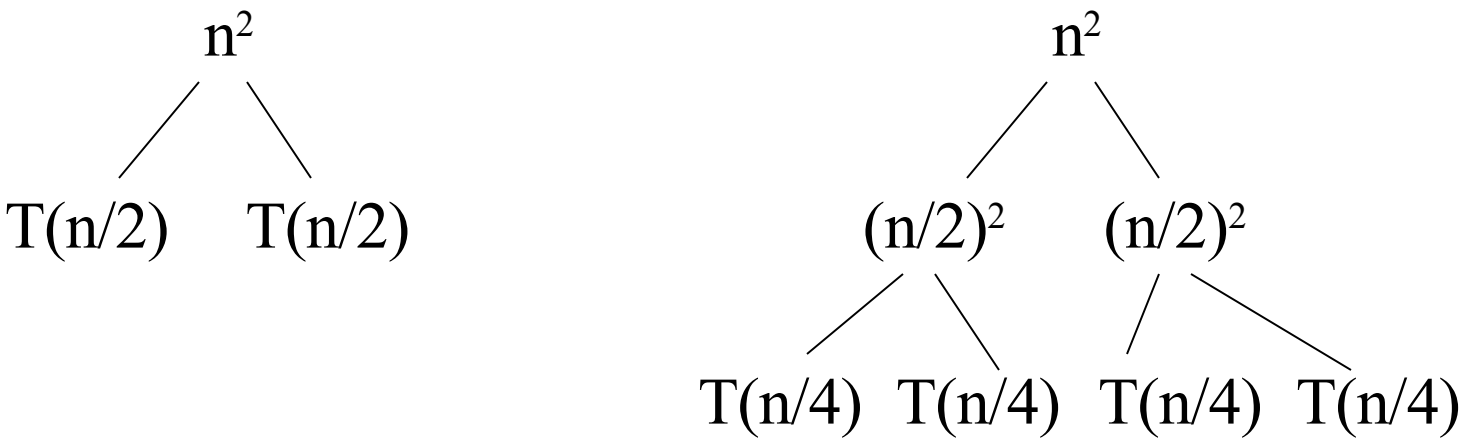
Albero di Riscorsione

Albero di ricorsione

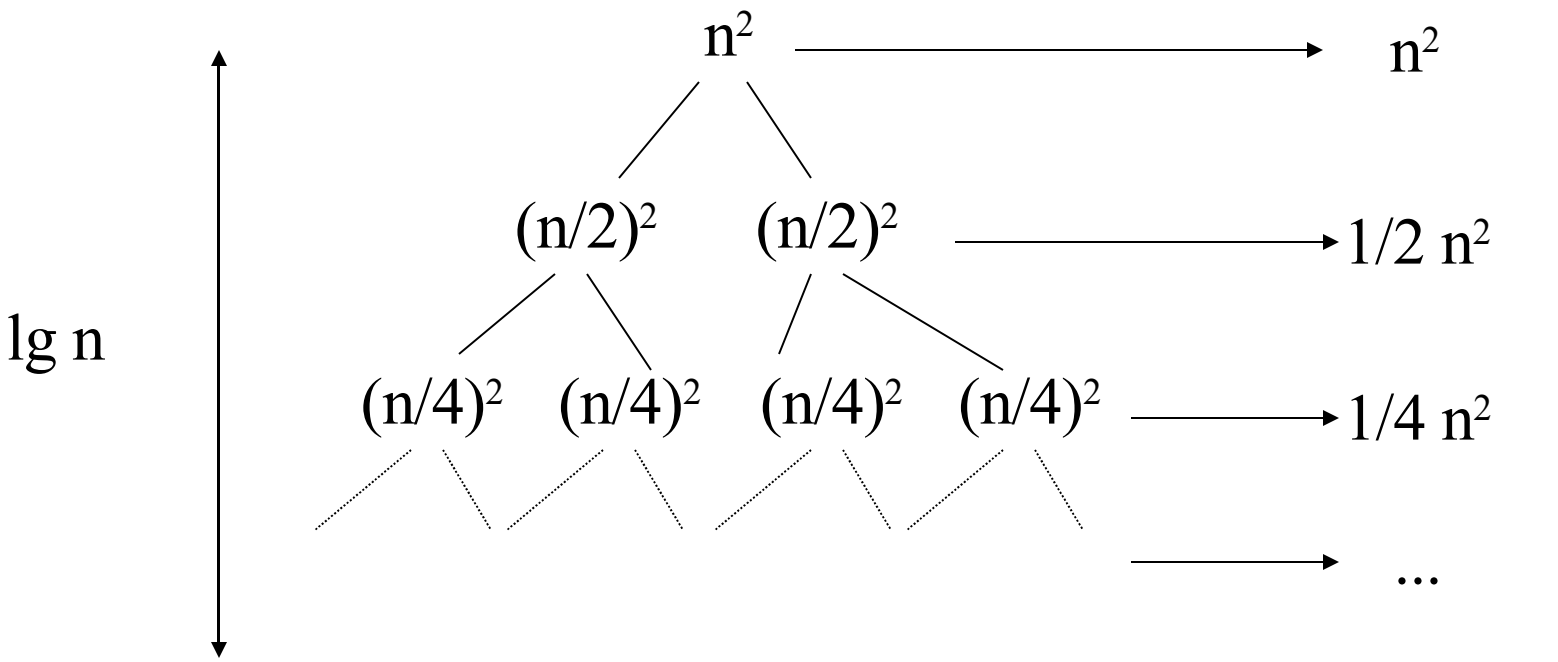
- Un albero di ricorsione è un modo di visualizzare cosa accade in un algoritmo divide et impera
- L'etichetta della radice rappresenta il costo non ricorsivo della fase divide e combina
- la radice ha tanti figli quanti sottoproblemi generati ricorsivamente
- ogni sottoproblema ha come etichetta il costo della fase divide e combina del problema per la nuova dimensione
- per ogni livello si indica la somma dei costi
- si conteggia il numero di livelli necessari per terminare la ricorsione

Esempio di albero di ricorsione

- Per $T(n)=2T(n/2)+n^2$ si ha:



Esempio di albero di ricorsione



$$\text{Totale} = n^2 \sum_{i=0..\ \lg n} 1/2^i < n^2 \sum_{i=0..\infty} 1/2^i = 2\ n^2$$

$$\text{Totale} = \Theta(n^2)$$

Altezza di albero di ricorsione

- Per il calcolo dell'altezza dell'albero di ricorsione, ovvero del numero di passi necessario per terminare la ricorsione, si considera il cammino più lungo dalla radice ad una foglia
- $n \rightarrow 1/2n \rightarrow (1/2)^2 n \rightarrow (1/2)^3 n \rightarrow \ldots \rightarrow 1$
- Dato che:

  $(1/2)^i n = 1$

  $2^i = n$

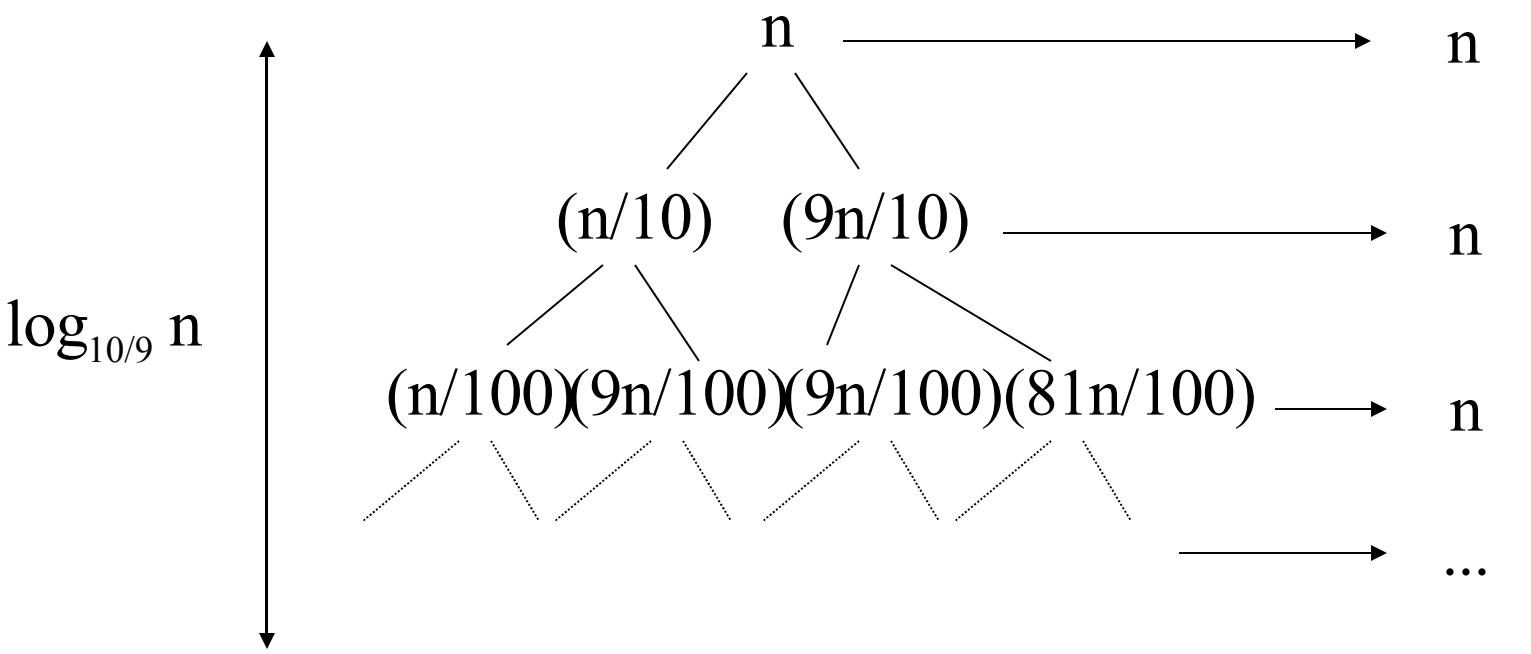
  $\log_2 2^i = \log_2 n$

  $i = \lg n$
- ovvero $h = \lg n$

Concetti intuitivi sul caso medio QuickSort

- Si consideri un partizionamento “abbastanza” sbilanciato che divida il problema in due sotto sequenze di dimensione 1/10 e 9/10
- $T(n)=T(n/10)+T(9n/10)+n$
- dall’albero di ricorsione si ha che ogni livello costa n
- il numero di livelli è:
  - $n \rightarrow 9/10\ n \rightarrow (9/10)^2\ n \rightarrow (9/10)^3\ n \rightarrow \ldots \rightarrow 1$
  - ovvero $(9/10)^i\ n = 1$
  - $n = (10/9)^i$
  - $i = \log_{10/9} n = \log n / (\log 10/9) = \log n / 0.0457.. = 21.85 \log n$

Albero di ricorsione



Totale = $\Theta(n \lg n)$

Forte sbilanciamento costante

- Con un partizionamento di 1 a 9 il quicksort viene eseguito in un tempo asintoticamente eguale al caso migliore di partizionamento bilanciato
- Anche una suddivisione 1 a 99 mantiene la proprietà di esecuzione in tempo $n \lg n$ (sebbene con un fattore moltiplicativo 300 rispetto al caso bilanciato)
- Il motivo è che qualsiasi suddivisione in proporzioni costanti produce un albero di altezza $\lg n$ in cui ogni livello costa $\Theta(n)$
- Tuttavia non possiamo aspettarci un partizionamento costante nei casi reali

Alternanza pessimo-ottimo

- Si suppone che in media si abbiano delle partizioni buone e cattive distribuite durante lo svolgimento dell'algoritmo
- Si suppone per semplificare che le partizioni buone si alternino a quelle cattive
- Si suppone che la partizione buona sia la migliore possibile, cioè quella derivante da un bilanciamento perfetto
- E quella cattiva la peggiore possibile, cioè quella derivante da una partizione 1, n-1

Calcolo complessita caso medio

- Si ha pertanto un primo passo in cui si partiziona il problema in due sottosequenze 1 e n-1 di costo n
- poi un secondo passo in cui
  - si risolve in tempo costante il problema sulla sottosequenza unitaria
  - si partiziona ulteriormente la sequenza di lunghezza n-1 in due sottosequenze di (n-1)/2 e (n-1)/2 elementi ad un costo di n-1
- Il risultato è di aver raddoppiato il numero di passi rispetto al caso di partizionamento ottimo, ma di essere sostanzialmente arrivati nella stessa situazione di buon partizionamento (n/2 contro (n-1)/2) con un costo complessivo $\Theta(n)$

Concetti intuitivi sul caso medio

- In sintesi il costo del QuickSort per un caso medio in cui si alternano partizioni buone a cattive è dello stesso ordine di grandezza del caso migliore, e cioè O(n lg n) anche se con costanti più grandi

Algoritmo Randomizzato

- Il caso medio si ottiene quando una qualsiasi permutazione dei dati in ingresso è *equiprobabile*
- tuttavia nei casi reali questo può non essere vero: ad esempio ci sono situazioni in cui i dati sono parzialmente ordinati
- in questi casi le prestazioni del quicksort peggiorano
- piuttosto che ipotizzare una distribuzione dei dati in ingresso è possibile *imporre* una distribuzione
- si chiama *randomizzato* un algoritmo il cui comportamento è determinato non solo dall'ingresso ma anche da un generatore di numeri casuale

Randomized QuickSort One Pass

- Prima di avviare la procedura di sort si esegue una permutazione casuale degli elementi del vettore di ingresso
- Il calcolo della permutazione *deve* avere un costo O(n) altrimenti diventa una procedura computazionalmente dominante rispetto alle altre operazioni

- Nessuna sequenza in ingresso comporta il caso peggiore: ma è sempre possibile avere una cattiva permutazione prodotta dal generatore casuale
- ...ma l'onere adesso si e' spostato sull'algoritmo di generazione (pseudo)casuale, non sui dati in ingresso
- ...ovvero si può garantire che si generino pochissime permutazioni cattive in media

PseudoCodice per la permutazione casuale

```
Randomize(A,p,r)
1  for i ← 1 to length[A]
2  do  j ← Random(p,r)
3      A[i] ↔ A[j]
```

Randomized QuickSort Random pivot

- Ad ogli passo, prima di partizionare l'array si scambia A[p] con un elemento scelto a caso in A[p..r]
- E' come se si scegliesse cioè l'elemento pivot in modo casuale ad ogni partizionamento
- Questo assicura che il pivot sia in modo equiprobabile uno qualsiasi degli elementi in A[p..r]
- Ci si aspetta cosi' che in media la suddivisione dell'array sia ben bilanciata

PseudoCodice

```
Randomized-Partition(A,p,r)
1  i ← Random(p,r)
2  A[p] ↔ A[i]
3  return Partition(A,p,r)

Randomized-QuickSort(A,p,r)
1 if p < r
2 then q ← RandomizedPartition(A,p,r)
3      Randomized-Quicksort(A,p,r)
4      Randomized-Quicksort(A,q+1,r)
```

Variante 1

- Se la sequenza e' gia' ordinata in senso crescente e' svantaggioso prendere il primo elemento, se ordinata in senso decrescente e' svantaggioso l'ultimo:
- L'elemento pivot e' scelto come l'elemento di valore intermedio fra il primo, l'ultimo e un terzo elemento con indice mediano
- In questo modo per sequenze quasi ordinate si prende come pivot con alta probabilita' proprio l'elemento che realizzera' un partizionamento bilanciato

Variante 2

- Chiamate ricorsive per dividere sequenze piccole (una decina di elementi) sono costose:
- Il passo base non e' quando si hanno sequenze di 1 elemento ma quando la sequenza ha una dimensione piccola prefissata
- La sequenza rimanente viene ordinata con un altro algoritmo che abbia complessita' con costanti moltiplicative piccole e che abbia buone prestazioni per sequenze quasi ordinate (ex. insertion sort)