Programmazione e Strutture Dati (PR&SD)
I° ANNO – Informatica
Prof. V. Fuccella

Algoritmi Efficienti di Ordinamento

Merge Sort
Quicksort

Il Problema dell'Ordinamento

- Elencare gli elementi di un insieme secondo una sequenza stabilita da una *relazione d'ordine*. Esempi:
  1. Ordinare una breve sequenza di numeri
  2. Mettere un elenco di nomi in ordine alfabetico
  3. Ordinare i record degli studenti Unisa secondo la data di nascita
- Nel caso 3, dobbiamo ordinare dei *record* in base ad una *chiave*
  - La chiave può essere un singolo campo o la combinazione di più campi

Algoritmi di Ordinamento
Proprietà

- **Stabile**: due elementi con la medesima chiave mantengono lo stesso ordine con cui si presentavano prima dell'ordinamento.
- **In loco**: in ogni dato istante al più è allocato un numero costante di variabili, oltre all'array da ordinare
- **Adattivo**: Il numero di operazioni effettuate dipende dall'input
- **Interno vs esterno**:
  - Interno: i dati sono contenuti nella memoria RAM.
  - Esterno: I dati sono residenti su disco o su nastro

Algoritmi Semplici e Avanzati

- Tutti gli algoritmi elencati in basso ordinano **per confronti**
- Algoritmi semplici. Numero di operazioni quadratico rispetto alla taglia dell'input: $O(n^2)$
  - *selection sort*
  - *insertion sort*
  - *bubble sort*
- Algoritmi avanzati. Più efficienti.
  - *Merge sort* (von Neumann, 1945)
    - Numero di operazioni rispetto alla taglia dell'input: O(n log n)
  - *Quicksort* (Hoare, 1961)
    - O(n log n) nel caso medio
    - quadratico nel caso peggiore

Hoare, C. A. R. (1961): Partition: Algorithm 63, Quicksort: Algorithm 64, and Find: Algorithm 65., Comm. ACM 4, pp. 321–322

Algoritmi di Ordinamento

Nome	Migliore	Medio	Peggiore	Memoria	Stabile	In Loco
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Sì
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sì	Sì
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sì	Sì
Quicksort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(n)$*	No	Sì
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	Sì	No

*Spazio aggiuntivo dovuto alla gestione della ricorsione

Divide et Impera

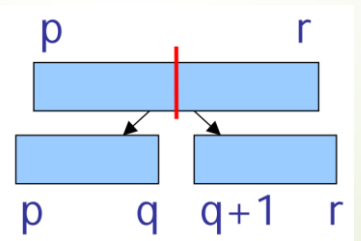
- Approccio per la risoluzione di problemi computazionali
  - **Divide:** si procede alla suddivisione dei problemi in problemi di dimensione minore;
  - **Impera:** i problemi vengono risolti in modo ricorsivo. Quando i sottoproblemi arrivano ad avere una dimensione sufficientemente piccola, essi vengono risolti direttamente tramite il caso base;
  - **Combina:** si ricombina l'output ottenuto dalle precedenti chiamate ricorsive al fine di ottenere il risultato finale.

Mergesort

- Inventato da von Neumann nel 1945
- Esempio del paradigma algoritmico del divide et impera
- E' facile implementare una versione stabile
- Richiede spazio ausiliario (O(N))
- E' implementato come algoritmo standard nelle librerie di alcuni linguaggi (Perl, Java)

Mergesort
Progettazione

- Ricorsivo, "divide et impera"
- Divide:
  - due sottovettori SX e DX rispetto al centro del vettore.



Mergesort
Progettazione

- Impera
  - merge sort su sottovettore SX
  - merge sort su sottovettore DX
  - condizione di terminazione: con 1 (p=r) o 0 (p>r) elementi è ordinato
- Combina
  - Usa ***merge*** per fondere i due sottovettori ordinati in un vettore ordinato.
    - Si estrae ripetutamente il minimo dei due sottovettori e lo si pone nella sequenza in uscita

Merge
Analisi

- **Dati di ingresso:** Array a1 di n1 elementi, a2 di n2 elementi
  - **Precondizione:** ∀ 1<i<n1 a1[i-1] <= a1[i]; ∀ 1<j<n2 a2[j-1] <= a2[j]
- **Dati di uscita:** Array a di n1+n2 elementi
  - **Postcondizione:** ∀ el1 ∈ a1: el1 ∈ a AND ∀ el2 ∈ a2: el2 ∈ a

∀ 1<i<n1+n2: a[i-1] <= a[i]

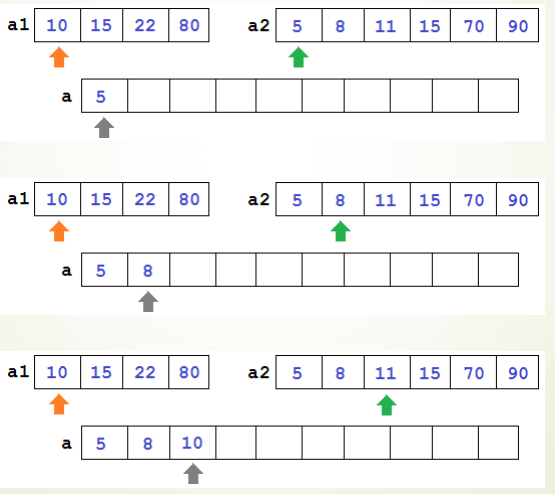
Dizionario dei dati

Identificatore	*Tipo*	*Descrizione*
a1,a2	array	array di interi in input
n1, n2	intero	# di elementi negli array a1, a2
a	array	array di interi in output
I, j	intero	usati per indicizzare i vettori

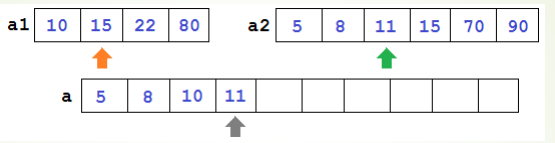
Merge
Progettazione

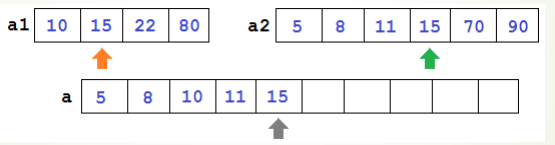
- Scorriamo i due vettori a1 e a2 utilizzando due indici i e j, rispettivamente
- Confrontiamo a1[i] e a2[j] finché i<n1 e j<n2
  - Se a1[i] <= a2[j]: Inseriamo a1[i] in a e incrementiamo i;
  - Altrimenti: inseriamo a2[j] in a e incrementiamo j;
- Riversiamo tutti gli elementi restanti in a1 o in a2 in a

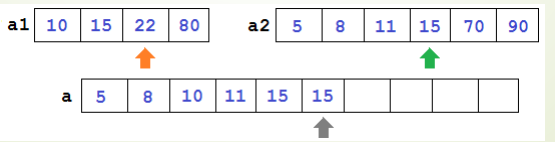
Merge Esempio



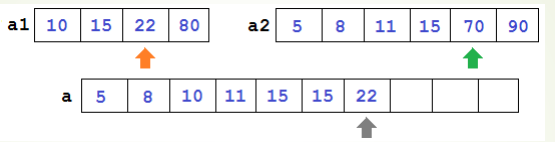
Merge Esempio

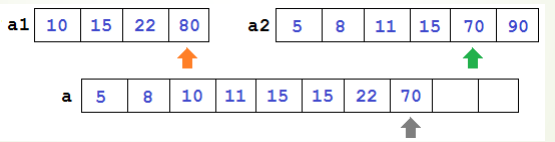


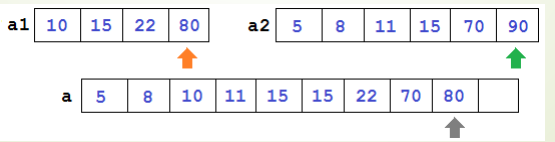




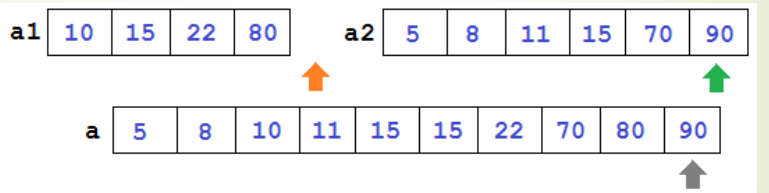
Merge Esempio





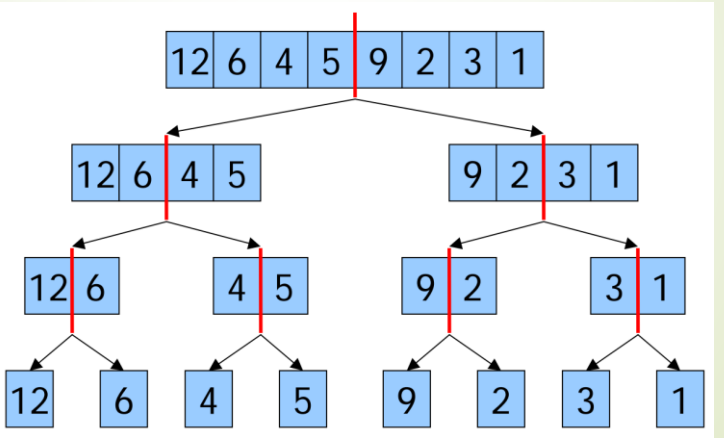


Merge Esempio

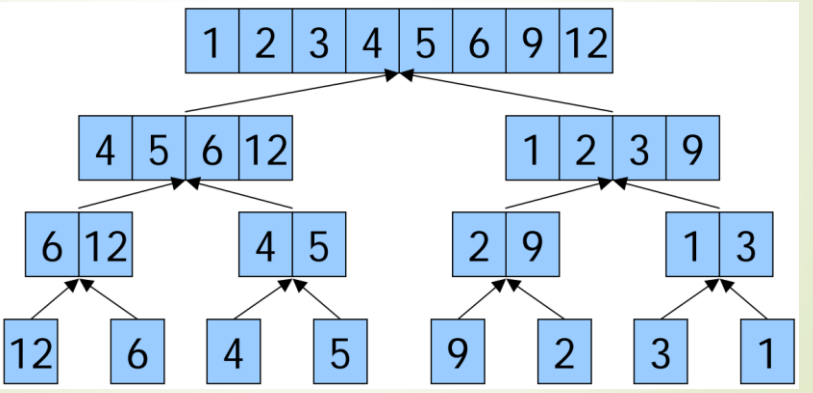


Esempio

Divisione Ricorsiva

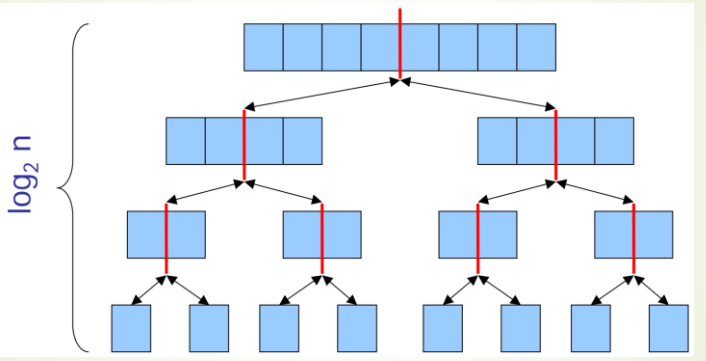


Esempio
Combinazione



Mergesort: costo

- Intuitivamente:



Livelli di ricorsione: $\log_2 n$

Operazioni per livello: n

Operazioni totali: $n \log_2 n$

Mergesort

costo

- Equazione alle ricorrenze:

  $T(n) = 2T(n/2) + \Theta(n) \qquad n \geq 2$

  $T(1) = 1$

- Soluzione:

  $T(n) = \Theta(n \log n)$

Ricorsione e valutazione della complessità

1. Lavoro di combinazione costante
   a) $T(n) = a_1 T(n-1) + a_2 T(n-2) + \ldots a_h T(n-h) + b$ per $n > h$
      - Esponenziale con n: se sono presenti almeno 2 termini (l'algoritmo contiene almeno 2 chiamate ricorsive)
      - Lineare con n: se è presente un solo termine (singola chiamata ricorsiva)
   b) $T(n) = a\, T(n/p) + b$ per $n > 1$
      - $\log n$ se $a = 1$ (singola chiamata ricorsiva)
      - $n^{\log_p a}$ se $a > 1$ (più chiamate ricorsive)
2. Lavoro di combinazione lineare
   a) $T(n) = T(n-h) + b\, n + d$ per $n > h$
      Quadratico con n
   b) $T(n) = a\, T(n/p) + b\, n + d$
      - Lineare con n se $a < p$
      - $n \log n$ se $a = p$
      - $n^{\log_p a}$ se $a > p$

Programmazione e Strutture Dati (PR&SD)
I° ANNO – Informatica
Prof. V. Fuccella

Algoritmi Efficienti di Ordinamento

Merge Sort
Quicksort

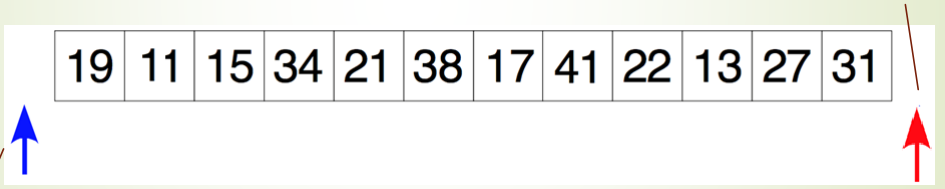
Quicksort

- Ricorsivo, "divide et impera"
- In loco
- Non stabile
- Idea di base per ordinare un array: Partition
  - scegli un elemento (detto pivot)
  - metti a sinistra gli elementi ≤ pivot
  - metti a destra gli elementi ≥ pivot

Partition: idea di base

- Esempio. pivot=17

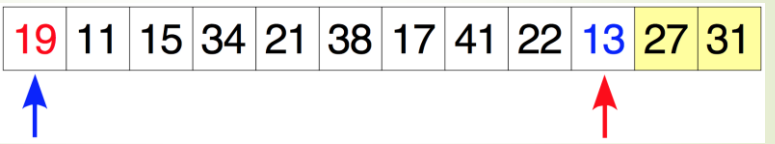
Il puntatore rosso arretra finché non trova un elemento <= pivot



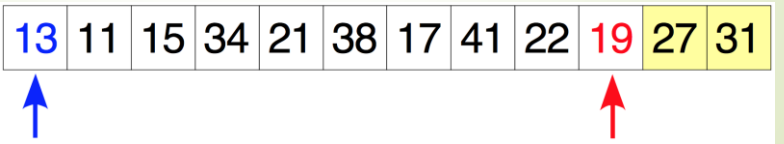
Il puntatore blu avanza finché non trova un elemento >= pivot

Partition: idea di base

- Spostiamo i puntatori, facendo crescere le due regioni, fino a quando è possibile.

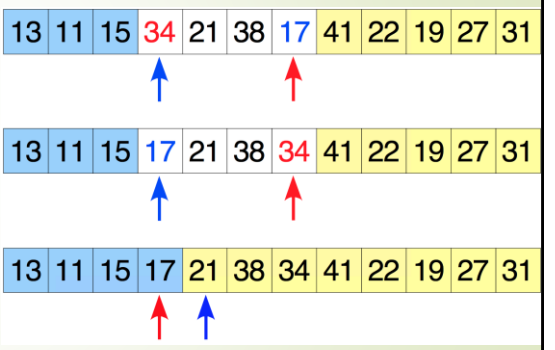


- Quando entrambi i puntatori non possono avanzare scambiamo gli elementi



Partition: idea di base

- A questo punto i due puntatori possono riprendere ad avanzare.
- La procedura termina quando i due puntatori si incrociano
  - Per evitare che il puntatore blu esca fuori dal range, si inserisce un controllo sulla posizione del puntatore rosso (che viene restituito)



Partition

- Pivot x = A[p]
- while(1)
  - Individua A[i] e A[j] elementi "fuori posto"
    - Decrementa **j** fino a trovare un elemento minore o uguale del pivot x
    - Incrementa **i** fino a trovare un elemento maggiore o uguale del pivot x
  - If (i>=j) return j
  - Scambia A[i] e A[j]

Quicksort

- Divide: partiziona il vettore A[p..r] in due sottovettori SX e DX rispetto ad un pivot x
  Il pivot si troverà in posizione q.
- Impera:
  - quicksort su sottovettore SX A[p..q]
  - quicksort su sottovettore DX A[q+1..r]
  - Base della ricorsione: se il vettore ha 1 elemento è ordinato

Quicksort

- Esempio: dobbiamo ordinare

19	21	15	34	21	38	41	17	22	31	27	13

- se prendiamo come pivot 19, otteniamo

19	13	15	17	21	38	41	34	22	31	27	21

Quicksort Quicksort

13	15	17	19	21	21	22	27	31	34	38	41

Analisi

- Trattandosi di un algoritmo di ordinamento basato sui confronti, analizziamo il numero di confronti in funzione del numero *n* di oggetti da ordinare
- tutti i confronti sono eseguiti nella procedura PARTITION.
- quando PARTITION è eseguita su un sottoarray di lunghezza *m* vengono eseguiti *m* confronti (ogni elemento è confrontato una volta con il pivot).

Analisi

- Se l'array da ordinare viene partizionato in due array di dimensione *r* e *n-r* abbiamo che il numero di confronti *T(n)* soddisfa alla ricorrenza:

$$T(n) = \begin{cases} 0 & \text{se } n = 1 \\ T(r) + T(n-r) + n & \text{se } n > 1 \end{cases}$$

- Questa ricorrenza non è del tipo che si può ricondurre ai casi presentati (anche perché il parametro *r* cambia ad ogni chiamata della procedura!)
- Studieremo il caso peggiore e il caso migliore, e cercheremo di farci un'idea del caso medio.

Analisi

- Efficienza legata al bilanciamento delle partizioni
- A ogni passo PARTITION ritorna:
  - caso peggiore un vettore da 1 elemento e l'altro da n-1
  - caso migliore due vettori da n/2 elementi
  - caso medio due vettori di dimensioni diverse.
- Bilanciamento legato alla scelta del pivot.

Caso Peggiore

- Caso peggiore: pivot =minimo o massimo
- Sfortunatamente, con la nostra scelta del pivot il caso peggiore si presenta quando l'array è già ordinato (sia in ordine crescente che decrescente).
- Equazione di ricorrenza:

$$T(n) = T(n-1) + n \qquad n \geq 2$$

$$T(1) = 1$$

$$T(n) = \Theta(n^2)$$

Ricorsione e valutazione della complessità

1. Lavoro di combinazione costante
   a) $T(n) = a_1 T(n-1) + a_2 T(n-2) + \ldots a_h T(n-h) + b$ per $n > h$
      - Esponenziale con n: se sono presenti almeno 2 termini (l'algoritmo contiene almeno 2 chiamate ricorsive)
      - Lineare con n: se è presente un solo termine (singola chiamata ricorsiva)
   b) $T(n) = a\,T(n/p) + b$ per $n > 1$
      - $\log n$ se $a = 1$ (singola chiamata ricorsiva)
      - $n^{\log_p a}$ se $a > 1$ (più chiamate ricorsive)
2. Lavoro di combinazione lineare
   a) $T(n) = T(n-h) + b\,n + d$ per $n > h$
      Quadratico con n
   b) $T(n) = a\,T(n/p) + b\,n + d$
      - Lineare con n se $a < p$
      - $n \log n$ se $a = p$
      - $n^{\log_p a}$ se $a > p$

Caso Migliore

- Il caso migliore si verifica quando ad ogni passo l'array viene partizionato in due regioni uguali. Abbiamo allora:

$$T(n) = 2T(n/2) + n \qquad n \geq 2$$

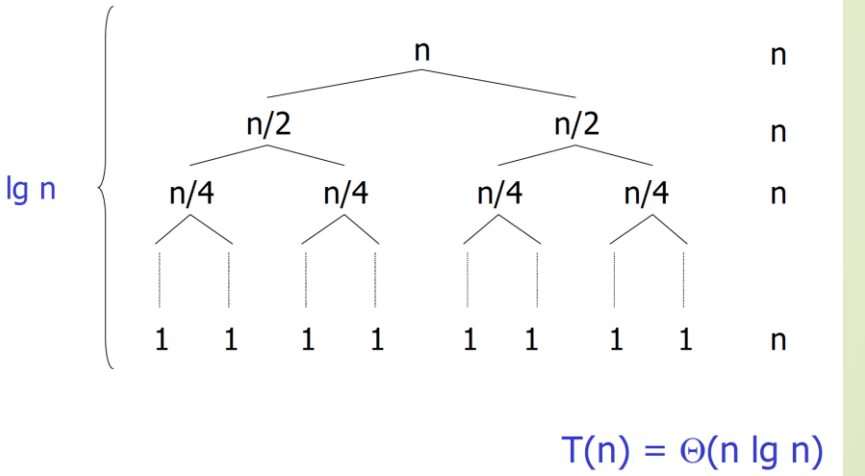
$$T(1) = 1$$

$$T(n) = \Theta(n \log n)$$

Ricorsione e valutazione della complessità

1. Lavoro di combinazione costante
   a) $T(n) = a_1 T(n-1) + a_2 T(n-2) + \dots a_h T(n-h) + b$ per $n > h$
      - Esponenziale con n: se sono presenti almeno 2 termini (l'algoritmo contiene almeno 2 chiamate ricorsive)
      - Lineare con n: se è presente un solo termine (singola chiamata ricorsiva)
   b) $T(n) = a\,T(n/p) + b$ per $n > 1$
      - $\log n$ se $a = 1$ (singola chiamata ricorsiva)
      - $n^{\log_p a}$ se $a > 1$ (più chiamate ricorsive)
2. Lavoro di combinazione lineare
   a) $T(n) = T(n-h) + b\,n + d$ per $n > h$
      Quadratico con n
   b) $T(n) = a\,T(n/p) + b\,n + d$
      - Lineare con n se $a < p$
      - $n \log n$ se $a = p$
      - $n^{\log_p a}$ se $a > p$

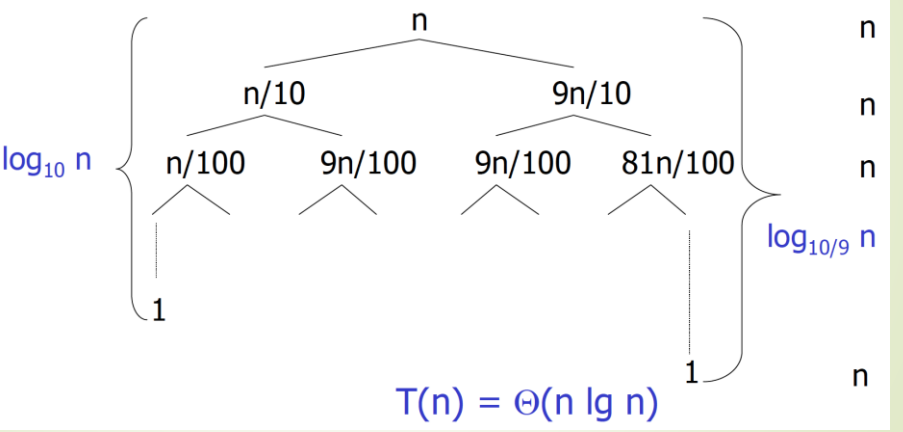
Caso Migliore



Caso Medio

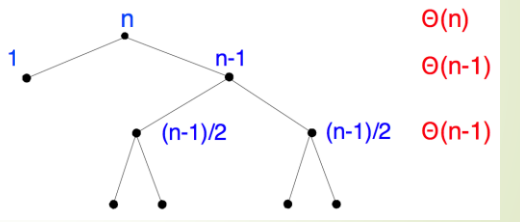
- Ancora una volta il caso peggiore e il caso migliore risultano molto diversi tra loro.
- È naturale chiedersi se dobbiamo aspettarci più di frequente un tempo $\Theta(n^2)$ o un tempo $\Theta(n \log n)$ o qualcosa di intermedio.
- Per farci un'idea sul caso medio studiamo il caso (altamente improbabile) che la procedura PARTITION crei sempre due regioni di cui una sia 9 volte più grande dell'altra.

Caso Medio



Caso Medio

- Studiamo ora il caso in cui si alternano una partizione cattiva e una buona.
- Nei livelli dispari il problema viene spezzato in [1,n-1], nei livelli pari in [n/2,n/2].



- Per scendere di un livello nell'albero il costo è Θ(n).
- L'altezza dell'albero è circa 2log n quindi il costo complessivo è Θ(nlog n).

Riassumendo

- Se le partizioni sono spesso molto sbilanciate, il costo può risultare molto alto.
- Se le partizioni non sono "troppo sbilanciate" si ha un costo $\Theta(n \log n)$ cioè asintoticamente uguale al caso ottimo.

Random pivoting

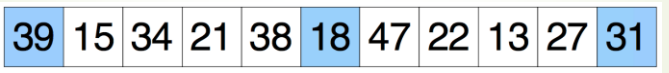
- Elemento a caso: genera un numero casuale i con p ≤i ≤r, poi scambia A[1] e A[i], usando come pivot A[1]
- Si può dimostrare matematicamente che il tempo di esecuzione di Quicksort con pivot casuale è $\Theta(n \log n)$ con probabilità molto vicina ad 1.
- In altre parole, devo essere "molto sfortunato" per avere un tempo di esecuzione asintoticamente superiore a $\Theta(n \log n)$.

Pivot "medio di 3"

- La generazione dei numeri casuali rallenta la procedura PARTITION e quindi tutto l'algoritmo Quicksort.
- Per questo motivo sono state proposte altre strategie di selezione del pivot che in pratica risultano *leggermente più veloci*.

Pivot “medio di 3”

- La strategia “medio di 3” consiste nel considerare gli elementi che sono nella prima e ultima posizione dell'array, e l'elemento che si trova in posizione mediana. Esempio:



- Di questi tre valori viene preso quello intermedio (nell'esempio qui sopra 31).