Algoritmi e Strutture Dati

Ordinamento

Domenico Fabio Savo

Ordinamento

Problema: dato un insieme **S** di **n** oggetti presi da un dominio totalmente ordinato, ordinare **S**

Esempi: ordinare una lista di nomi alfabeticamente, o un insieme di numeri, o un insieme di compiti d'esame in base al cognome dello studente.

- Subroutine in molti problemi

  E' possibile effettuare ricerche in array ordinati in tempo *O(log n)* (ricerca binaria)

Algoritmi e tempi tipici

- Numerosissimi algoritmi
- Tre tempi tipici: $O(n^2)$, $O(n \log n)$, $O(n)$

n	10	100	1000	10^6	10^9
$n \log_2 n$	~ 33	~ 665	$\sim 10^4$	$\sim 2 \cdot 10^7$	$\sim 3 \cdot 10^{10}$
n^2	100	10^4	10^6	10^{12}	10^{18}

Algoritmi di ordinamento

Approccio incrementale

Approccio incrementale:

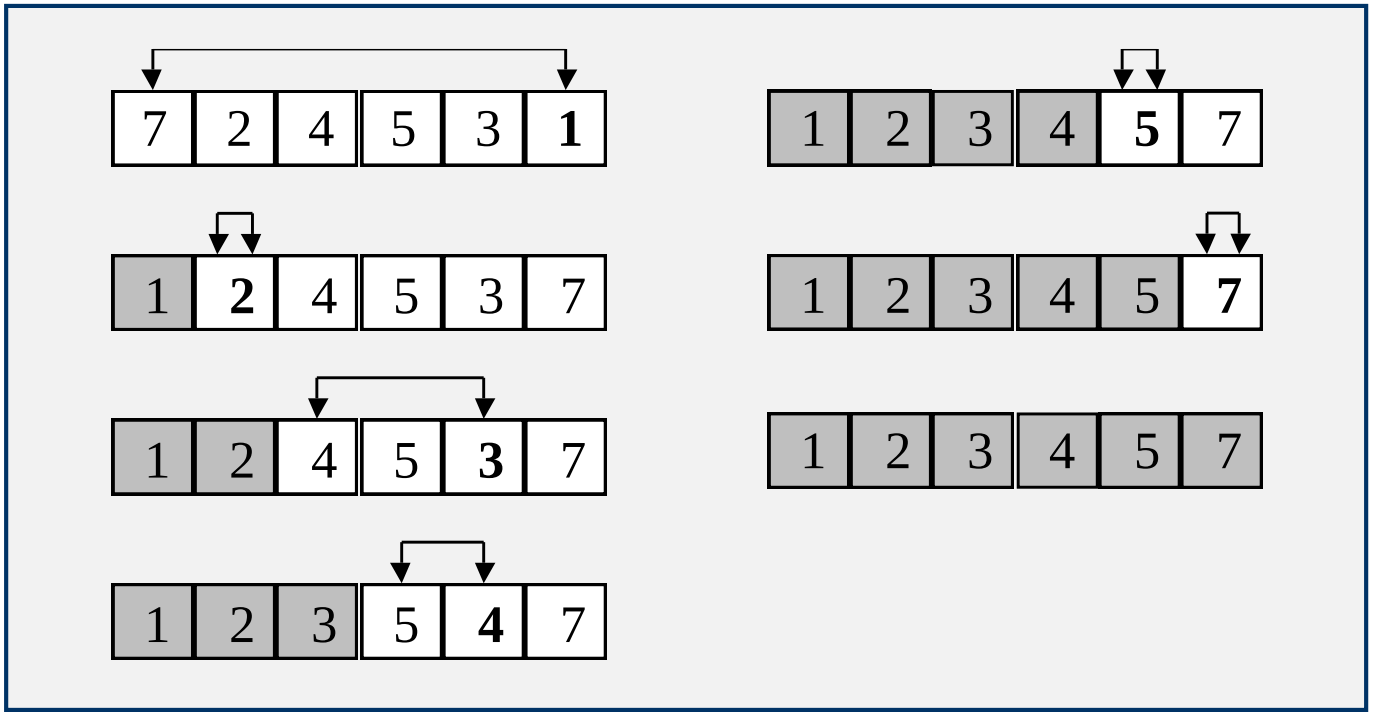
- Supponiamo che *k* elementi di un array di *n* siano già ordinati.
- Estendiamo l'ordinamento da *k* a *k+1*.

Come fare?

- `selectionSort`
- `insertionSort`

SelectionSort

Idea: estende l'ordinamento da k a k+1 elementi, scegliendo il minimo degli n-k elementi non ancora ordinati e mettendolo in posizione k+1



SelectionSort: pseudocodice

```
Algoritmo: selectionSort(array S di dimensione n)
int m;
//Seleziono l'elemento più piccolo nella porzione di array non ordinata
//e lo aggiungo come più grande elemento alla porzione ordinata.
//La porzione di array non ordinata va (ad ogni ciclo) da k a n-1
for(int k = 0; k<= n-2; k++){
        m = k;
        //Seleziona il più piccolo elemento nella pozione di array da
        // ordinare. Tale elemento dovrà essere messo nella posizione k
        for(int j = k+1; j < n; j++){
                if(S[j] < S[m]) then m = j;
        }
        //Se l'elemento individuato si trova in una posizione m diversa da k
        //scambia il valore nella posizione m con quello nella posizione k
        if(m!=k) then swap(S, m, k);
                //La funzione swap scambia tra loro i valori contenuti
                //nelle posizioni n e k dell'array S
        }
}
```

SelectionSort: analisi

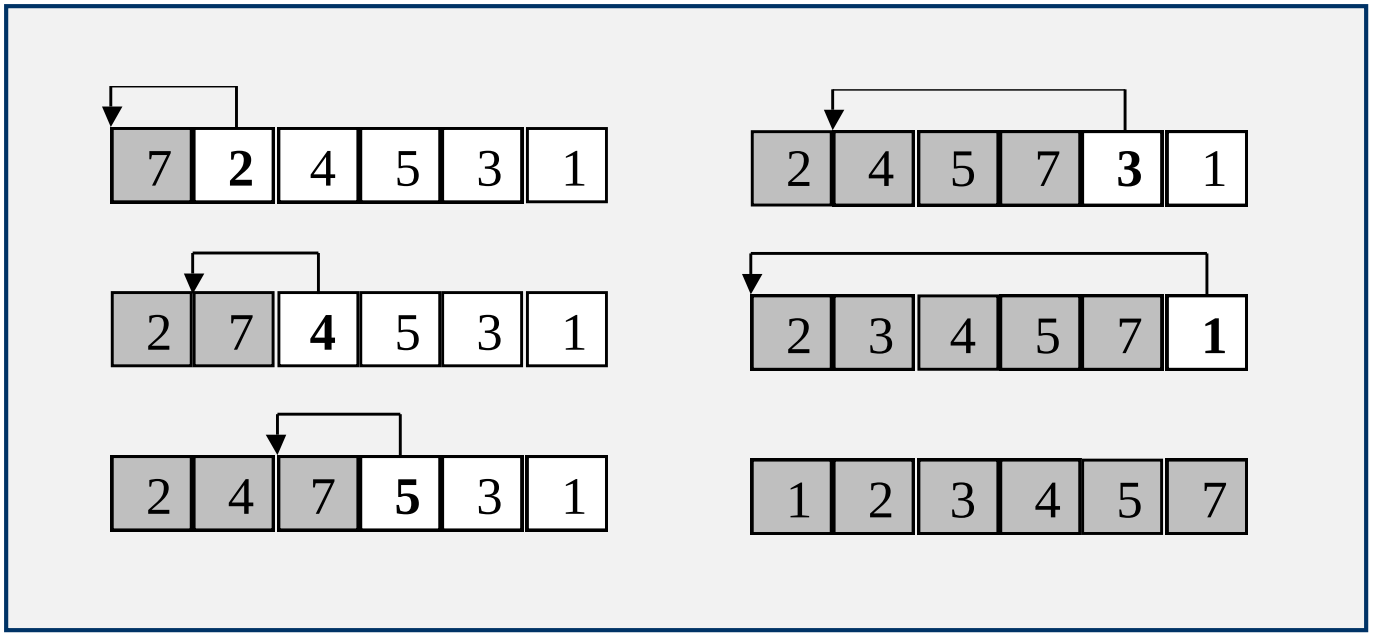
Individuare il k-esimo minimo costa tempo O(n-k). Poichè il ciclo esterno viene eseguito n-1 volte, il numero totale di confronti è dato da:

$$\sum_{k=0}^{n-2} O(n-k) = O\left(\sum_{i=1}^{n-1} i\right) = \mathbf{O(n^2)}$$

usando il cambiamento di variabile i = n-k e la serie aritmetica (vedere Paragrafo 17.2 in Appendice al libro).

InsertionSort

Idea: estende l'ordinamento da k a k+1 elementi, posizionando l'elemento (k+1)-esimo nella posizione corretta rispetto ai primi k elementi



InsertionSort: pseudocodice

```
Algoritmo: insertionSort(array S di dimensione n)
//comincio ad ordinare gli elementi dal secondo elemento nell'array
for(int k = 1; k < n; k++){
    int elm = S[k]; //elem contiene l'elemento da riposizionare
    //cerco l'indice del più piccolo elemento più grande di elm tra
    //quelli che precedono elm in S
    int p = k;
    for(int j = 0; j < k; j++){
        if(S[j] > elm) then{
        p = j;
        break;}
    }
    //Se p precede k, sposto elem da k a p. Prima devo spostare tutti
    // gli elementi tra p e k-1 di una posizione in avanti.
    if(p < k) then {
        for(int i = k; i >= p+1; i--){
        S[i] = S[i-1]; }
    S[p] = elm; //ora che c'è spazio copio elm nella posizione p.
    }
}
```

InsertionSort: analisi

L'inserimento del k-esimo elemento nella posizione corretta rispetto ai primi k richiede tempo O(k) nel caso peggiore

In totale, il tempo di esecuzione è pertanto:

$$O\left(\sum_{k=1}^{n-1} k\right) = \mathbf{O(n^2)}$$

usando la serie aritmetica

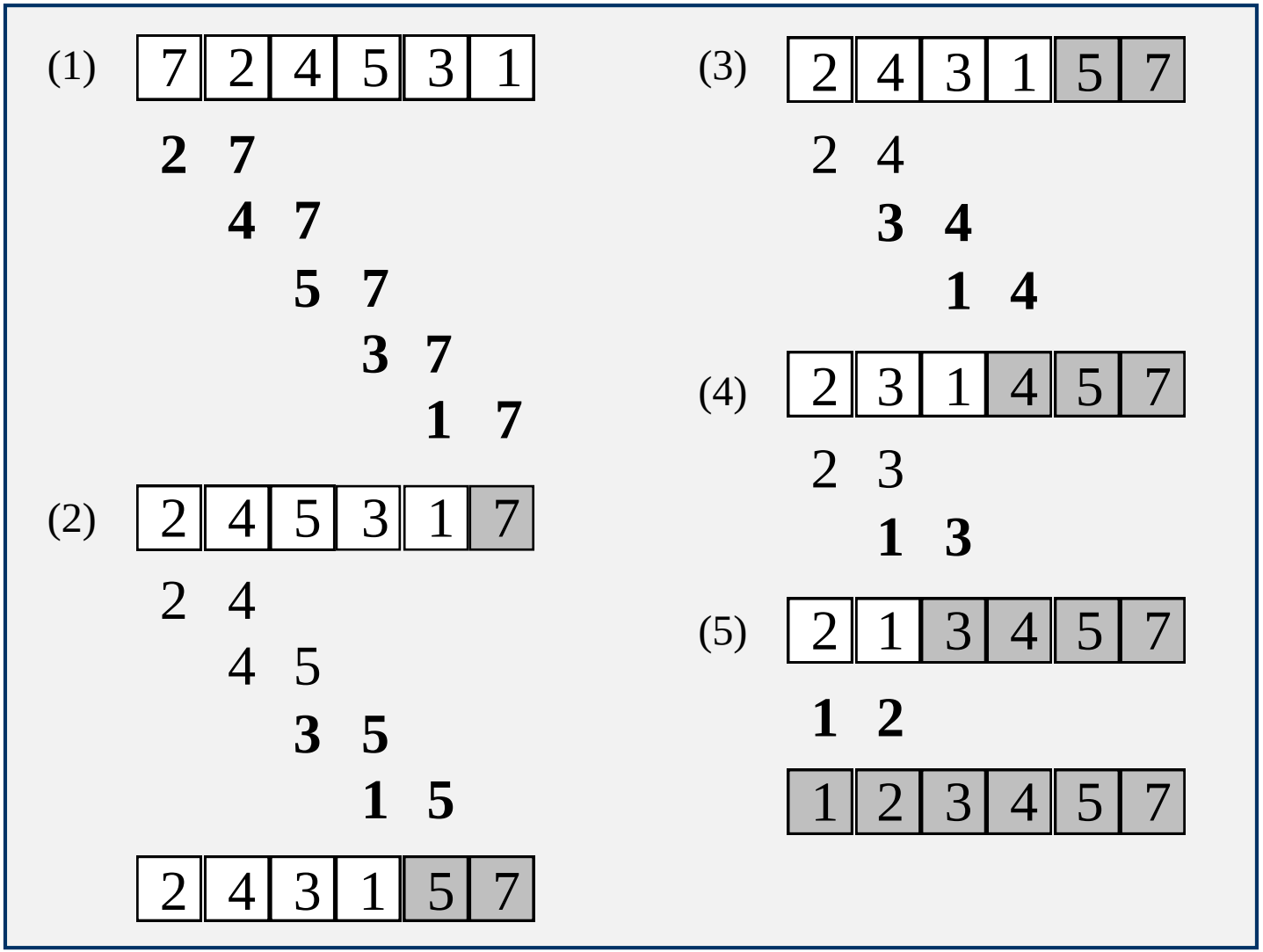
BubbleSort

Esegue una serie di scansioni dell'array:

- In ogni scansione confronta coppie di elementi adiacenti, scambiandoli se non sono nell'ordine corretto
- Dopo una scansione in cui non viene effettuato nessuno scambio l'array è ordinato

- Dopo la k-esima scansione, i k elementi più grandi sono correttamente ordinati ed occupano le k posizioni più a destra.

BubbleSort: esempio di esecuzione



BubbleSort: pseudocodice

```
Algoritmo: bubbleSort(array S di dimensione n)
boolean scambi = false;
for(int i = 0; i < n; i++){
        scambi = false;
        //Confronto tra loro tutte le coppie tra n-i fino
        //all'inizio dell'array. Dati due elementi, se non sono
        //ordinati allora scambio la loro posizione
        for(int j = 1; j < n-i; j++){
                if(S[j-1] > S[j]) then {
                        swap(S, j-1, j);
                        scambi = true;
                }
        }
        if(not scambi) then break;
}
```

BubbleSort: analisi

Dato che dopo la k-esima scansione, i k elementi più grandi sono correttamente ordinati, effettueremo al più (n-1) cicli. Nel caso peggiore, alla k-esima scansione vengono eseguito (n-k) confronti. Pertanto:

$$\sum_{k=1}^{n-1} O(n-k) = O\left(\sum_{j=1}^{n-1} j\right) = \mathbf{O(n^2)}$$

usando il cambiamento di variabile j = n-i e la serie aritmetica (vedere Paragrafo 17.2 in Appendice al libro).

$O(n^2)$ non è un buon risultato: stiamo facendo tutti i confronti possibili!

Possiamo fare di meglio?

HeapSort

- Utilizza lo stesso approccio incrementale utilizzato nel **selectionSort**, ma esegue nel caso peggiore un numero inferiore di confronti grazie all'uso di una struttura dati che individua l'elemento **più grande** in tempo O(log n).
- **NOTA:** eseguiamo l'ordinamento a partire dall'elemento più grande (a differenza del selectionSort dove cominciavamo dall'elemento più piccolo).

- **Struttura dati heap:** associata ad un insieme S di elementi è un **albero binario** con le seguenti proprietà:
  1) È completo fino al penultimo livello (un albero è completo se tutte le foglie sono allo stesso livello)
  2) gli elementi di S sono memorizzati nei nodi dell'albero. Ogni nodo **v** contiene un solo elemento di S che indichiamo con **chiave(v)**.
  3) Vale sa seguente regola: il valore dell'elemento in un nodo è sempre maggiore o uguale al valore degli elementi dei suoi figli.

HeapSort: proprietà di ordinamento

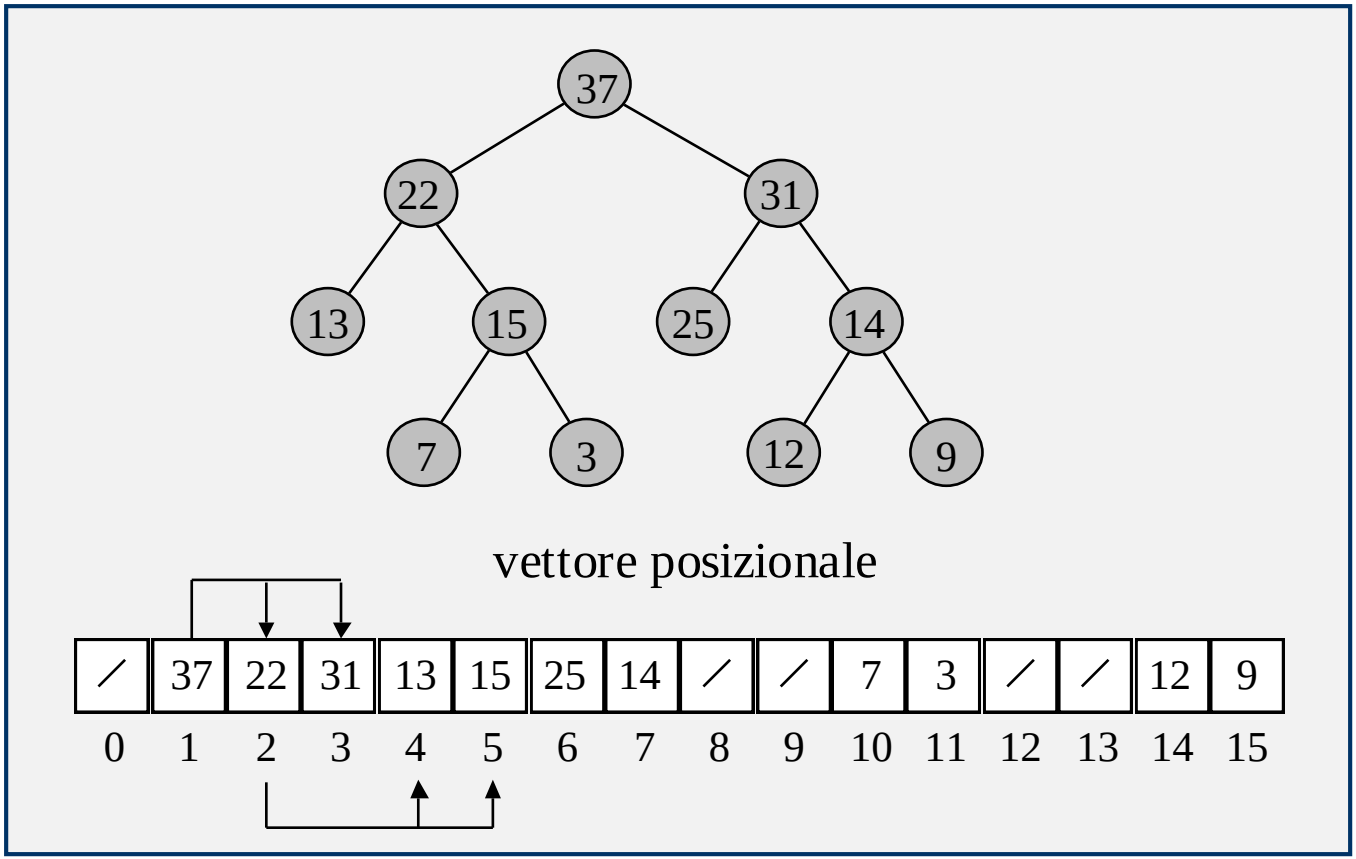
Se dato un nodo **v**, denotiamo rispettivamente con **sin(v)** e **des(v)** il figlio sinistro e destro del nodo **v**, possiamo sintetizzare la **proprietà 3)** di un Heap nel seguente modo:

$$\textbf{chiave(v)} \geq \textbf{chiave(sin(v))}$$

$$\textbf{chiave(v)} \geq \textbf{chiave(des(v))}$$

Struttura dati heap

Rappresentazione ad albero e con vettore posizionale

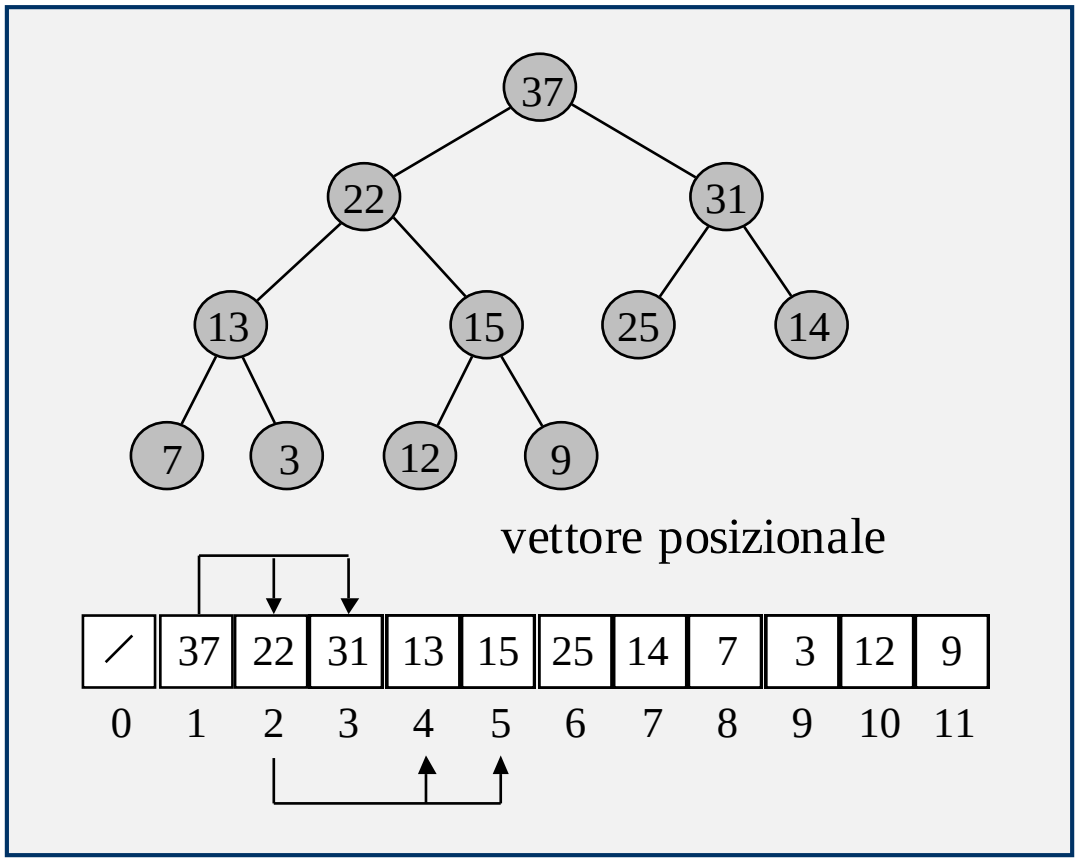


sin(i) = 2i

des(i) = 2i+1

Heap con struttura rafforzata

Le foglie nell'ultimo livello sono compattate a sinistra



Il vettore posizionale ha esattamente dimensione n

Proprietà salienti degli heap

1) Il **massimo** è contenuto nella radice

2) L'albero ha altezza **O(log n)**

3) Gli heap con struttura rafforzata possono essere rappresentati in un array di dimensione pari a **n**

La procedura fixHeap

Se tutti i nodi di H tranne v soddisfano la proprietà di ordinamento a heap, possiamo ripristinarla come segue:

```
Algoritmo fixHeap(nodo v, heap H)
    if (v è una foglia) then return
    else
        sia u il figlio di v con chiave massima
        if ( chiave(v) < chiave(u) ) then
            scambia chiave(v) e chiave(u)
            fixHeap(u,H)
```

Tempo di esecuzione: **O(log n)**

Estrazione del massimo

Dato un heap H vogliamo individuare il massimo valore in esso contenuto e rimuoverlo da H. **Come fare?**

- Sappiamo che il massimo è sempre contenuto nella radice!
- Con quale valore devo però sostituire il valore nella radice affinché la proprietà di ordinamento dell'heap sia ancora valida?

getMax(heap H) → elem **e**

1. elem e ← chiave(radice di H);
2. copio nella radice la chiave contenuta nella la foglia più a destra dell'ultimo livello;
3. rimuovo la foglia;
4. ripristino la proprietà di ordinamento dell'heap richiamando **fixHeap** sulla radice;
5. **return e**;

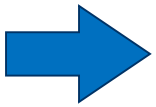
Tempo di esecuzione: **O(log n)**

Costruzione dell'heap

Costruisco un albero binario usando gli elementi del vettore S e poi lo ordino usando il seguente algoritmo ricorsivo basato sul *divide et impera*.

```
Algoritmo heapify(albero H)
    if (H è vuoto) then return
    else
        heapify(sottoalbero sinistro di H)
        heapify(sottoalbero destro di H)
        fixHeap(radice di H, H)
```

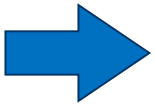
Tempo di esecuzione: **T(n)= 2T(n/2) + O(log n)**



T(n) = **O(n)** dal *Teorema Master*

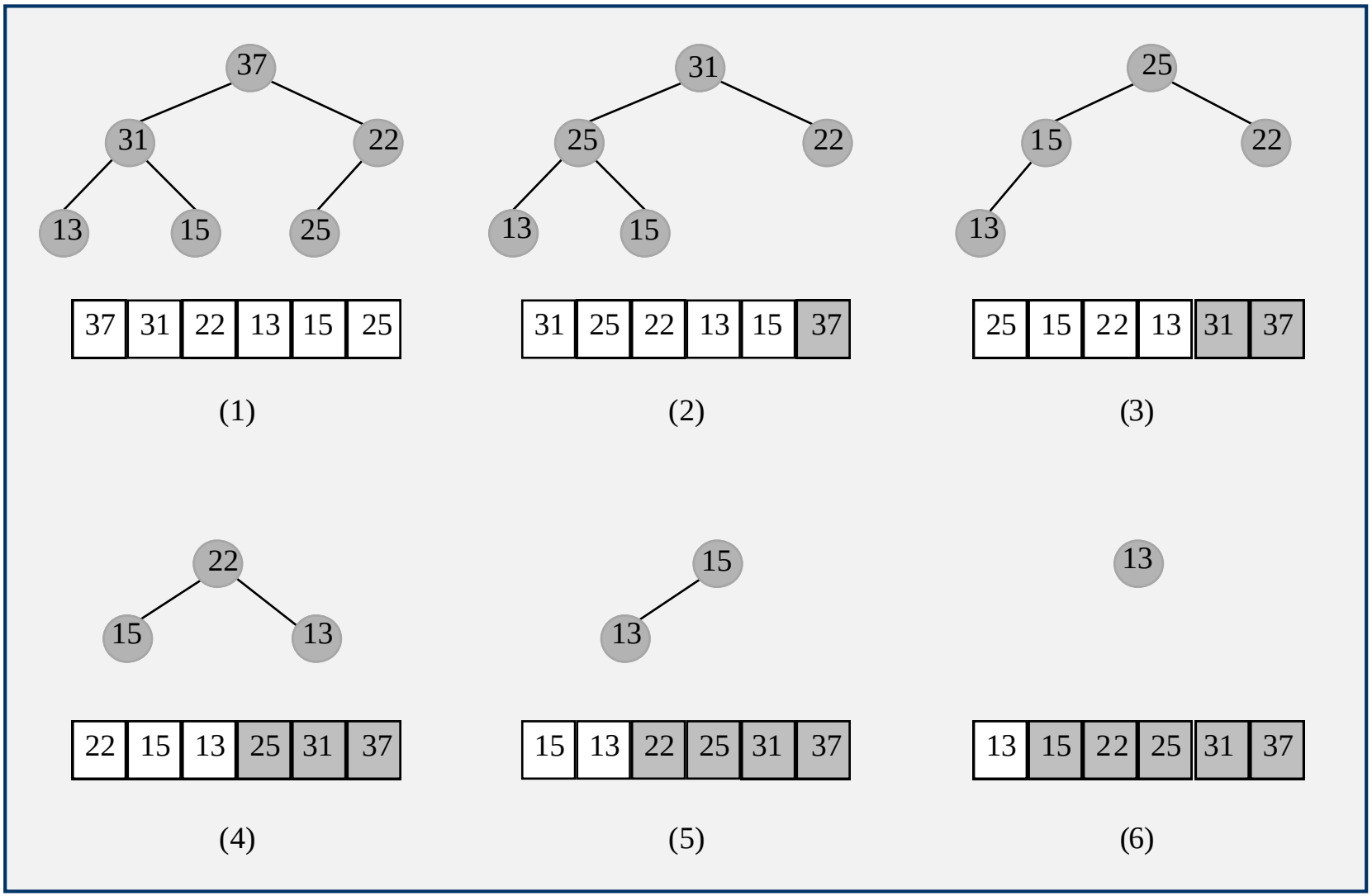
L'algoritmo HeapSort

```
Algoritmo heapSort(array S di dimensione n)
if (S è vuoto) then return S;
else {
    - costruisci un albero binario H con gli elementi di S;
    - ordinalo usando heapyfy(H);//costa O(n)
    - crea un coda C vuota.
    - while(H non è vuoto){        //cicla n volte
        C.enqueue(getMax(H));      //getMax ha costo O(log n)
    - copia gli elementi di C in S nel giusto ordine
    - return S
    }
```



ordina in tempo **O(n log n)**

HeapSort: esempio di esecuzione



MergeSort (1/2)

- Basato sulla tecnica *divide et impera*:
- Vediamo prima il passo ***Impera***: dati due **array ordinati** A e B fonderli a creare un nuovo **array ordinato:**

merge(array A ordinato, array B ordinato) → array X ordinato

1. Sia X un nuovo array vuoto;
2. estrai ripetutamente il minimo di A e B e copialo in X, finché A oppure B non diventa vuoto;
3. copia gli elementi dell'array non vuoto alla fine di X;
4. Restituisci **X**;

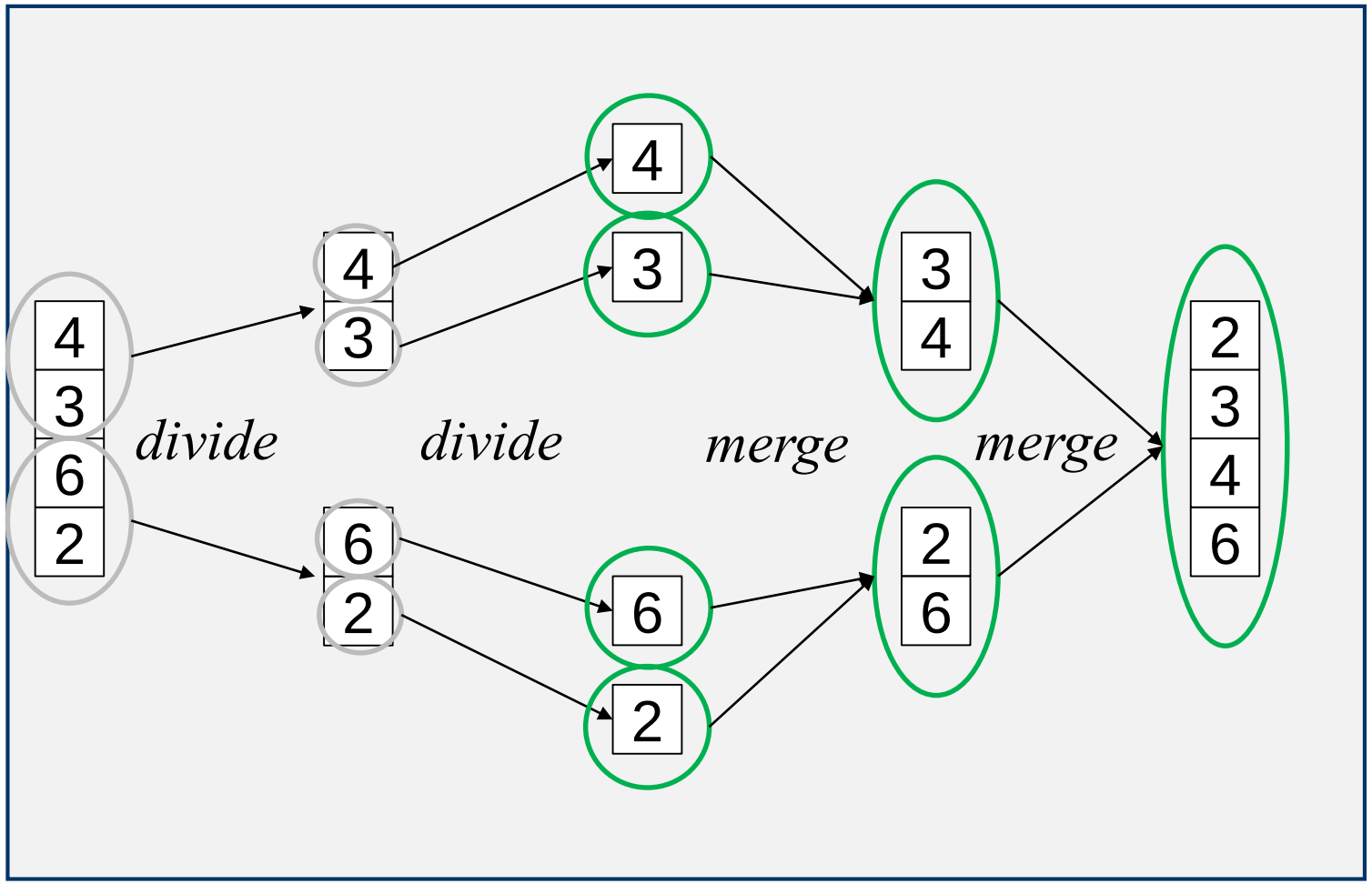
La procedura **merge(A,B)** esegue al più **|A| + |B| +1** confronti

MergeSort (2/2)

Passo ***divide**:* sappiamo come fondere due array ordinati, quindi possiamo dividere l'array in due parti, ordinarli ricorsivamente e poi fonderli nuovamente.

```
Algoritmo mergeSort(array S) → array
if (S è vuoto) then return S;
else { if(|S| = 1) then return S
       else {
              S1 ← mergeSort(metà sinistra di S)
              S2 ← mergeSort(metà destra di S)
              return merge(S1, S2)
              }
     }
```

MergeSort: esempio di esecuzione



Tempo di esecuzione

- Il numero di confronti del MergeSort è descritto dalla seguente relazione di ricorrenza:

$$T(n) = T(n/2) + O(n)$$

- Usando il *Teorema Master* si ottiene:

$$T(n) = \mathbf{O(n \log n)}$$

(stesso costo dell'HeapSort!)

QuickSort

Anche il QuickSort adotta la tecnica del *divide et impera*:

Sia **A** un array:

1. ***Divide***: scegli un elemento **x** di **A** (chiameremo **x** *perno*) e partiziona gli elementi di **A** in due sotto-array in cui il primo contiene tutti gli elementi di **A più piccoli o uguali** ad **x**, ed il secondo tutti gli elementi di **A strettamente maggiori** di **x.**
2. Risolvi i due sottoproblemi ricorsivamente.
3. ***Impera:*** restituisci la concatenazione dei due sotto-array ordinati

A differenza del MergeSort il passo *divide* è complesso, mentre il passo *impera* (che qui consiste nella semplice fusione di due array) è molto semplice

QuickSort: pseudocodice

```
Algoritmo quickSort(array A)
scegli un elemento x di A;
Partizione A rispetto ad x calcolando i due array:
   A1 = { y ∈ A : y ≤ x }
   A2 = { y ∈ A : y > x }
if (|A1| > 1) then quickSort(A1);
if (|A2| > 1) then quickSort(A2);
copia la concatenazione di A1 e A2 in A
```

Calcolare le due partizioni richiede **O(n)** confronti

QuickSort: analisi nel caso peggiore

- Nel caso peggiore, il perno scelto ad ogni passo è il minimo o il massimo degli elementi nell'array.

- Il numero di confronti nel caso peggiore è descritto dalla seguente relazione di ricorrenza:

$$C(n)=C(n-1) + O(n)$$

- Svolgendo per iterazione si ottiene

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

QuickSort: randomizzazione

- Possiamo evitare il caso peggiore scegliendo come perno un elemento dell'array A "a caso".
- Poiché ogni elemento ha la stessa probabilità, pari a 1/n, di essere scelto come perno, il numero di confronti nel caso atteso è:

$$C(n)= \sum_{a=0}^{n-1} \frac{1}{n} \left[n-1+C(a)+C(n-a-1) \right] = n-1+ \sum_{a=0}^{n-1} \frac{2}{n} C(a)$$

dove a e (n-a-1) sono le dimensioni dei sottoproblemi risolti ricorsivamente

QuickSort: analisi nel caso medio

La relazione di ricorrenza: $C(n) = n-1 + \sum_{a=0}^{n-1} \frac{2}{n} C(a)$

ha soluzione $C(n) \leq 2\, n \log n$, pertanto il numero di confronti atteso è **O(n log n)**

Quindi il *quickSort* è meno efficiente nel caso peggiore del *mergeSort* e del *heapSort* e per evitare il caso peggiore lo rendiamo "*randomizzato*".

Il *quickSort* è però molto utilizzato in quanto esistono sue varianti ottimizzate estremamente veloci.

Utilizzando algoritmi come il *mergeSort* e l'*heapSort* possiamo ordinare un insieme di elementi con un numero di confronti pari a **O(n log n)**.

Possiamo fare di meglio?

Lower bound

- Dimostrazione (matematica) che non possiamo andare più veloci di una certa quantità.
- Più precisamente, ogni algoritmo all'interno di un certo **modello di calcolo** deve avere tempo di esecuzione almeno pari a quella quantità.
- Non significa necessariamente che non è possibile fare di meglio, infatti potrebbe essere possibile fare di meglio al di fuori di quel modello di calcolo.

Lower bound per l'ordinamento

- Delimitazione inferiore: quantità di risorsa necessaria per risolvere un determinato problema.
- Dimostrare che **Ω(n log n)** è **lower bound** al numero di confronti richiesti per ordinare n oggetti.

Come?

Dobbiamo dimostrare che tutti gli algoritmi richiedono almeno **Ω(n log n)** confronti!

Modello basato sui confronti

- In questo modello, per ordinare è possibile usare solo confronti tra oggetti
- Primitive quali operazioni aritmetiche (somme o prodotti), logiche (and e or), o altro (shift) sono proibite.
- Sufficientemente generale per catturare le proprietà degli algoritmi più noti

Come dimostrare il lower bound

- Consideriamo un qualsiasi algoritmo **A**, che ordina solo mediante confronti.
- Non abbiamo assolutamente idea di come funziona **A**!
- Ciò nonostante, dimostreremo che **A** deve eseguire almeno **Ω(n log n)** confronti per ordinare un insieme di elementi con cardinalità **n**.
- Dato che l'unica ipotesi su **A** è che ordini mediante confronti, il **lower bound** sarà valido per tutti gli algoritmi basati su confronti!

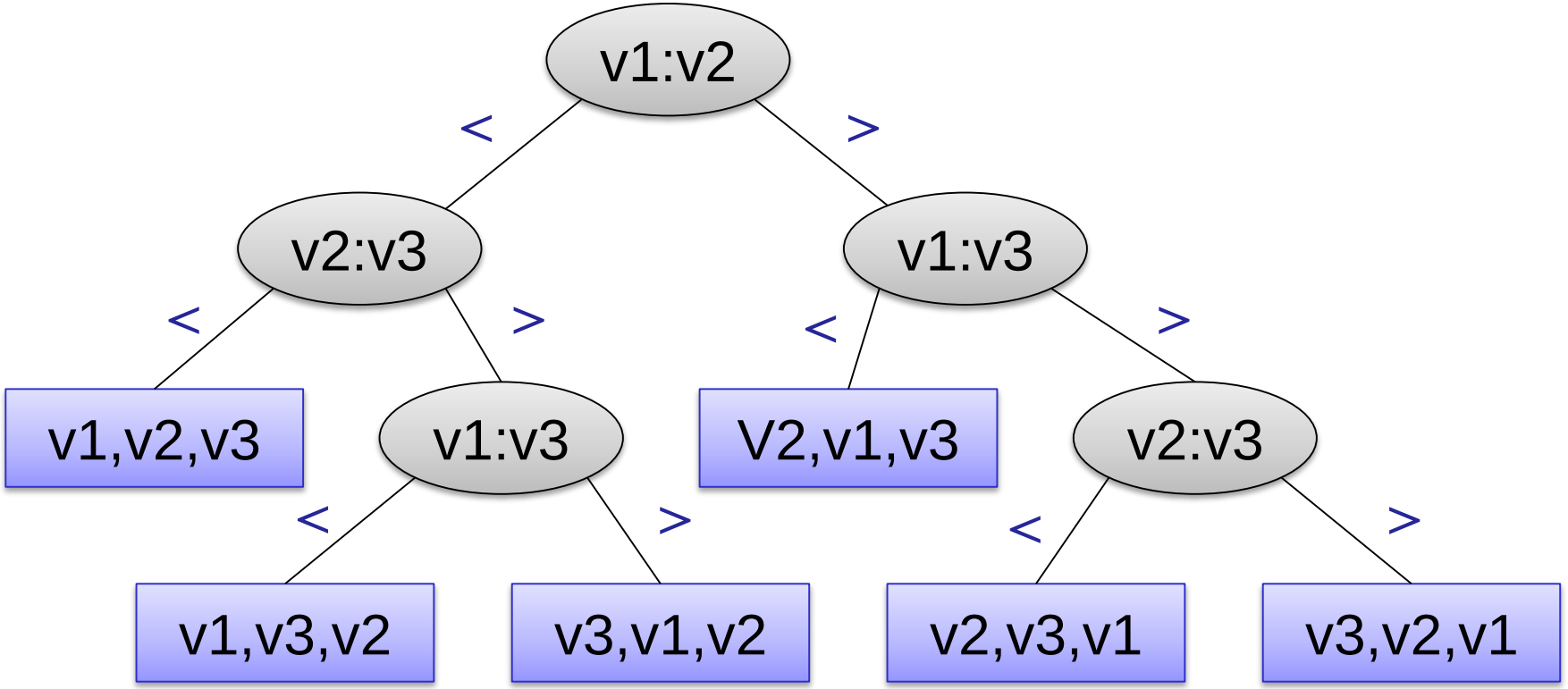
Come dimostrare il lower bound

- Osservazione fondamentale: tutti gli algoritmi devono confrontare elementi
- Dati due valori v1 e v2, i casi possibili sono tre:

  v1 < v2 , v2 > v1 , oppure v1 = v2.
- Per semplicità assumeremo che tutti gli elementi sono tra loro diversi
- Si assume dunque che tutti i confronti abbiano la forma v1 < v2 , e il risultato del confronto sia *vero* o *falso*

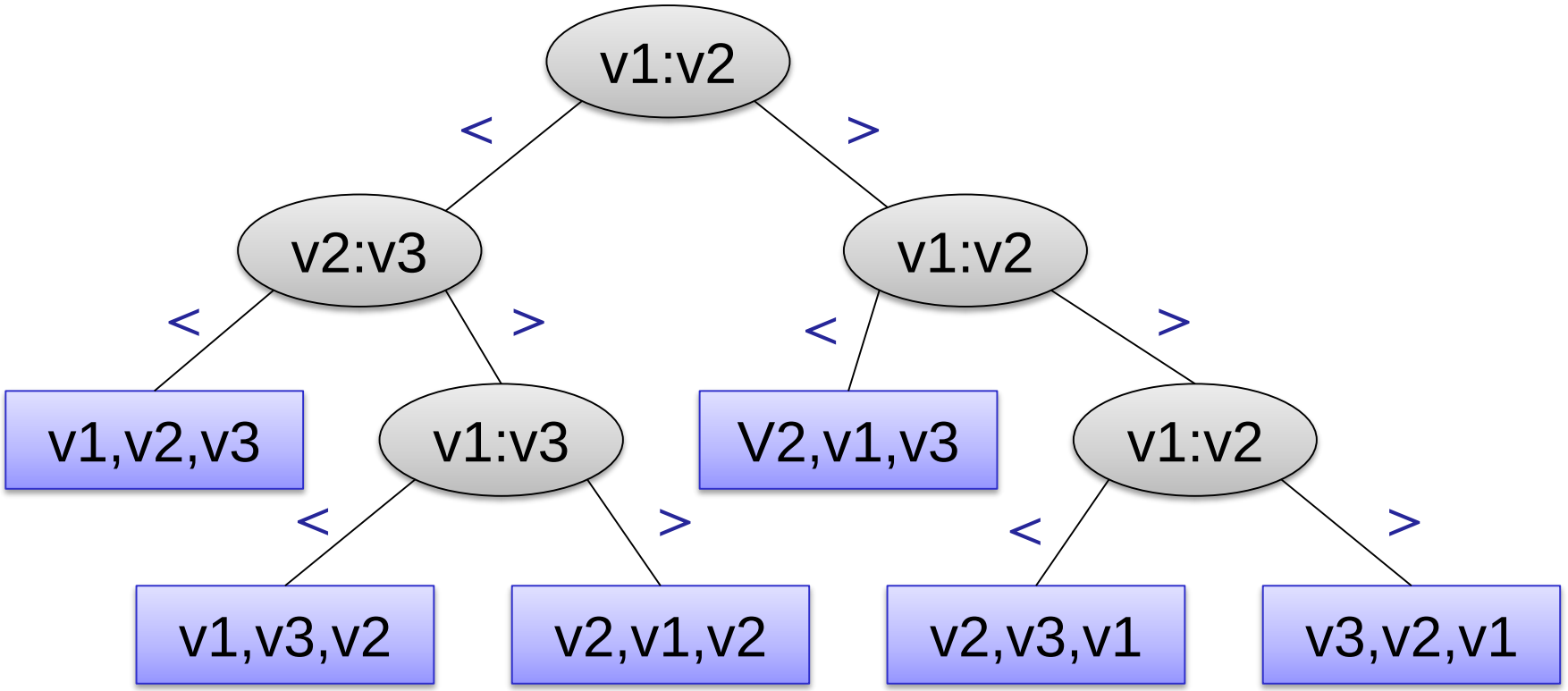
Alberi di decisione (1/3)

- Un albero di decisione rappresenta i confronti eseguiti da un algoritmo su un dato input
- Il seguente esempio indica l'albero di decisione sull'insieme {v1,v2,v3}



Alberi di decisione (2/3)

- Dato un insieme di **n** valori, sono possibili **n!** permutazioni (possibili ordinamenti) → l'albero di decisione deve quindi avere **n!** foglie.
- L'esecuzione di un algoritmo **A** corrisponde ad un cammino sull'albero di decisione corrispondente all'input dato.



Alberi di decisione (2/4)

- Riassumendo:
  - Albero binario;
  - Deve contenere n! foglie
  - Il cammino da una radice ad una foglia rappresenta l'esecuzione dell'algoritmo su un'istanza.
- Il più **lungo** cammino dalla radice ad una foglia rappresenta il numero di confronti che l'algoritmo deve eseguire nel **caso peggiore**.

Il lower bound $\Omega(n \log n)$

- **Teorema:** qualunque albero di decisione che ordina n elementi ha altezza **Ω(n log n)**

- **Corollario:** nessun algoritmo di ordinamento basato sui confronti ha complessità migliore di **Ω(n log n)**

Dimostrazione del teorema

- Un albero di decisione è binario;
- Un albero binario di altezza **h** non ha più di $\mathbf{2^{h-1}}$ foglie
- Il nostro albero deve avere almeno **n!** foglie, quindi:
  - $2^{h-1} \geq n! \rightarrow h-1 \geq \log(n!)$
  - $n! \geq (n/e)^n$ (approssimazione di Stirling)
  - $h-1 \geq \log(n!) \geq \log(n/e)^n = n\log(n) - n\log(e) =$ **Ω(n log n)**

Quindi?

- Un importante corollario del teorema che fissa **Ω(n log n)** come lower bound per il problema dell'ordinamento nel modello basato sui confronti è che gli algoritmi **heapSort** e **mergeSort** sono algoritmi **ottimi** (con complessità asintotica ottima).

Ordinamenti lineari

(per dati di input con proprietà particolari)

Ordinamento di n interi in [1,n]

Abbiano n interi distinti con valore compreso nell'intervallo [1,n]

In quanto tempo possiamo ordinarli?

- O(1) → l'unico ordinamento possibile è {1,2,3,4,5,6,…,n}
- Contraddice il lower bound?
- No! Perché?

Ordinamento di n interi in [1,n]

Abbiano n interi distinti con valore compreso nell'intervallo [1,n]

In quanto tempo possiamo ordinarli?

- O(1) → l'unico ordinamento possibile è {1,2,3,4,5,6,…,n}
- Contraddice il lower bound?
- No! Perché? → **Siamo fuori dal modello basato sui confronti**

IntegerSort: fase 1

Nuovo problema, meno banale:

"ordinare n interi con valori in [1,k]"

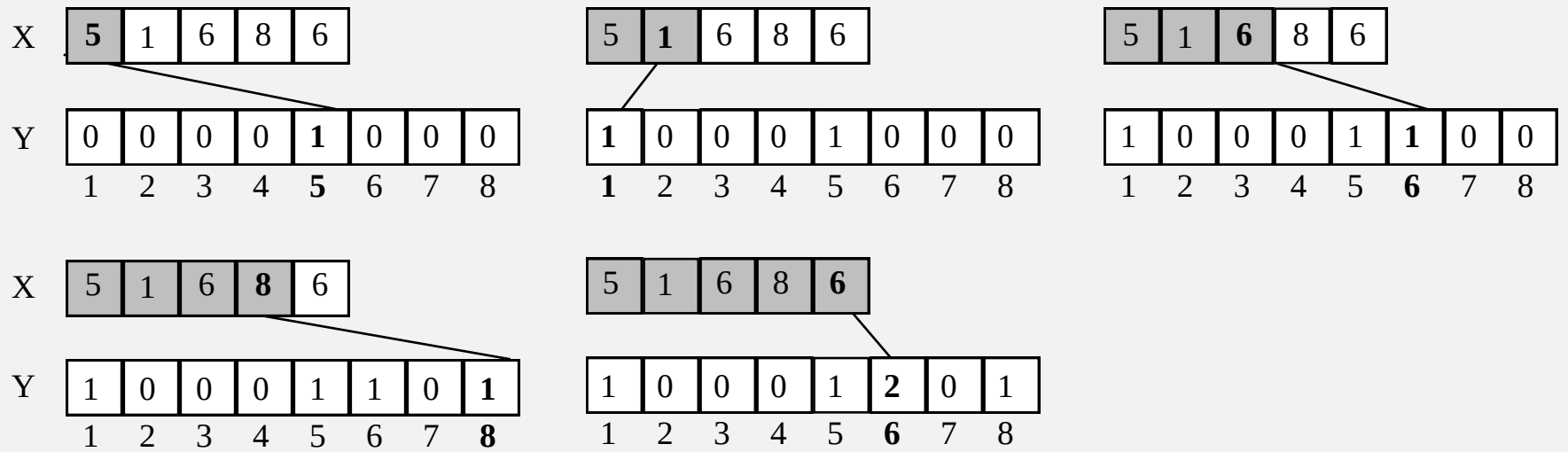
Strategia: mantiene un array Y di k contatori tale che

Y[x] = numero di volte che il valore x compare nell'array di input X

IntegerSort: fase 1

- ESEMPIO:

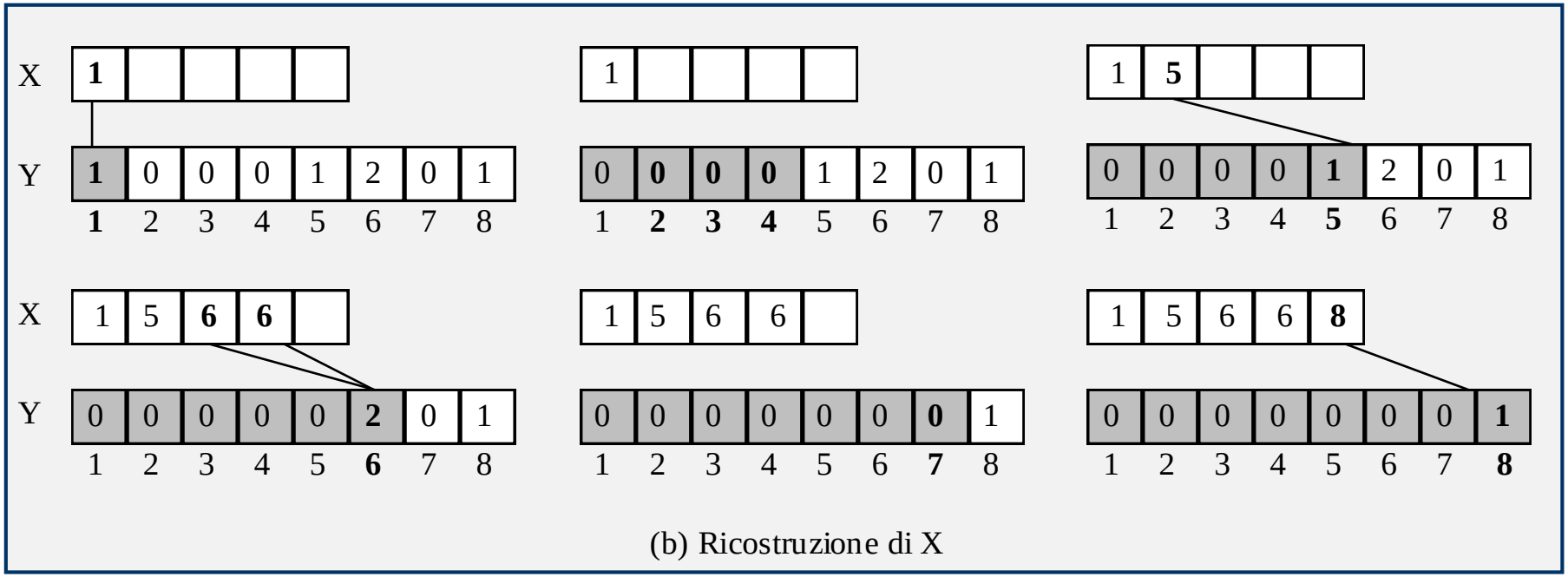
array X di **5** interi compresi tra 1 e 8



(a) Calcolo di Y

IntegerSort: fase 2

Dopo aver contato quanto volte si ripete ogni elemento in X, scorriamo Y da sinistra verso destra e, se Y[x]=k, scriviamo in X il valore x per k volte



(b) Ricostruzione di X

IntegerSort: pseudocodice

```
Algoritmo integerSort(intero k, array X di dimensione n)
Sia Y un array di dimensione k;
//inizializzo tutte le celle di Y a 0;
for(int i ← 0; i < k; i++) { Y[i] ← 0; }
//scandisco X ed ogni volta che X contiene un valore p
//incremento di uno il valore in Y[p]
for(int h ← 0; h < n; h++) { Y[X[h]] ← Y[X[h]] + 1; }
j ← 1;
//scandisco Y, se Y[t]=k con k > 0, scrivo in X
//il valore t per k volte
for( t ← 0; t < k; t++ ) {
    while( Y[t] > 0 ){
        X[j] ← t; j++; Y[t] ← Y[t]-1;
    }
}
```

IntegerSort: analisi

L'algoritmo IntegerSort richiede:

- Tempo **O(k)** per inizializzare **Y** (creo **Y** e inizializzo ogni sua cella a 0)
- Tempo **O(n)** per scandire **X** (contare i sui valori e incrementare i contatori in **Y**)
- Tempo **O(n+k)** per ricostruire **X** (scandire **Y** e riscrivere gli **n** valori in **X**)

Quindi *integerSort* richiede un tempo pari a **O(n+k)**. Questo significa che, finché **k** è proporzionale con **n** (**k = O(n)**), il costo di **integerSort** è lineare con **n** (**O(n)**). Questa proprietà di perde se si possono avere valori di **k** molto grandi.

Es. per $\mathbf{k = n^5}$ il costo di *integerSort* sarà $\mathbf{O(n + n^5) = O(n^5)}$.

BucketSort

E se invece di interi vogliamo ordinare altri tipi di elementi (esempio le persone in una lista in base al loro mese di nascita)?

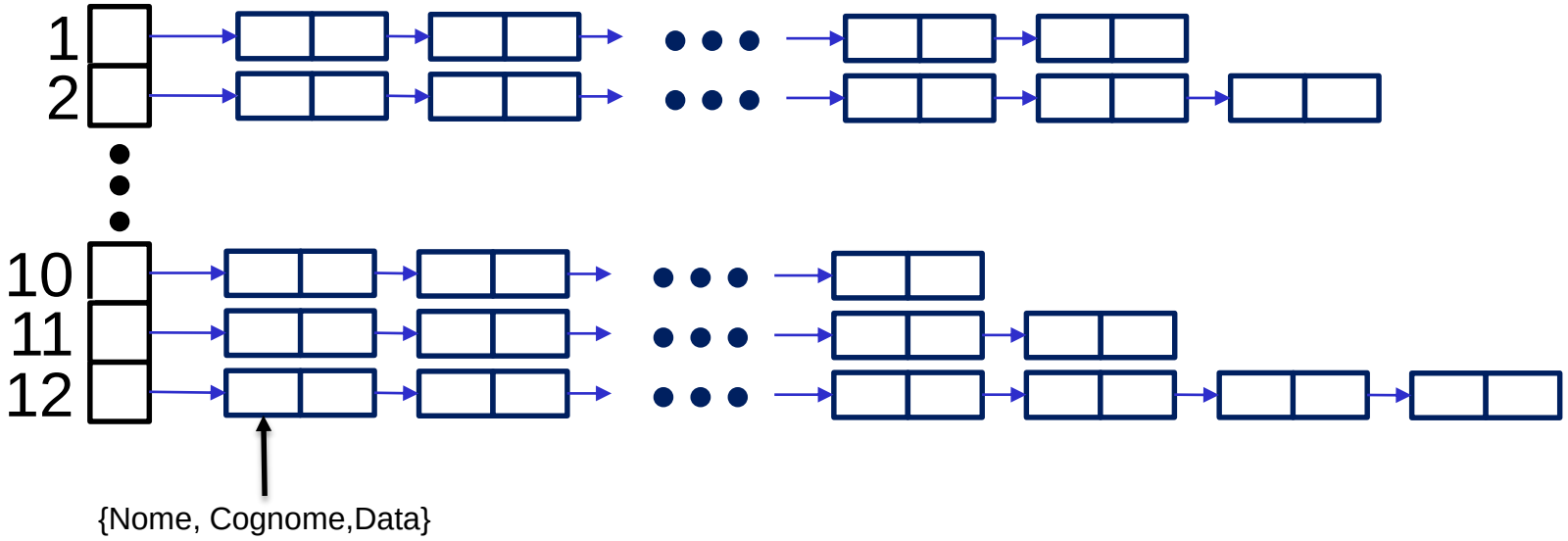
Per ordinare **n** record con chiavi intere in **[1,k]**

- Basta mantenere un **array di liste**, anziché di contatori, ed operare come per IntegerSort

- La lista **Y[x]** conterrà gli elementi con chiave uguale a **x**
- Per ottenere una lista ordinata vado poi a concatenare le liste in Y
- Tempo **O(n+k)** come per IntegerSort

BucketSort: esempio

- Ho una lista di 1000 record contenenti le seguenti informazioni sulle persone:
  - Nome; Cognome; Data di nascita (gg/mm/aaaa)
- Ordinare i record sulla base del mese di nascita delle persone
- Ci sono solo 12 mesi quindi userò un array di liste con 12 caselle



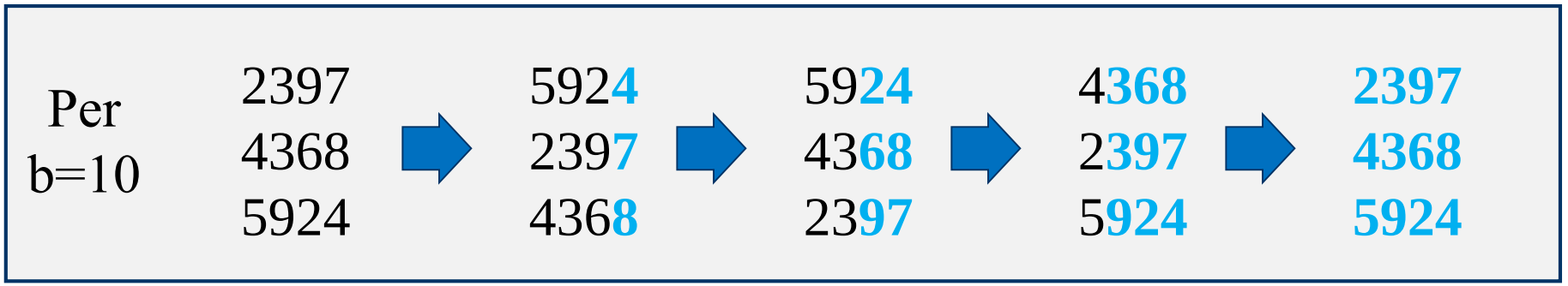
Stabilità

- Un algoritmo è stabile se preserva l'ordine iniziale tra elementi con la stessa chiave
- Il *BucketSort* può essere reso stabile appendendo gli elementi di X in coda alla opportuna lista Y[i]

RadixSort

Come abbiamo visto l'*BucketSort* funziona bene quando **k** è dello stesso ordine di **n.** Che fare quando **k** è molto grande?

- Rappresentiamo gli elementi in base **b** (binario, decimale. esadecimale, ecc.), ed eseguiamo una serie di *BucketSort*
- Partiamo ordinando i numero usando come chiave la cifra meno significativa e poi ci spostiamo verso quella più significativa.



RadixSort: correttezza

- Se **x** e **y** hanno una diversa **t**-esima cifra, la **t**-esima passata di *BucketSort* li ordina
- Se **x** e **y** hanno la stessa **t**-esima cifra, la proprietà di stabilità del *BucketSort* li mantiene ordinati correttamente

Dopo la **t**-esima passata di *BucketSort*, i numeri sono correttamente ordinati rispetto alle **t** cifre meno significative

RadixSort: tempo di esecuzione

Se usiamo come base per il *bucketSort* un valore **b = Θ(n),** l'algoritmo *radixSort* ordina **n** numeri interi compresi tra **[1,k]**, in tempo:

$$T(n) = O(n(\frac{\log(k)}{\log(n)}))$$