

Quicksort
(vs Mergesort)
Relazioni di ricorrenza

24 marzo 2023

Algoritmi basati sulla tecnica Divide et Impera

In questo corso:

- Ricerca binaria
- Mergesort (ordinamento)
- Quicksort (ordinamento)
- Moltiplicazione di interi
- Moltiplicazione di matrici

NOTA: nonostante la tecnica *Divide et impera* sembri così «semplice» ben due «top ten algorithms of the 20° century» sono basati su di essa:

Fast Fourier Transform (FFT)

Quicksort

Ordinamento

INPUT: un insieme di n oggetti a_1, a_2, \dots, a_n
presi da un dominio totalmente ordinato secondo \leq

OUTPUT: una permutazione degli oggetti a'_1, a'_2, \dots, a'_n tale che
$$a'_1 \leq a'_2 \leq \dots \leq a'_n$$

Applicazioni:

- Ordinare alfabeticamente lista di nomi, o insieme di numeri, o insieme di compiti d'esame in base a cognome studente
- Velocizzare altre operazioni (per es. è possibile effettuare ricerche in array ordinati in tempo $O(\log n)$)
- Subroutine di molti algoritmi (per es. *greedy*)
-

Algoritmi per l'ordinamento

Data l'importanza, esistono svariati algoritmi di ordinamento, basati su tecniche diverse:

Insertionsort

Selectionsort

Heapsort

Mergesort

Quicksort

Bubblesort

Countingsort

.....



Figure 2.1: Sorting a hand of cards using insertion sort.

Ognuno con i suoi aspetti positivi e negativi.

Il Mergesort e il Quicksort sono entrambi basati sulla tecnica Divide et Impera, ma risultano avere differenti prestazioni

Ricordate che di **ogni** algoritmo che studieremo dovrete sapere:

- Quale problema computazionale risolve
- Spiegarne il funzionamento tramite **pseudo-codice** (o verbalmente, ma in maniera precisa) facendo riferimento alla
- **Tecnica** di progettazione su cui si basa
- Saperlo **eseguire** su un dato input
- Spiegare perché funziona (**correttezza**)
- **Analizzarne** il **tempo** di esecuzione, ed eventualmente lo **spazio** di memoria ausiliaria utilizzato

Quicksort

Nota: sul libro di testo trovate solo una versione randomizzata (cap. 13).
Potete fare riferimento al libro di Cormen, Leiserson, Rivest, (Stein)
Introduzione agli algoritmi, o ad altri testi consigliati.

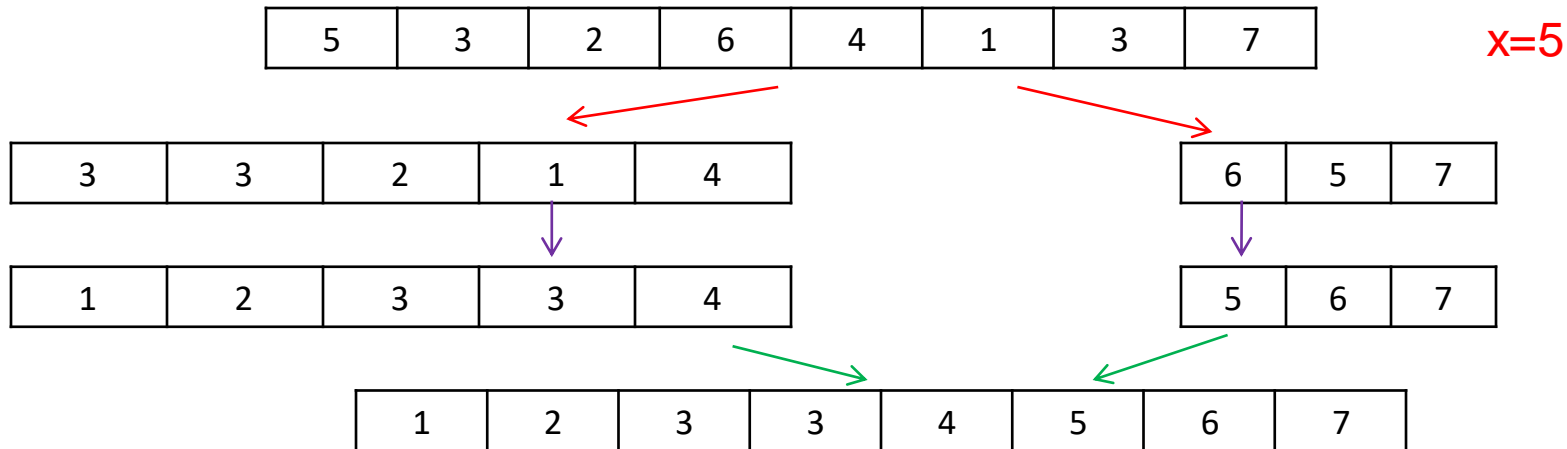
Quicksort

Dato un array di n elementi

I) **Divide**: scegli un elemento x dell'array (detto "pivot" o perno) e **partiziona** la sequenza in elementi $\leq x$ ed elementi $\geq x$

II) Risolvi i due sottoproblemi **ricorsivamente**

III) **Impera**: restituisci la concatenazione dei due sotto-array ordinati



Scelta del pivot

L'algoritmo funziona per qualsiasi scelta (primo / ultimo / ...), ma se vogliamo algoritmo “deterministico” devo fissare la scelta; nel seguito sceglieremo il **primo**.

Altrimenti: scelgo “random” e avrò “algoritmi randomizzati”
(vedi Kleinberg & Tardos, cap. 13)

Partizionamento

Partiziona l'array in elementi $\leq x$ ed elementi $\geq x$

Banalmente:

scorro l'array da 1 ad n e inserisco gli elementi \leq pivot in un nuovo array e quelli \geq del pivot in un altro nuovo array

Però:

- 1) avrei bisogno di array ausiliari
- 2) di che dimensione? I due sotto-array avrebbero un numero variabile di elementi

Partizione “in loco”

Partition:

- pivot = $A[1]$
- Scorri l'array da destra verso sinistra (con un indice j) e da sinistra verso destra (con un indice i) :
 - da destra verso sinistra, ci si ferma su un elemento \leq del pivot
 - da sinistra verso destra, ci si ferma su un elemento \geq del pivot;
- Scambia gli elementi
- Riprendi la scansione finché i e j si incrociano

Partition (Hoare 1962)

```
Partition (A, p, r)
x = A[p]
i = p-1
j = r+1
while True
    do repeat j=j-1 until A[j] ≤ x
    repeat i=i+1 until A[i] ≥ x
    if i < j
        then scambia A[i] ↔ A[j]
    else return j
```

Esiste un diverso algoritmo per il partizionamento dovuto a N. Lomuto ed esistono piccole varianti di questo (che potreste incontrare cambiando libro di testo)

Attenzione: repeat op until cond

significa che: eseguo op; se cond è verificata esco, altrimenti ripeto.

j si ferma su un elemento $\leq x$; i si ferma su un elemento $\geq x$.

Partizione in loco: un esempio

5	3	2	6	4	1	3	7
---	---	---	---	---	---	---	---

pivot = 5

5	3	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Scambia 3 con 5

3	3	2	6	4	1	5	7
---	---	---	---	---	---	---	---

Scambia 1 con 6

3	3	2	6	4	1	5	7
---	---	---	---	---	---	---	---

3	3	2	1	4	6	5	7
---	---	---	---	---	---	---	---

3	3	2	1	4	6	5	7
---	---	---	---	---	---	---	---

≤ 5

≥ 5

Restituisce $q = j$. Gli elementi $< x$ staranno a sinistra; gli elementi $> x$ a destra; quelli $= x$ possono stare sia a sinistra che a destra.

Partition: un altro esempio

5	3	2	6	4	1	5	7
---	---	---	---	---	---	---	---

pivot = 5

5	3	2	6	4	1	5	7
---	---	---	---	---	---	---	---

Scambia 5 con 5

5	3	2	6	4	1	5	7
---	---	---	---	---	---	---	---

Scambia 1 con 6

5	3	2	6	4	1	5	7
---	---	---	---	---	---	---	---

5	3	2	1	4	6	5	7
---	---	---	---	---	---	---	---

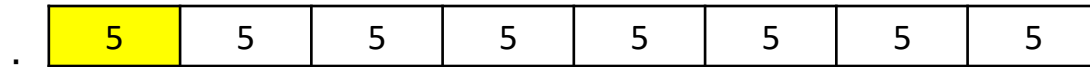
5	3	2	1	4	6	5	7
---	---	---	---	---	---	---	---

≤ 5

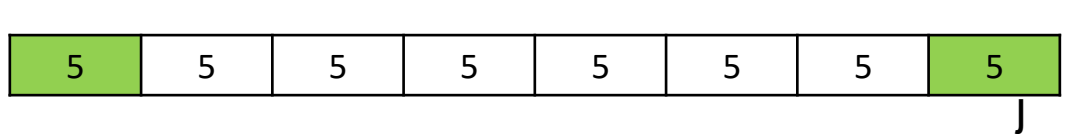
≥ 5

Restituisce $q = j$. Gli elementi $< x$ staranno a sinistra; gli elementi $> x$ a destra; quelli $= x$ possono stare sia a sinistra che a destra.

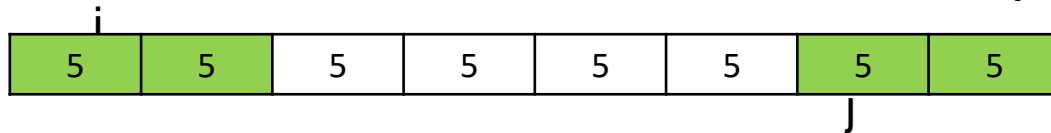
Partition su un array di elementi tutti uguali



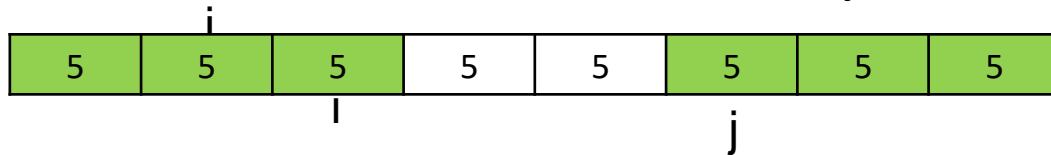
pivot = 5



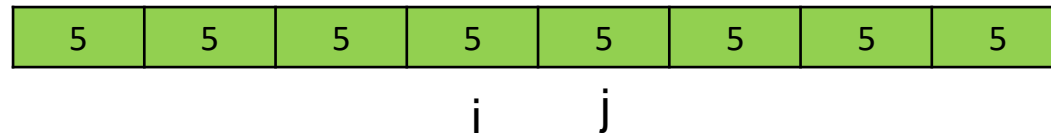
Scambia 5 con 5



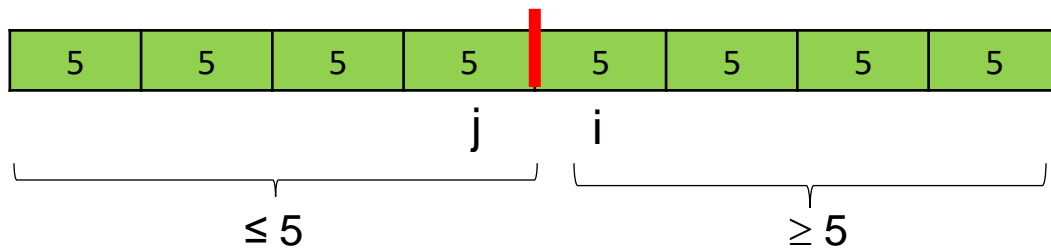
Scambia 5 con 5



Scambia 5 con 5



Scambia 5 con 5



Restituisce $q = j$

Correttezza di Partition

Perché funziona?

Ad ogni iterazione (quando raggiungo il while):

la “parte verde” di **sinistra** (da p ad i) contiene elementi ≤ 5 ;

la “parte verde” di **destra** (da j a r) contiene elementi ≥ 5 .

Tale affermazione è vera all’inizio e si mantiene vera ad ogni iterazione (per induzione); alla fine implica la correttezza di Partition.

Nota: `Partition` restituisce $q \geq p$: al massimo j si ferma sul primo elemento, che è \leq pivot.

Analisi Partition

Il tempo di esecuzione è $\Theta(n)$

```
Quicksort (A, p, r)
  if p < r then
    q = Partition (A,p,r)
    Quicksort(A, p, q)
    Quicksort(A, q+1, r)
```

Correttezza: la concatenazione di due array ordinati in cui l'array di sinistra contiene elementi minori o uguali degli elementi dell'array di destra è un array ordinato

Analisi: $T(n) = \Theta(n) + T(k) + T(n-k)$

dove k sono gli elementi da p a q (e $n-k$ i rimanenti da $q+1$ a r) con $1 \leq k \leq n-1$. *Ricorda:* Partition restituisce $q \geq p$.

Analisi Quicksort (caso peggiore)

Un primo caso: ad ogni passo il pivot scelto è il minimo o il massimo degli elementi nell'array (la partizione è $1 \mid n-1$):

$$T_{\text{worst}}(n) = T_{\text{worst}}(n-1) + T_{\text{worst}}(1) + \Theta(n)$$

essendo $T_{\text{worst}}(1) = \Theta(1)$

$$T_{\text{worst}}(n) = T_{\text{worst}}(n-1) + \Theta(n)$$

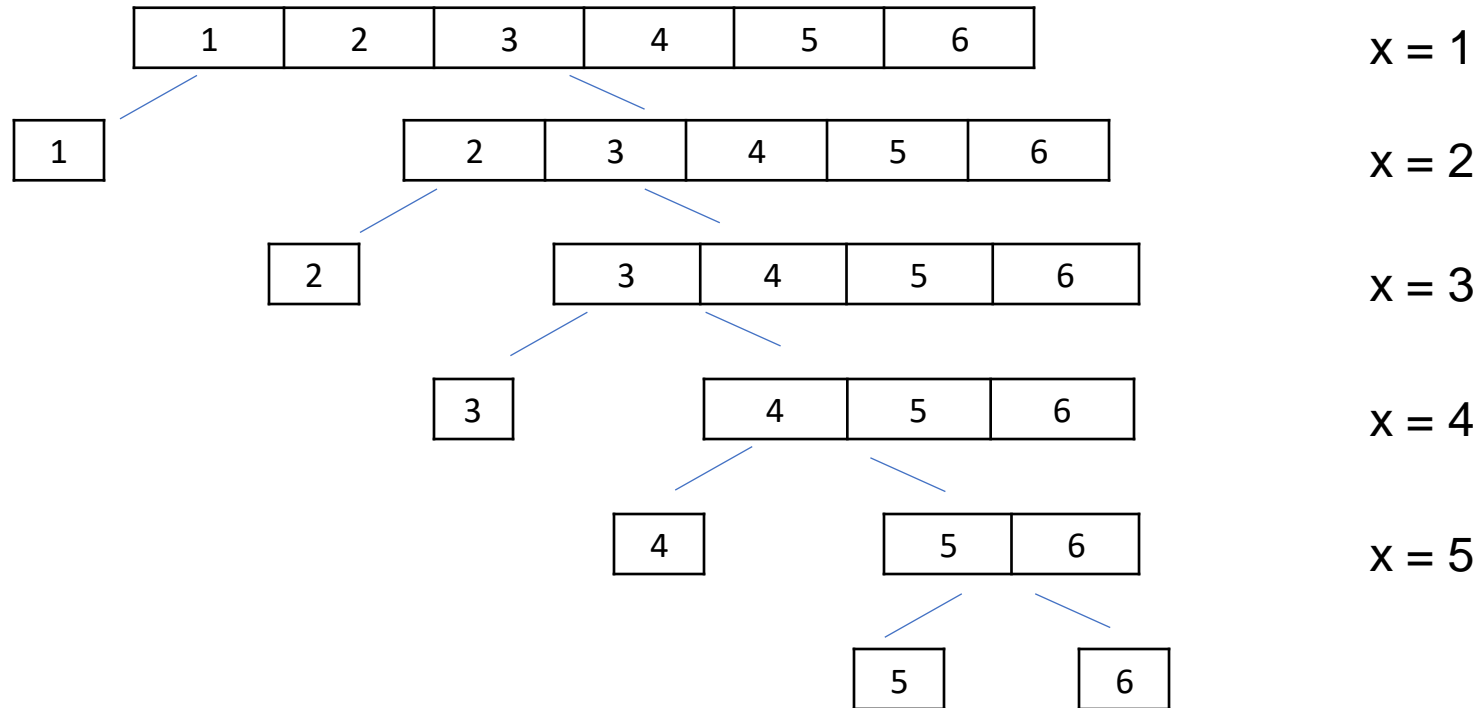
La cui soluzione è $T_{\text{worst}}(n) = \Theta(n^2)$

Si può dimostrare che questo è il **caso peggiore**; quindi per il Quicksort:

$$T(n) = O(n^2)$$

Un esempio del caso peggiore del Quicksort

Un array ordinato



Analisi Quicksort (caso migliore)

Un altro caso: ad ogni passo il pivot scelto è la “**mediana**” degli elementi nell’array (la partizione è $n/2 \mid n/2$):

$$T_{\text{best}}(n) = 2 T_{\text{best}}(n/2) + \Theta(n)$$

La cui soluzione è $T_{\text{best}}(n) = \Theta(n \log n)$

(è la stessa relazione di ricorrenza del Mergesort)

Si può dimostrare che questo è il **caso migliore**; quindi: **$T(n) = \Omega(n \log n)$**

Riassumendo, per il Quicksort: $T(n) = O(n^2)$ e $T(n) = \Omega(n \log n)$

Il caso migliore è diverso dal caso peggiore quindi

$T(n)$ **non** è Θ di nessuna funzione

Is Quicksort ... quick?

Il **Quick**sort non ha un «buon» caso peggiore. A dispetto di ciò, è spesso la migliore scelta nella pratica, perché è molto efficiente nel **caso medio**: il suo tempo è $\Theta(n \log n)$ e le costanti nascoste sono piccole.

Per questo si può considerare una sua versione «randomizzata»

Algoritmo randomizzato:

- Introduce una chiamata a **random(a, b)** (che restituisce un numero a caso fra a e b ($a < b$))
- Forza l'algoritmo a comportarsi come nel caso medio
- Non esiste una distribuzione d'input «peggiore» a priori

Nota: sul libro di testo trovate solo una versione randomizzata.

Per il resto potete fare riferimento al libro di Cormen, Leiserson, Rivest, (Stein)

Introduzione agli algoritmi, o ad altri testi consigliati nel programma.

QuickSort randomizzato

```
Random-Partition (A, p, r)
```

```
  i ← random(p,r)
```

```
  scambia A[i] <-> A[p]
```

```
  return Partition(A, p, r)
```

```
Random-Quicksort (A, p, r)
```

```
  if p < r then
```

```
    q ← Random-Partition (A,p,r)
```

```
    Random-Quicksort(A, p, q)
```

```
    Random-Quicksort(A, q+1, r)
```

Quicksort vs Mergesort

(entrambi Divide et Impera)

Fase		MergeSort	Tempi	QuickSort	Tempi
I	Divide	$q = \lfloor (p + r)/2 \rfloor$	$\Theta(1)$	PARTITION	$\Theta(n)$
II	Ricorsione	$\lfloor n/2 \rfloor \mid \lceil n/2 \rceil$	$2T(n/2)$	$k \mid n - k$	$T(k) + T(n-k)$
III	Combina	MERGE	$\Theta(n)$	niente	$\Theta(1)$
			$T(n) = 2T(n/2) + \Theta(n)$		$T(n) = T(k) + T(n - k) + \Theta(n)$
			$T(n) = \Theta(n \log n)$		$T(n) = O(n^2), T(n) = \Omega(n \log n)$

Da ricordare sulla complessità dell'ordinamento

Esistono algoritmi di ordinamento con tempo nel caso peggiore $\Theta(n^2)$ e $\Theta(n \log n)$

Esistono anche algoritmi di ordinamento con tempo nel caso peggiore $\Theta(n)$, ma ... **non** sono basati sui confronti e funzionano **solo** sotto certe ipotesi.

Inoltre si può dimostrare che **tutti** gli algoritmi di ordinamento basati sui confronti richiedono **$\Omega(n \log n)$** confronti nel caso peggiore!

Si dice che $\Omega(n \log n)$ è una delimitazione inferiore (*lower bound*) al problema dell'ordinamento, cioè al numero di confronti richiesti per ordinare n oggetti.

Delimitazione inferiore (*lower bound*) =
quantità di risorsa **necessaria** per risolvere un determinato problema.
Indica la difficoltà intrinseca del problema.

QuickSort ungherese

Altre relazioni di ricorrenza

Consideriamo la sotto-famiglia

- $T(n) = qT(n/2) + cn$ con $T(2)=c$

per $q=1$ allora $T(n) = \Theta(n)$

$q=2$ allora $T(n) = \Theta(n \log_2 n)$ (MergeSort)

$q>2$ allora $T(n) = ?$

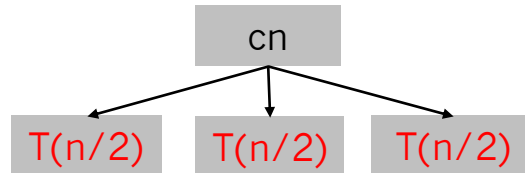
- $T(n) = 2T(n/2) + cn^2$

$T(n) = ?$

Albero di ricorsione per il caso $q=3$

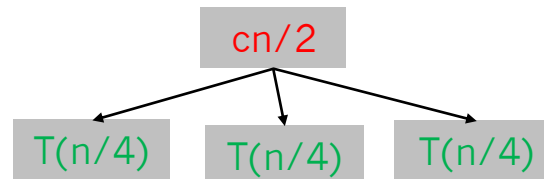
$$T(n) = 3 T(n/2) + cn$$
$$T(2) = c$$

Albero per $T(n)$:

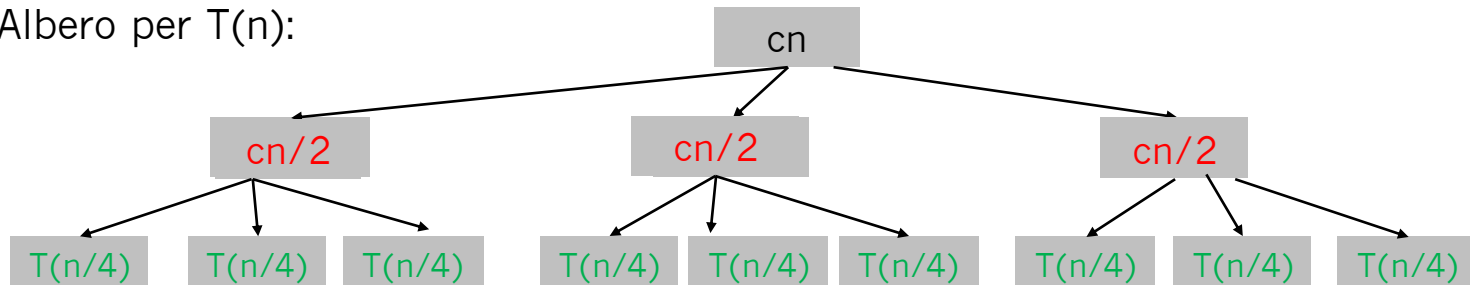


Albero per $T(n/2)$:

$$T(n/2) = 3 T(n/4) + cn/2$$



Albero per $T(n)$:



Albero di ricorsione per il caso $q=3$

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

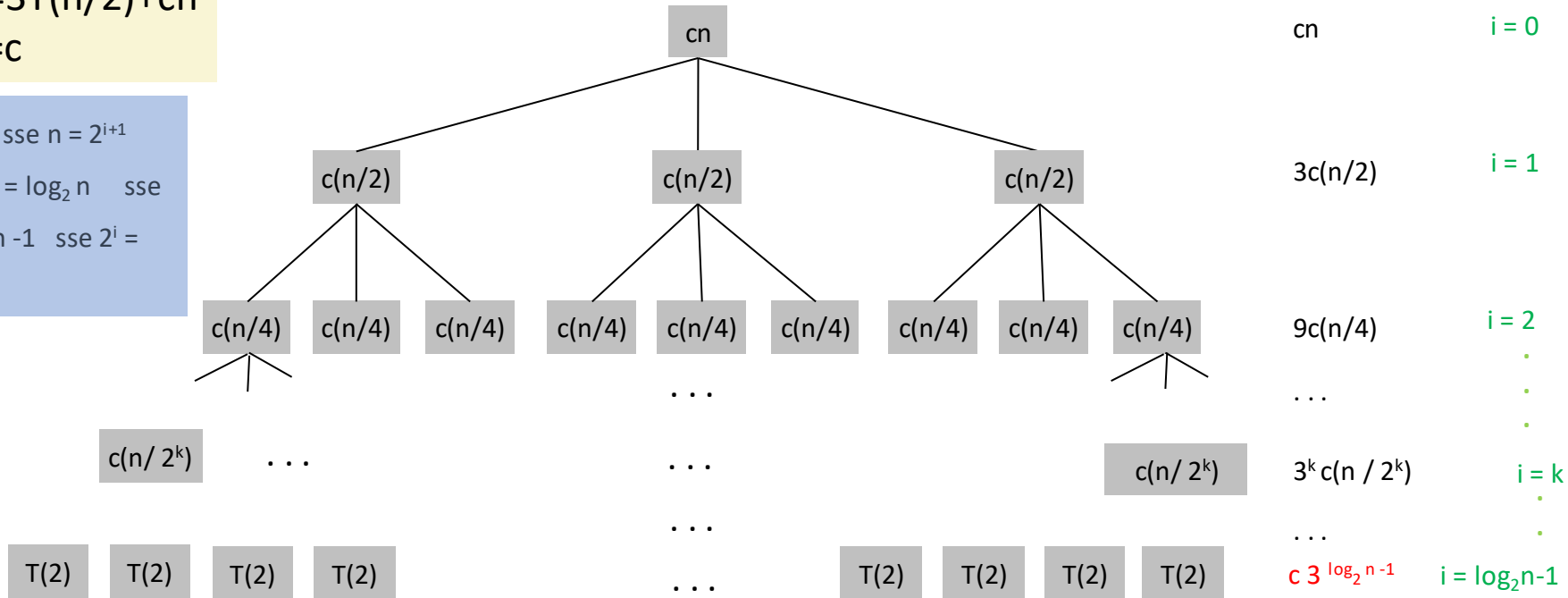
$$T(2) = c$$

$$n/2^i = 2 \text{ sse } n = 2^{i+1}$$

$$\text{sse } i+1 = \log_2 n \text{ sse}$$

$$i = \log_2 n - 1 \text{ sse } 2^i =$$

$$n/2$$



$$T(n) = c (3^{\log_2 n - 1}) + cn (1 + 3/2 + (3/2)^2 + \dots + (3/2)^{\log_2 n - 2})$$

Da ricordare: serie geometrica



Somme finite: Se $\alpha \neq 1$ allora

$$\sum_{i=0}^n \alpha^i = \frac{\alpha^{n+1} - 1}{\alpha - 1} \quad (1)$$

Somme infinite: Se $0 < \alpha < 1$. Allora

$$\sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha} \quad (2)$$

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

$$T(2) = c$$

$$T(n) = c (3^{\log_2 n - 1}) + cn (1 + 3/2 + (3/2)^2 + \dots + (3/2)^{\log_2 n - 2})$$

$$\text{Poiché } c (3^{\log_2 n - 1}) \leq cn (3/2)^{\log_2 n - 1} \text{ (verifica per esercizio)}$$

$$T(n) \leq cn \sum_{i=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^i$$

$$\text{Pongo } r = \frac{3}{2}$$

$$T(n) \leq cn \left(\frac{r^{\log_2 n} - 1}{r - 1} \right) \leq cn \left(\frac{r^{\log_2 n}}{r - 1} \right) = \left(\frac{c}{r - 1} \right) nr^{\log_2 n}$$

$$r^{\log_2 n} = n^{\log_2 r} = n^{\log_2 \frac{3}{2}} = n^{(\log_2 3 - 1)}$$

$$T(n) \leq \left(\frac{c}{r - 1} \right) n n^{(\log_2 3 - 1)} = \left(\frac{c}{r - 1} \right) n^{\log_2 3}$$

$$T(n) = O(n^{\log_2 3})$$

Albero di ricorsione per il caso $q > 2$

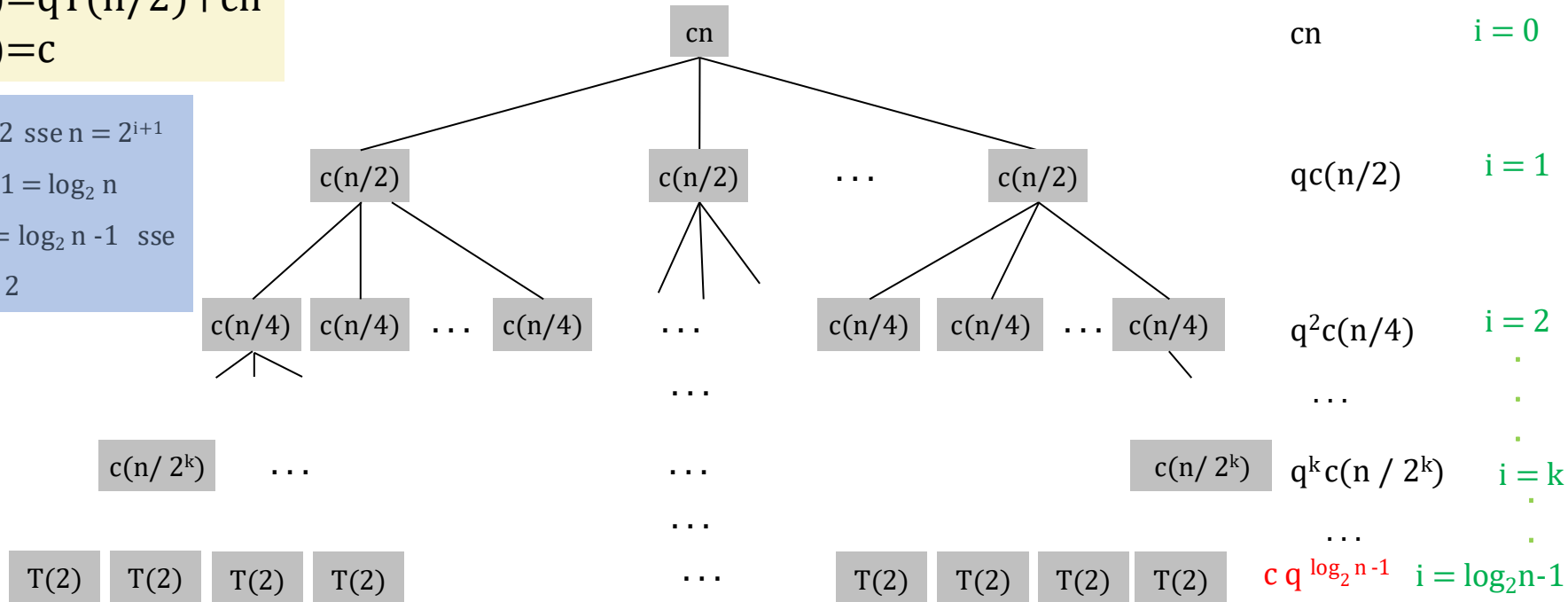
$$T(n) = qT(n/2) + cn$$

$$T(2) = c$$

$$n/2^i = 2 \text{ sse } n = 2^{i+1}$$

$$\text{sse } i+1 = \log_2 n$$

$$\text{sse } i = \log_2 n - 1 \text{ sse } 2^i = n/2$$



$$T(n) = c (q^{\log_2 n - 1}) + cn (1 + q/2 + (q/2)^2 + \dots + (q/2)^{\log_2 n - 2})$$

$$T(n) = qT(n/2) + cn$$

$$T(2) = c$$

$$T(n) = c(q^{\log_2 n - 1}) + cn(1 + q/2 + (q/2)^2 + \dots + (q/2)^{\log_2 n - 2})$$

Poiché $c(q^{\log_2 n - 1}) \leq cn(q/2)^{\log_2 n - 1}$ (*verifica per esercizio*)

$$T(n) \leq cn \sum_{i=0}^{\log_2 n - 1} \left(\frac{q}{2}\right)^i$$

$$\text{Pongo } r = \frac{q}{2}$$

$$T(n) \leq cn \left(\frac{r^{\log_2 n - 1}}{r - 1} \right) \leq cn \left(\frac{r^{\log_2 n}}{r - 1} \right) = \left(\frac{c}{r - 1} \right) nr^{\log_2 n}$$

$$r^{\log_2 n} = n^{\log_2 r} = n^{\log_2 \frac{q}{2}} = n^{(\log_2 q - 1)}$$

$$T(n) \leq \left(\frac{c}{r - 1} \right) n n^{(\log_2 q - 1)} = \left(\frac{c}{r - 1} \right) n^{\log_2 q}$$

$$T(n) = O(n^{\log_2 q})$$

Altre relazioni di ricorrenza

Consideriamo la sotto-famiglia

- $T(n) = qT(n/2) + cn$ con $T(2)=c$

per $q=1$ allora $T(n) = \Theta(n)$

$q=2$ allora $T(n) = \Theta(n \log_2 n)$

$q>2$ allora $T(n) = O(n^{\log_2 q})$

- $T(n) = 2T(n/2) + cn^2$

$T(n) = ?$

Esempi

```
procedure daffy( $n$ )  
  if  $n = 1$  or  $n = 2$  then do qualcosa  
  else  
    daffy( $n - 1$ );  
    for  $i = 1$  to  $n$  do  
      qualcosa di nuovo  
    daffy( $n - 1$ )
```

Esempi

```
procedure elmer( $n$ )  
  if  $n = 1$  then do qualcosa  
  else if  $n = 2$  then do qualcos'altro  
  else  
    for  $i = 1$  to  $n$  do  
      elmer( $n - 1$ )  
    fa qualcosa di differente
```

Altri esempi

Sia $T(1) = 1$. Valutate

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

- $T(n) = T(9n/10) + n$

- $T(n) = 16T(n/4) + n^2$

- $T(n) = 7T(n/3) + n^2$

- $T(n) = 7T(n/2) + n^2$

- $T(n) = 2T(n/3) + \sqrt{n}$

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

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

Esercizio: ricerca ternaria

- **Progettare** un algoritmo per la ricerca di un elemento **key** in un array **ordinato** $A[1..n]$, basato sulla tecnica Divide-et-impera che nella prima fase divide l'array **in 3 parti** «uguali» (le 3 parti differiranno di al più 1 elemento).
- Scrivere la **relazione di ricorrenza** per il tempo di esecuzione dell'algoritmo proposto.
- *Risolvere la relazione di ricorrenza.*
- *Confrontare il tempo di esecuzione con quello della ricerca binaria.*

Alcuni esercizi sulla tecnica del Divide et Impera.

1. a) Descrivere gli aspetti essenziali della tecnica Divide et Impera, utilizzando lo spazio designato.
b) Descrivere ed analizzare un algoritmo **basato sulla tecnica Divide et Impera** che dato un array $A[1, \dots, n]$ di interi ne restituisca il massimo.
2. Sia $V[1..n]$ un vettore ordinato di 0 e 1.
Descrivere ed analizzare un algoritmo per determinare il numero di 0 presenti in $V[1..n]$ in tempo $O(\log n)$.
5. a) Fornire lo pseudocodice di un algoritmo ricorsivo che ordini un array $A[1..n]$ nel seguente modo: prima ordina ricorsivamente $A[1..n-1]$ e poi inserisce $A[n]$ nell'array ordinato $A[1..n-1]$.
b) Analizzare la complessita' di tempo dell'algoritmo proposto al punto b).

6. Sia dato un vettore binario ordinato $A[1..n]$.
 - (a) Progettare un algoritmo di complessita' $\Theta(n)$ nel caso peggiore, che conti il numero di occorrenze di 1 nel vettore A .
 - (b) Progettare un algoritmo di complessita' $O(\log n)$, che conti il numero di occorrenze di 1 nel vettore A .
7. Sia dato un vettore ordinato $A[1..n]$ di interi distinti. Progettare un algoritmo che determini, in tempo $O(\log n)$, se esiste o meno un intero i tale che $A[i] = i$.
8. Descrivere ed analizzare un algoritmo basato sul paradigma *divide et impera* che dato un vettore *ordinato* $A[1..n]$ di interi strettamente positivi (cioe' per ogni $1 \leq i \leq n$, $A[i] \geq 1$), restituisca il numero di occorrenze di 1 nel vettore A . L'algoritmo deve avere complessita' di tempo $O(\log n)$.