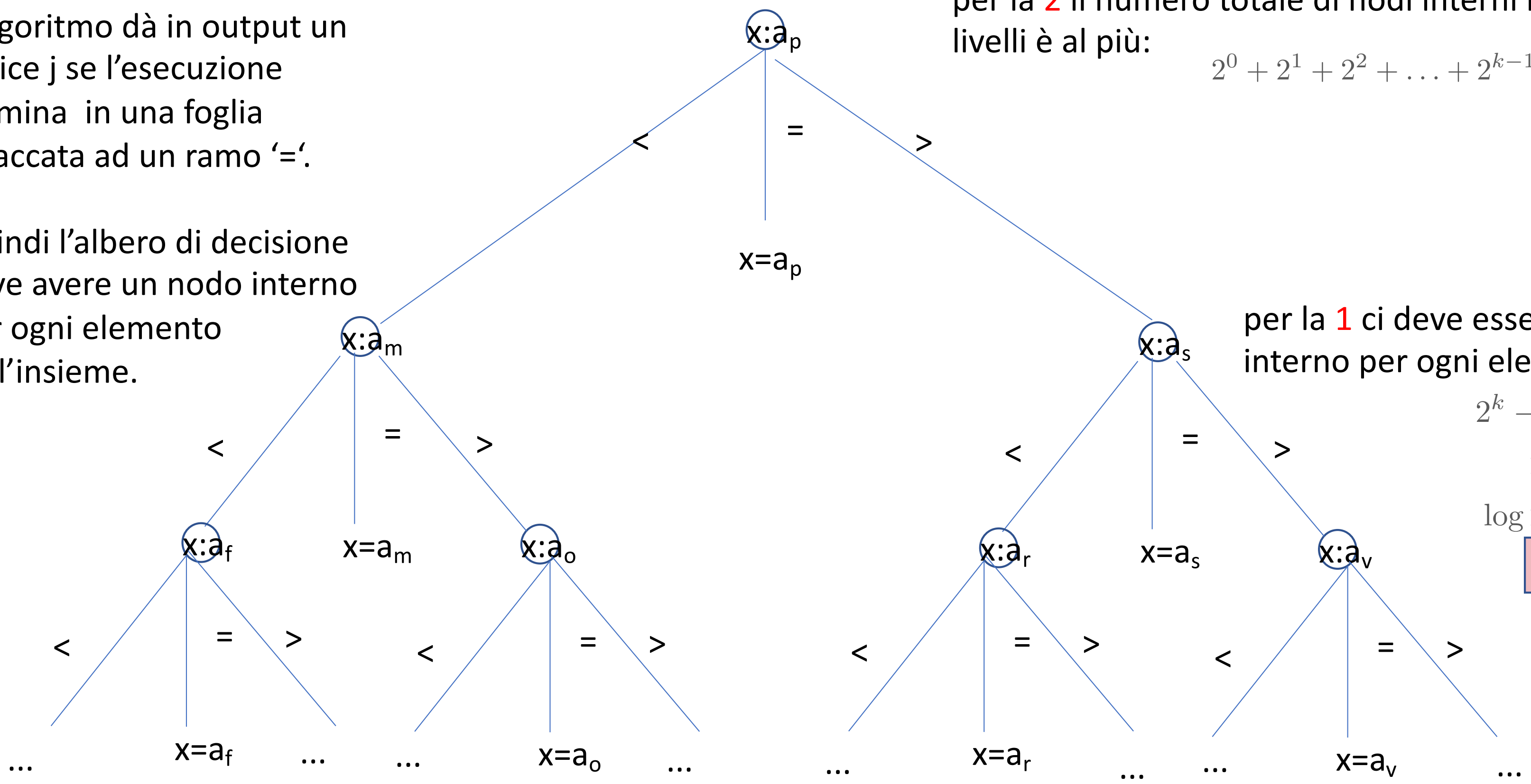


albero di decisione dell'algoritmo di ricerca binaria

Lower bound sulla ricerca su un insieme ordinato

1 L'algoritmo dà in output un indice j se l'esecuzione termina in una foglia attaccata ad un ramo '='.

Quindi l'albero di decisione deve avere un nodo interno per ogni elemento dell'insieme.



per la 2 il numero totale di nodi interni nei primi k livelli è al più:

$$
\begin{aligned}
2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} &= \sum_{i=0}^{k-1} 2^i \\
&= \frac{1-2^k}{1-2} \\
&= 2^k - 1
\end{aligned}
$$

per la 1 ci deve essere un nodo interno per ogni elemento:

$$
\begin{aligned}
2^k - 1 &\geq n \\
2^k &\geq n+1 \\
\log 2^k &\geq \log(n+1) \\
\Aboxed{k &\geq \log(n+1)}
\end{aligned}
$$

2 Ogni nodo interno "produce" al più due nodi interni per cui ci sono al più 2^i nodi interni (confronti) a livello i.
NB: i nodi nell'ultimo livello sono foglie ciascuna delle quali o è attaccata ad un ramo '=' (elemento trovato) o ad un ramo '<' o '>' (elemento non trovato).

Conseguenza: l'algoritmo di ricerca binaria è asintoticamente ottimo.