UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Ingegneria
"Enzo Ferrari"

Algoritmi e Strutture Dati

Algoritmi di ordinamento

Problema dell'ordinamento

Input: Una sequenza A = {a_1, a_2, …, a_n} di n valori distinti

Output: Una sequenza B = {b_1, b_2, …, b_n} permutazione di A tale che b_1,< b_2 < … < b_n

Il numero di permutazioni è n!

Dati v_i e v_j con i < j è sempre possibile partizionare l'insieme delle permutazioni in due sottoinsiemi:

- le permutazioni con $v_i < v_j$ (ammissibile)
- le permutazioni con $v_i > v_j$ (non ammissibile)

Qual è il numero k minimo di partizioni per giungere ad un sottoinsieme di una sola permutazione? $2^k = n!$

$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ formula di Stirling

$2^k = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ da cui $k = n\log\frac{n}{e} + \frac{1}{2}\log(2\pi n)$

ovvero $\boldsymbol{\Omega(n\,log\,n)}$

SELECTION SORT

Approccio naif

Cerco il valore massimo e lo scambio con il valore più a destra, riducendo il problema agli altri n - 1 valori

```
void SelectionSort(int v[], int dim){
int p;
while (dim > 1) {
    p = trovaPosMax(v, dim);
    if (p < dim-1) scambia(&v[p],&v[dim-1]);
    dim--;}
}

int trovaPosMax(int v[], int n){
int i, p=0; /* ipotesi: max = v[0] */
for (i=1; i<n; i++)
  if (v[p]<v[i]) p=i;
return p;}
```

SELECTION SORT

```
void SelectionSort(int v[], int dim){
int p;
while (dim > 1) {
    p = trovaPosMax(v, dim);
    if (p < dim-1) scambia(&v[p],&v[dim-1]);
    dim--;}
}

int trovaPosMax(int v[], int n){
int i, p=0; /* ipotesi: max = v[0] */
for (i=1; i<n; i++)
  if (v[p]<v[i]) p=i;
return p;}
```

	0	1	2	3	4	5	6
dim=7, p=4	7	4	2	1	8	3	5
dim=6, p=0	7	4	2	1	5	3	8
dim=5, p=4	3	4	2	1	5	7	8
dim=4, p=1	3	4	2	1	5	7	8
dim=3, p=0	3	1	2	4	5	7	8
dim=2, p=0	2	1	3	4	5	7	8
dim=1, p=4	1	2	3	4	5	7	8

SELECTION SORT

```
void SelectionSort(int v[], int dim){
int p;
while (dim > 1) {
    p = trovaPosMax(v, dim);
    if (p < dim-1) scambia(&v[p],&v[dim-1]);
    dim--;}
}

int trovaPosMax(int v[], int n){
int i, p=0; /* ipotesi: max = v[0] */
for (i=1; i<n; i++)
  if (v[p]<v[i]) p=i;
return p;}
```

Complessità nel caso medio, pessimo, ottimo?

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = n^2 - n/2 = O(n^2)$$

BUBBLE SORT

Corregge il difetto principale del selection sort ovvero quello di non accorgersi quando il vettore è già ordinato.

Opera tramite “passate” successive sul vettore.

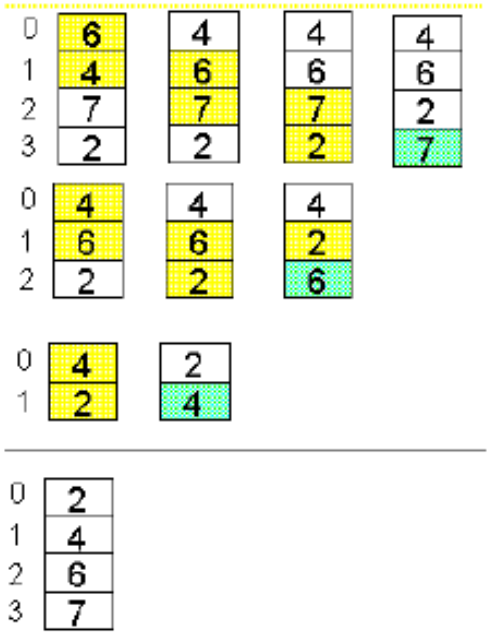
A ogni “passata”, considera una ad una tutte le possibili coppie di elementi adiacenti, scambiandoli se sono nell’ordine errato.

Al termine della passata, quindi, l’elemento massimo sarà in fondo alla parte di vettore considerato.

Se non si verificano scambi, il vettore è già ordinato e quindi l’algoritmo termina.

```
void bubbleSort(int v[], int dim){
int i;
int ordinato = 0;
while (dim>1 && !ordinato) {
    ordinato = 1; /* hp: è ordinato */
    for (i=0; i<dim-1; i++)
        if (v[i]>v[i+1]) {
            scambia(&v[i],&v[i+1]);
            ordinato = 0;}
dim--;}}
```

BUBBLE SORT



La complessità dell'algoritmo dipende dalla configurazione dell'input

Nel caso peggiore l'algoritmo richiede un numero di confronti analogo al precedente. La complessità asintotica è $O(n^2)$

Il caso migliore si verifica quando il vettore è già ordinato. Alla prima passata, con (n-1) confronti, l'algoritmo se ne accorge e non prosegue inutilmente.

Il numero di confronti varia quindi fra $(n^2-n)/2$ e (n-1).

INSERTION SORT

Approccio diverso: per ottenere un vettore ordinato basta *costruirlo ordinato, inserendo ogni elemento al posto giusto*.

Si basa sul principio di ordinamento di una "mano" di carte da gioco (e.g. scala quaranta)

Algoritmo efficiente per ordinare piccoli insiemi di elementi

```
void InsertionSort(int v[], int dim){
for(int i=1; i< dim, i++){
    int temp = v[i];
    int j = i;
    while(j > 0 && v[j-1] > temp){
        v[j] = v[j-1];
        j--;}
    v[j] = temp;
}}
```

INSERTION SORT

	0	1	2	3	4	5	6	temp
	7	4	2	1	8	3	5	
i = 1, j = 1	7	**7**	2	1	8	3	5	4
i = 1, j = 0	**4**	7	2	1	8	3	5	4
i = 2, j = 2	4	7	**7**	1	8	3	5	2
i = 2, j = 1	4	**4**	7	1	8	3	5	2
i = 2, j = 0	**2**	4	7	1	8	3	5	2

INSERTION SORT

	0	1	2	3	4	5	6	temp
i = 3, j = 3	2	4	7	**7**	8	3	5	1
i = 3, j = 2	2	4	**4**	7	8	3	5	1
i = 3, j = 1	2	**2**	4	7	8	3	5	1
i = 3, j = 0	1	2	4	7	8	3	5	1
i = 4, j = 4	1	2	4	7	**8**	3	5	8
i = 5, j = 5	1	2	4	7	8	**8**	5	3

INSERTION SORT

	0	1	2	3	4	5	6	temp
i = 5, j = 4	1	2	4	7	**7**	8	5	3
i = 5, j = 3	1	2	4	**4**	7	8	5	3
i = 5, j = 2	1	2	**3**	4	7	8	5	3
i = 6, j = 6	1	2	3	4	7	8	**8**	5
i = 6, j = 5	1	2	3	4	7	**7**	8	5
i = 6, j = 4	1	2	3	4	**5**	7	8	5

INSERTION SORT

Valutazione complessità

Caso migliore (vettore già ordinato).
Bastano (n-1) confronti

Caso peggiore (vettore ordinato al contrario).

$$\sum_{i=1}^{j=n} i = \frac{n*(n+1)}{2} = O(n^2)$$

Caso medio.

Ad ogni ciclo il nuovo elemento va inserito nella posizione centrale del vettore:

$$\sum_{i=1}^{j=n} \frac{i}{2} = \frac{n*(n+1)}{4} = O(n^2)$$

MERGE SORT

Divide et Impera

Merge Sort è basato sulla tecnica divide-et-impera

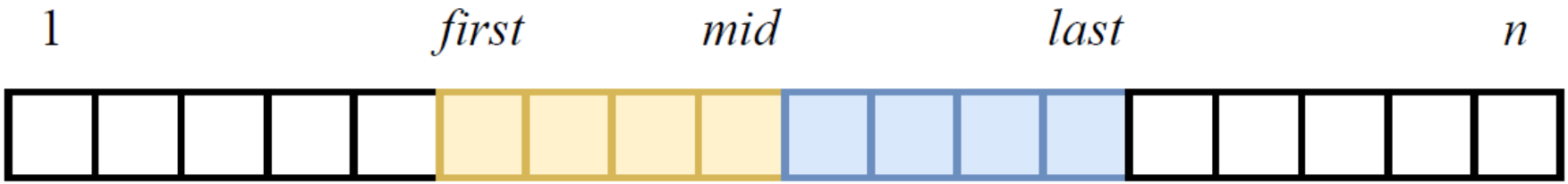
Divide: Spezza virtualmente il vettore di n elementi in due sottovettori di n/2 elementi

Impera: Chiama Merge Sort ricorsivamente sui due sottovettori

Combina: Unisci (merge) le due sequenze ordinate

Idea: Si sfrutta il fatto che i due sottovettori sono già ordinati per ordinare più velocemente

MERGE



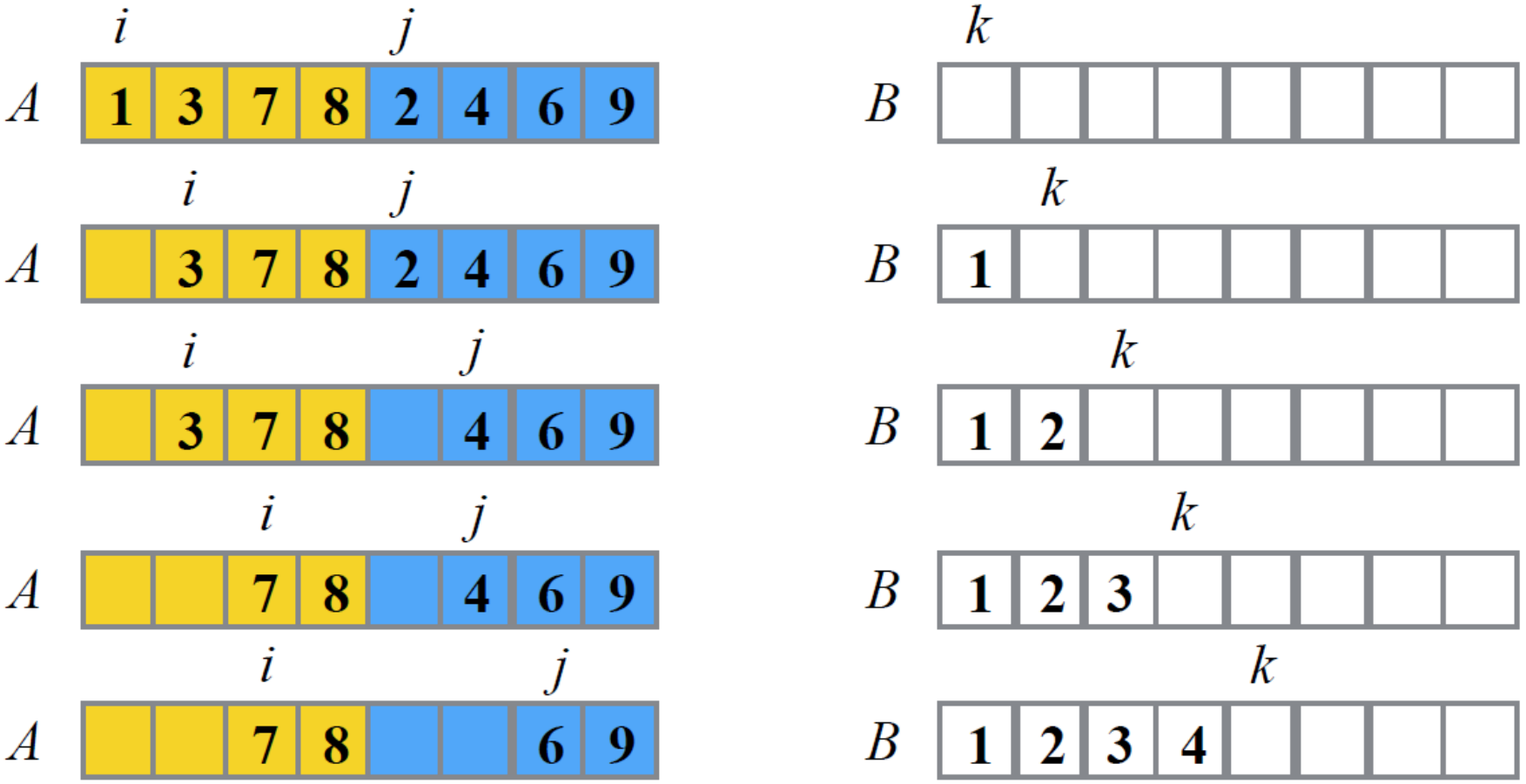
Input:

- A è un vettore di n interi
- $first, last, mid$ sono tali che $1 \leq first \leq mid < last \leq n$
- I sottovettori $A[first \ldots mid]$ e $A[mid + 1 \ldots last]$ sono già ordinati

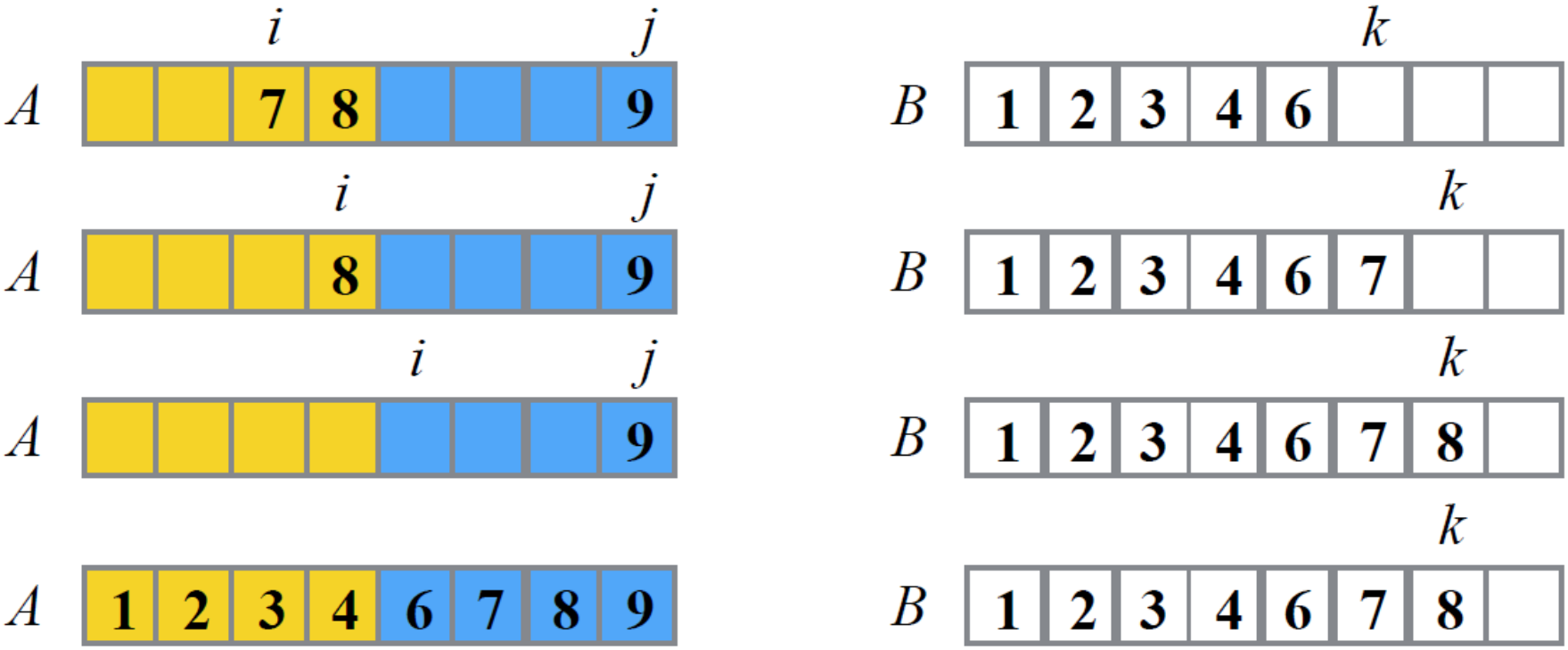
Output:

- I due sottovettori sono fusi in un unico sottovettore ordinato A[$first \ldots last$] tramite un vettore di appoggio B

MERGE



MERGE



MERGE

```
void Merge(int A[], int first, int last, int mid){
int i, j, k, h, B[N];
i = first;
j = mid + 1;
k = first;
while (i <= mid and j <= last) {
     if (A[i] < A[j]) {
          B[k] = A[i];
          i++; }
     else {
          B[k] = A[j];
          j++; }
     k++;
}
while ( i <= mid) { B[k] = A[i++]; k++;}
while ( j <= last) { B[k] = A[j++]; k++;}
for (i=first; i<=last; i++)
     A[i] = B[i];
}
```

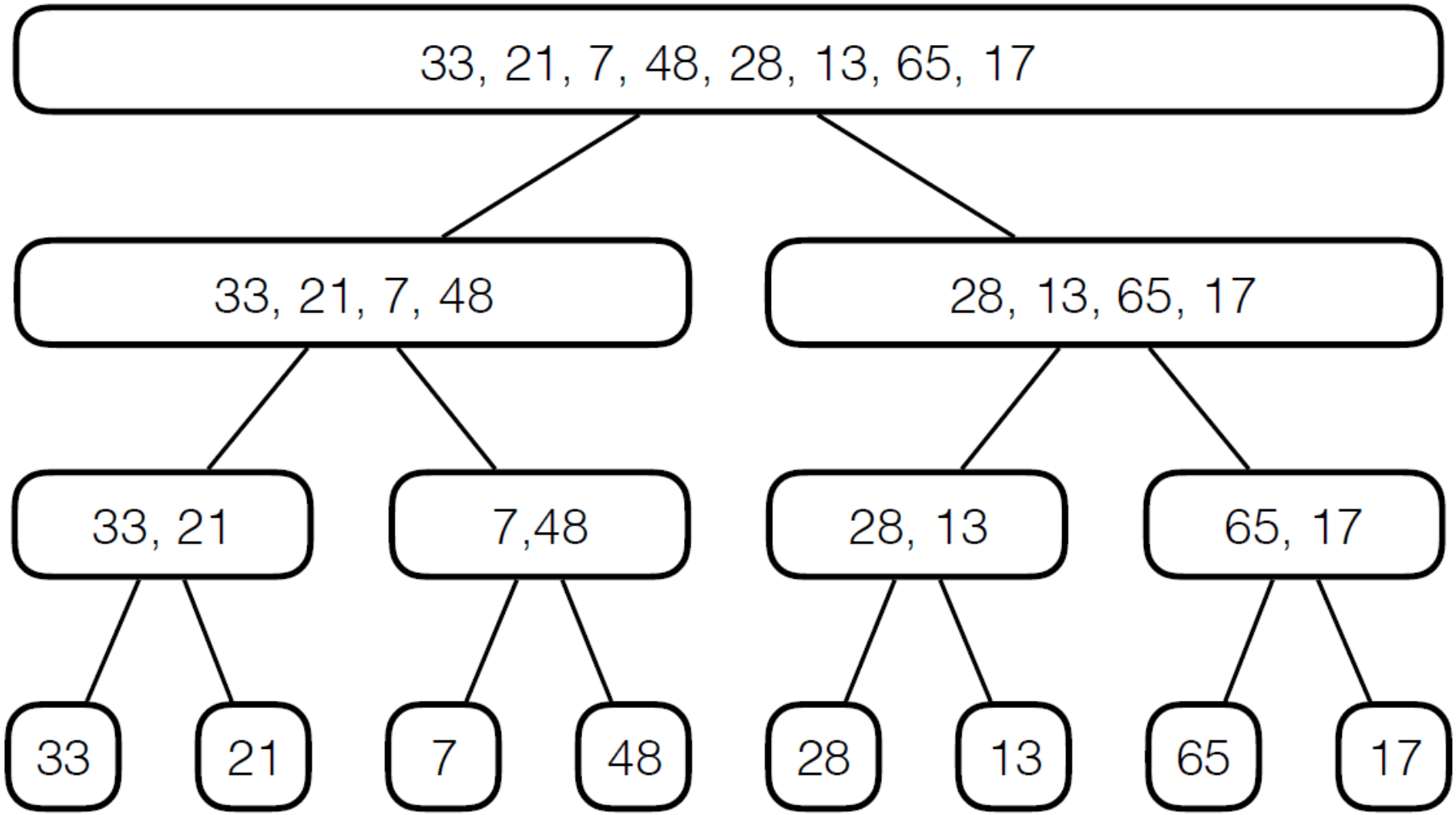
Costo computazionale: $\boldsymbol{O(n)}$

MERGE SORT

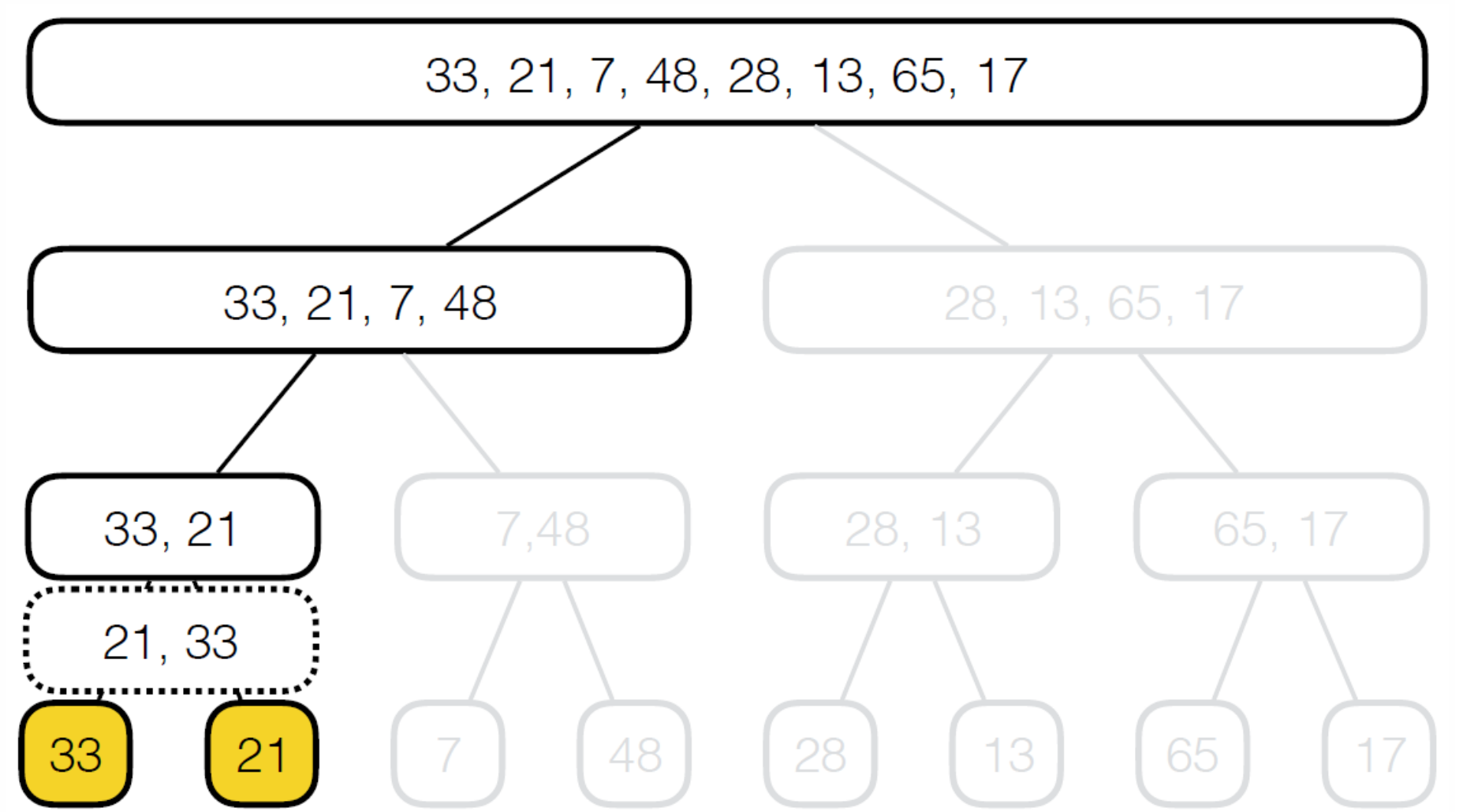
Programma completo:
Chiama ricorsivamente se stesso dividendo la sequenza in due sottosequenze di ugual dimensione e usa Merge() per unire i risultati
Caso base: sequenze di lunghezza 1 sono già ordinate

```
void MergeSort(int A[], int first, int last){
if (first < last) {
     int mid = (first + last)/2;
     MergeSort(A, first, mid);
     MergeSort(A, mid + 1, last);
     Merge(A, first, last, mid);
     }
}
```

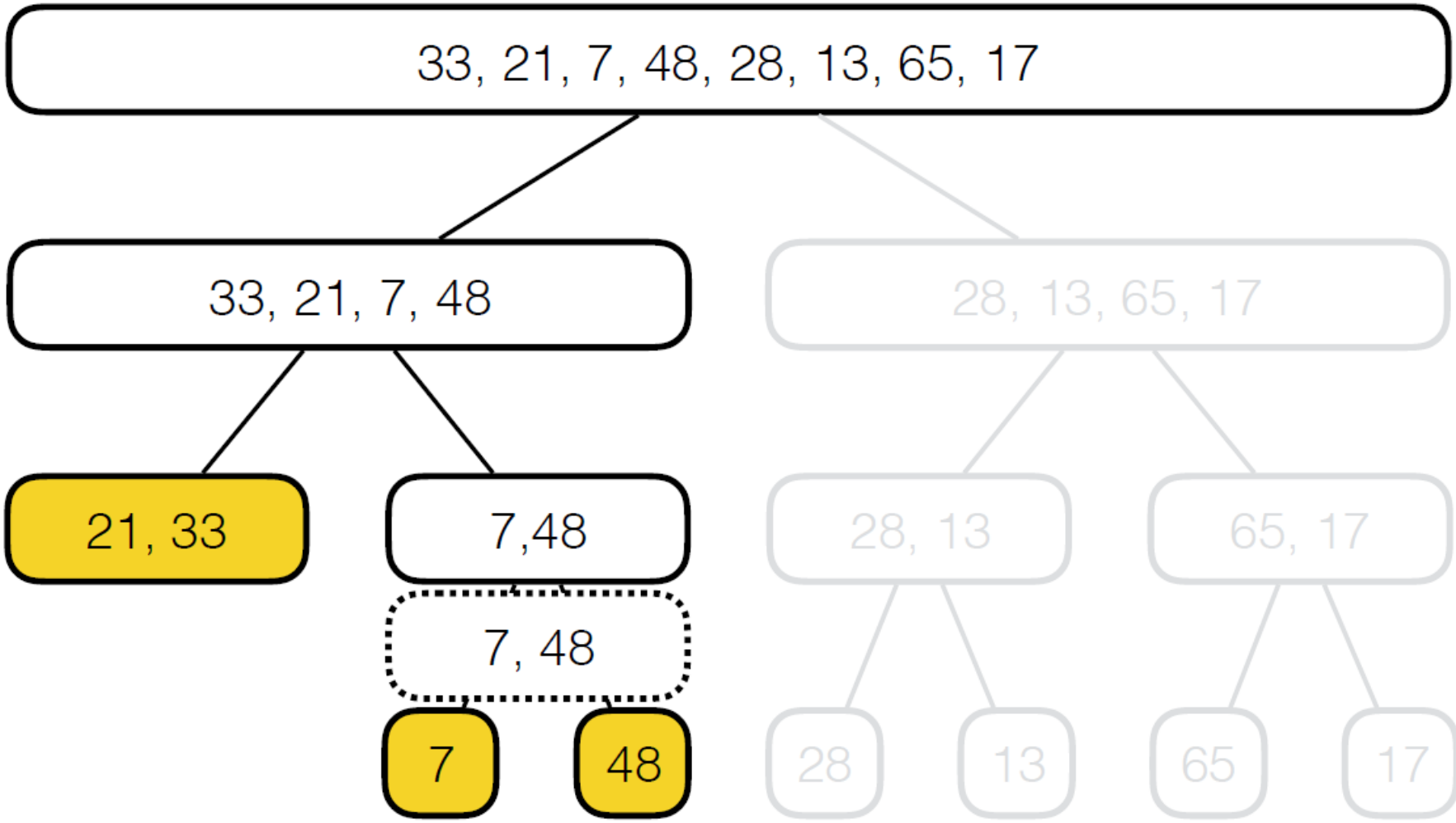
MERGE SORT: esecuzione



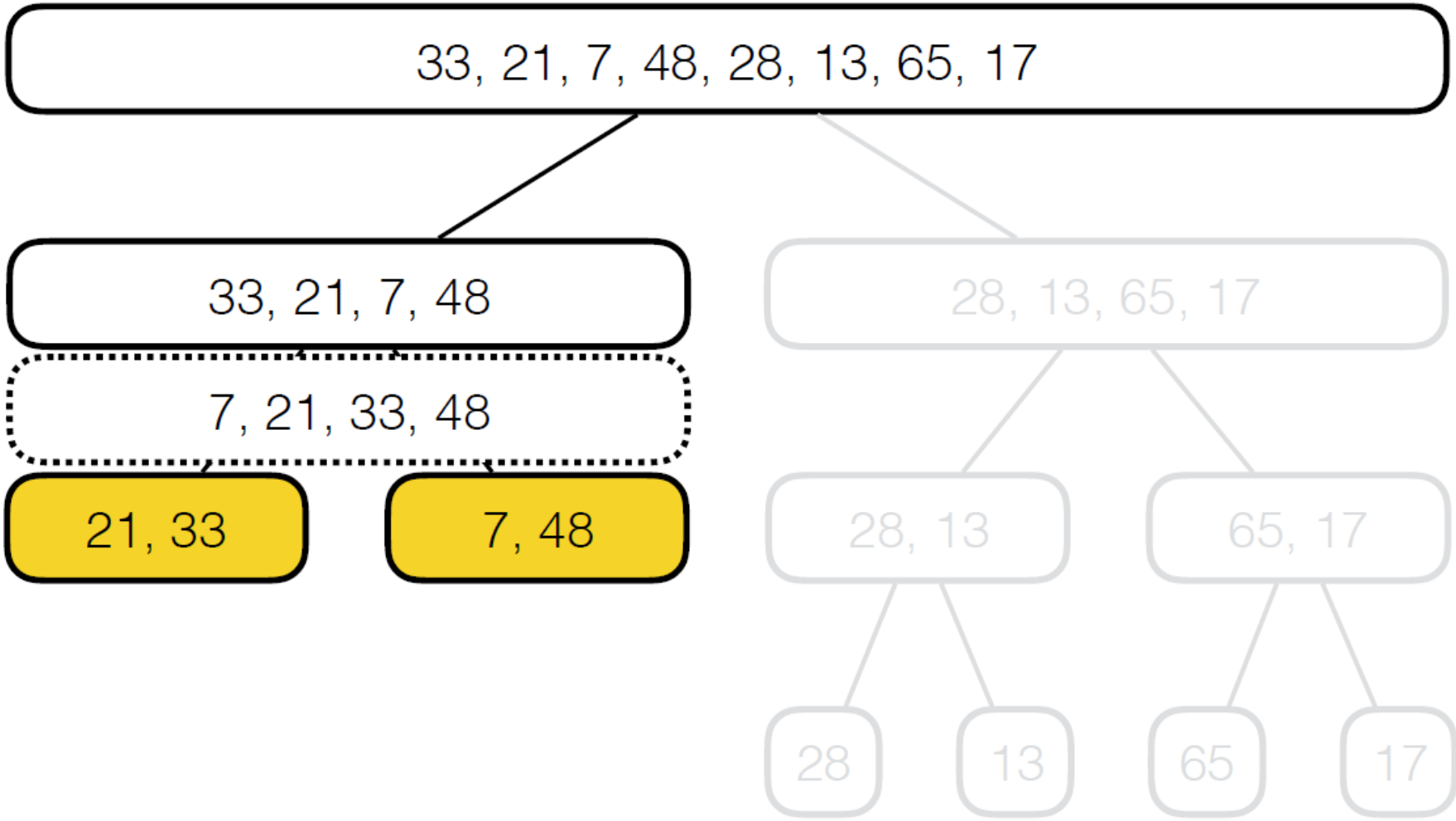
MERGE SORT: esecuzione



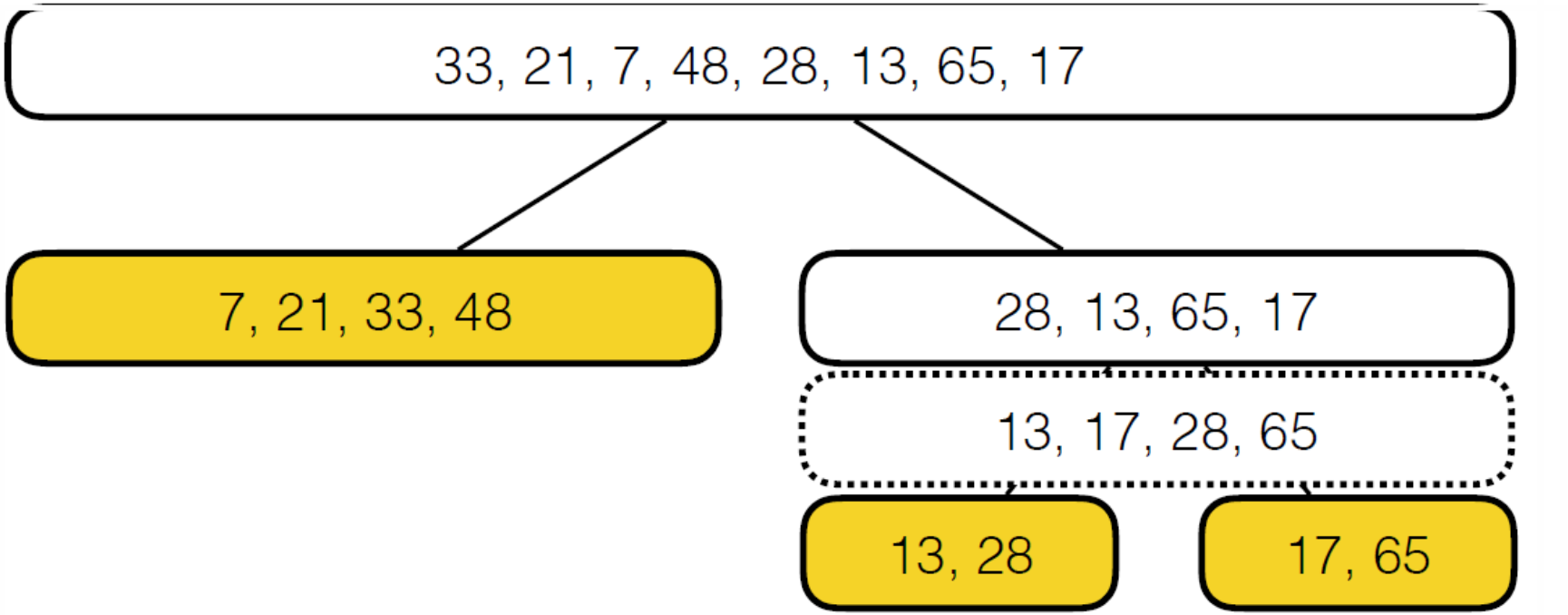
MERGE SORT: esecuzione



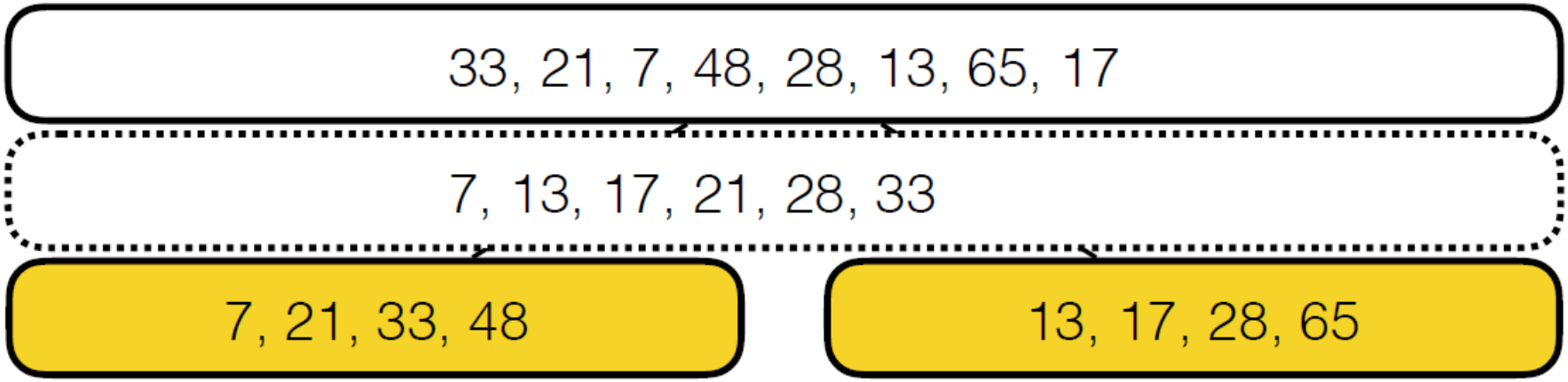
MERGE SORT: esecuzione



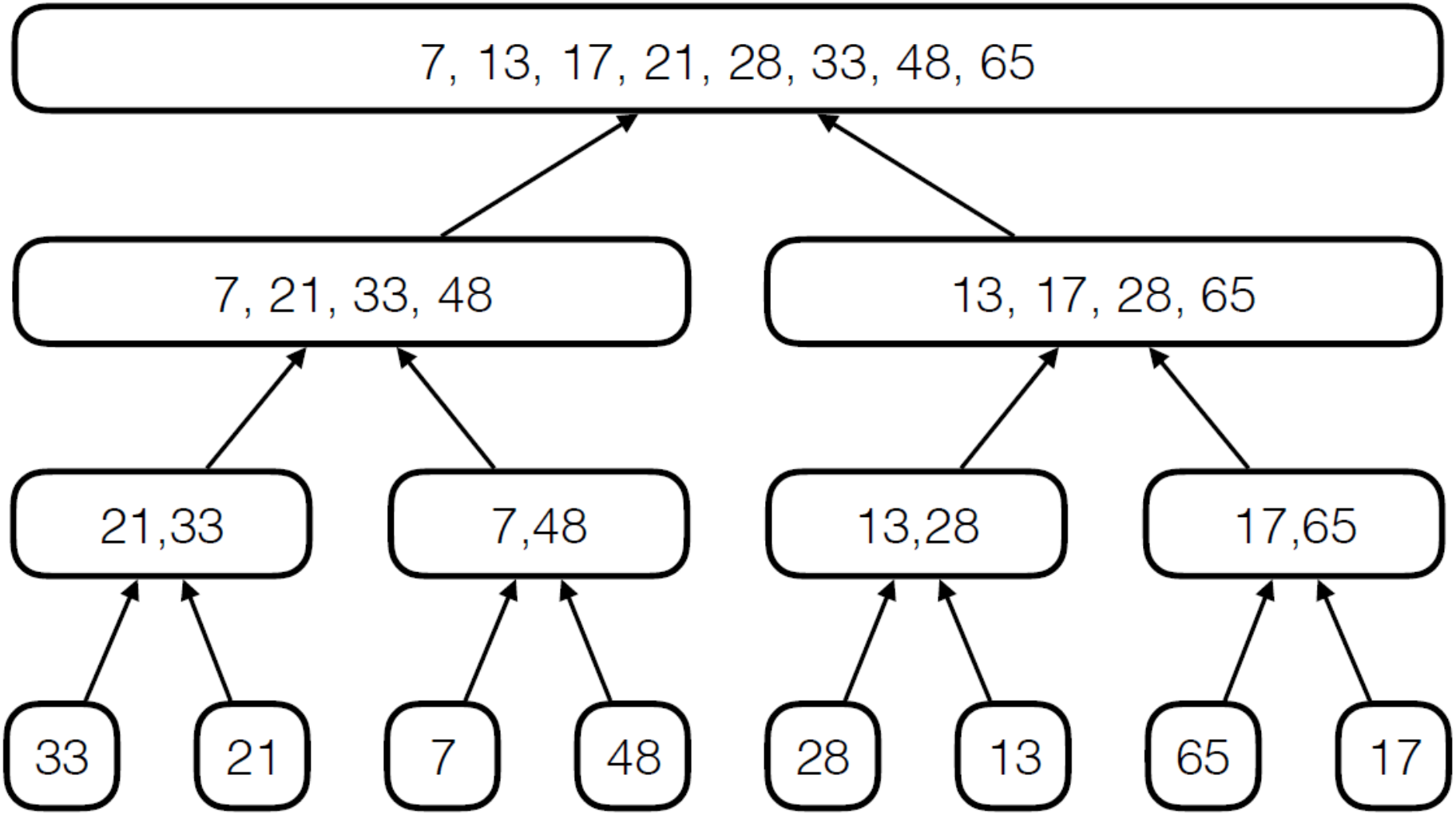
MERGE SORT: esecuzione



MERGE SORT: esecuzione



MERGE SORT: esecuzione



MERGE SORT: analisi

Un'assunzione semplificativa:

- $n = 2^k$, ovvero l'altezza dell'albero di suddivisioni è esattamente $k = \log n$;
- Tutti i sottovettori hanno dimensioni che sono potenze esatte di 2

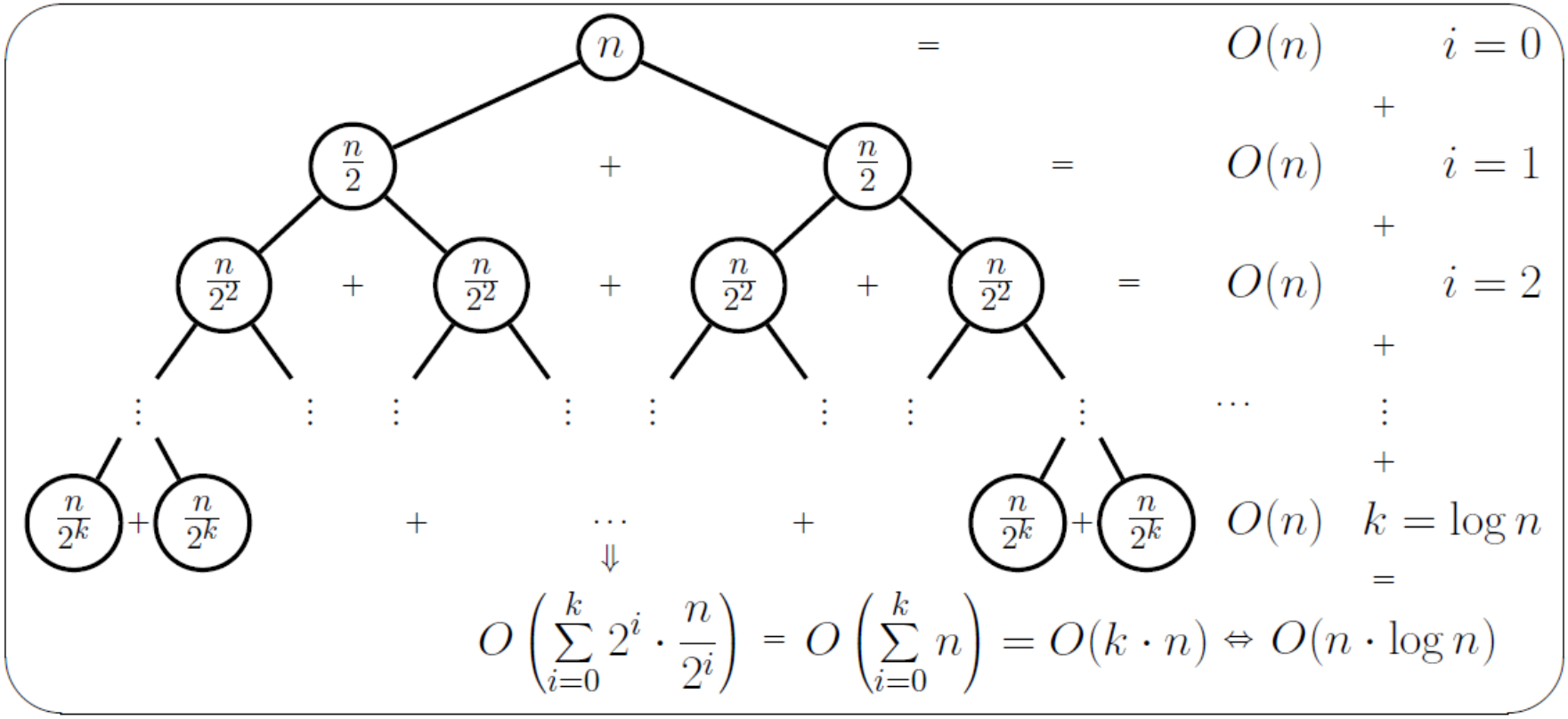
Costo computazionale

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + dn & n > 1 \end{cases}$$

MERGE SORT: analisi

Domanda

Qual è il costo computazionale di MergeSort()?



MERGE SORT: la storia

- Il censimento americano del 1880 aveva richiesto otto anni per essere completato
- Quello del 1890 richiese sei settimane, grazie alla Hollerith Machine
- Fra il 1896 e il 1924, la Hollerith & Co ha cambiato diversi nomi. L'ultimo? International Business Machines
- Le Collating Machines (1936) prendevano due stack di schede perforate ordinate e le ordinavano in un unico stack
- Nel 1945-48, John von Neumann descrisse per la prima volta il MergeSort partendo dall'idea delle Collating Machines.

Hollerith Machine

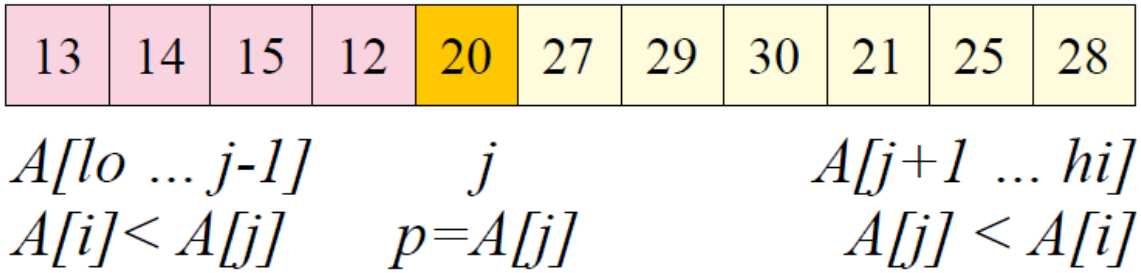
QUICK SORT (Hoare 1961)

Input

- Vettore $A[1 \ldots n]$,
- Indici lo, hi tali che $1 \leq lo \leq hi \leq n$

Divide

- Sceglie un valore $p \in A[lo \ldots hi]$ detto perno (pivot)
- Sposta gli elementi del vettore $A[lo \ldots hi]$ in modo che:



- L'indice j del perno va calcolato opportunamente

QUICK SORT (Hoare 1961)

Impera

Ordina i due sottovettori $A[lo \ldots j-1]$ e $A[j+1 \ldots hi]$ richiamando ricorsivamente Quicksort

Combina

Non fa nulla: infatti,

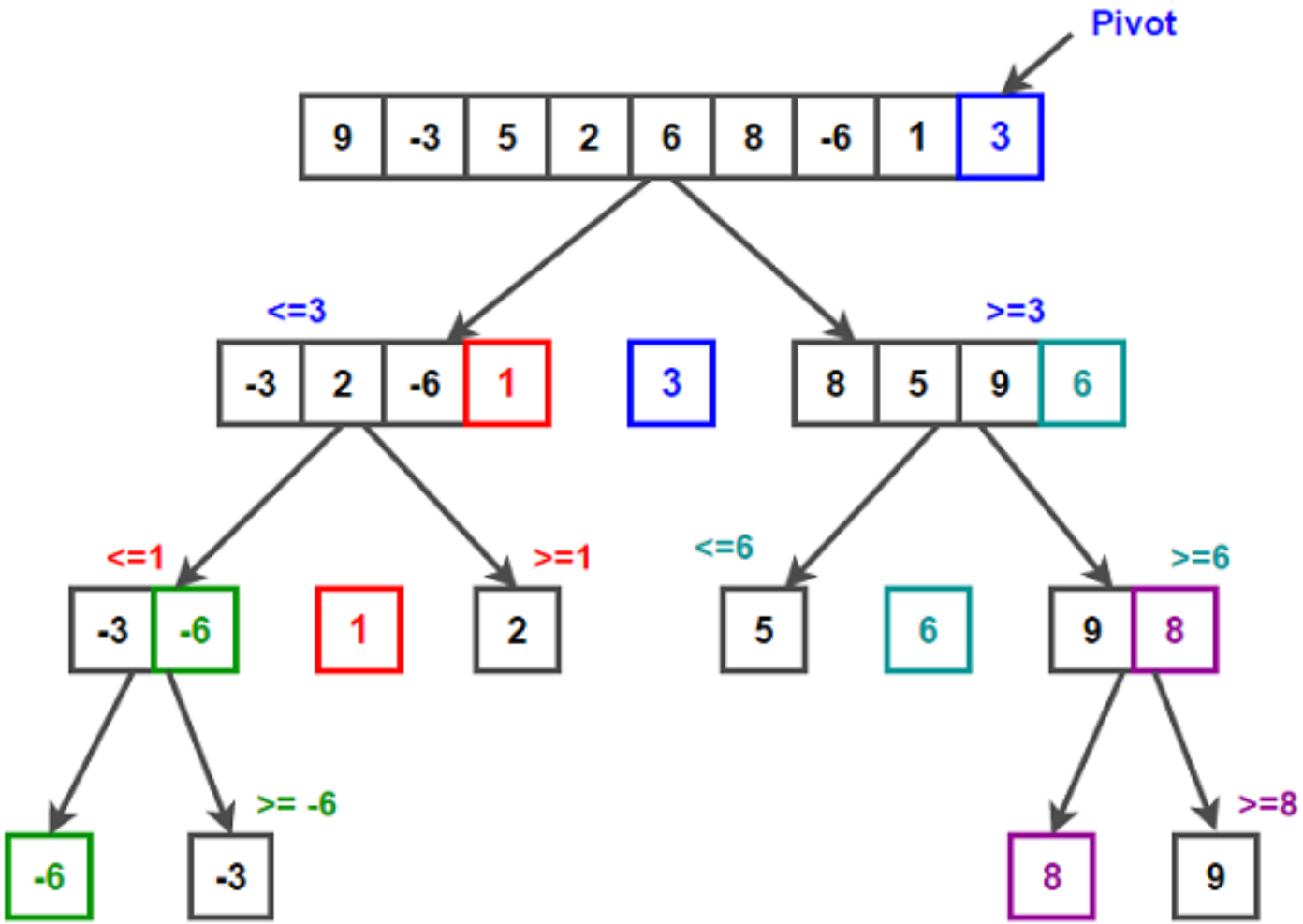
- il primo sottovettore,
- $A[j]$,
- il secondo sottovettore

formano già un vettore ordinato

QUICK SORT (Hoare 1961)

```
void QuickSort(int A[], int first, int last){
int i, j, pivot;
if (first<last) {
     i = first; j = last;
     pivot = v[last];
     do {
          while (v[i] < pivot) i++;
          while (v[j] > pivot) j--;
          if (i <= j) {
               scambia(&v[i], &v[j]);
               i++, j--;}
     } while (i <= j);
     quickSort(v, first, j);
     quickSort(v, i, last);
     }
}
```

QUICK SORT: esempio



QUICK SORT: analisi

Costo Quicksort: caso pessimo?

- Il vettore di dimensione n viene diviso in due sottovettori di dimensione 0 e $n-1$
- $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

Costo Quicksort: caso ottimo?

- Dato un vettore di dimensione n, viene sempre diviso in due sottoproblemi di dimensione $n/2$
- $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$

QUICK SORT: analisi

Caso medio

- Il costo dipende dall'ordine degli elementi, non dai loro valori
- Dobbiamo considerare tutte le possibili permutazioni
- Difficile dal punto di vista analitico

Caso medio: un'intuizione

- Alcuni partizionamenti saranno parzialmente bilanciati
- Altri saranno pessimi
- In media, questi si alterneranno nella sequenza di partizionamenti
- I partizionamenti parzialmente bilanciati "dominano" quelli pessimi

QUICK SORT: analisi

Algoritmo di ordinamento basato su divide-et-impera

- Caso medio: $O(n \log n)$
- Caso pessimo: $O(n^2)$

Caso medio vs caso pessimo

- Il fattore costante di Quicksort è migliore di Merge Sort
- "In-memory": non utilizza memoria addizionale
- Tecniche "euristiche" per evitare il caso pessimo
- Quindi spesso è preferito ad altri algoritmi

R. Sedgewick, "*Implementing Quicksort Programs*". Communications of the ACM, 21(10):847-857, 1978. http://portal.acm.org/citation.cfm?id=359631