Sorting algorithms

Luciano Bononi
Dip. di Scienze dell'Informazione
Università di Bologna

bononi@cs.unibo.it

Sorting

- Let's consider an array of n numbers v[1], v[2], ... v[n]
- We want to find a permutation

  p[1], p[2], ... p[n]

  of the integer values 1, ..., n such that

  v[p[1]] ≤ v[p[2]] ≤ ... ≤ v[p[n]]
- Example:
  - v = [7, 32, 88, 21, 92, -4]
  - p = [6, 1, 4, 2, 3, 5]
  - v[p[]] = [-4, 7, 21, 32, 88, 92]

Sorting

- In general:: is given an array of n elements, each one composed by:
  - a **key**, (mutually comparable)
  - An arbitrary **content**
- We want to create a permutation where the keys appear in increasing (or decreasing) order.

Definitions

- *Sorting is* on site
  - The algorithm creates a permutation on site, without additional array
- Sorting is stable
  - The algorithm preserves the original order of values with the same key in the original array

Anno nascita	Nome
1974	Mario Rossi
1953	Luigi Bianchi
1974	Antonio Verdi
1953	Fabio Neri

Not stable since the arrows intersects

Anno nascita	Nome
1953	Luigi Bianchi
1953	Fabio Neri
1974	Antonio Verdi
1974	Mario Rossi

Note

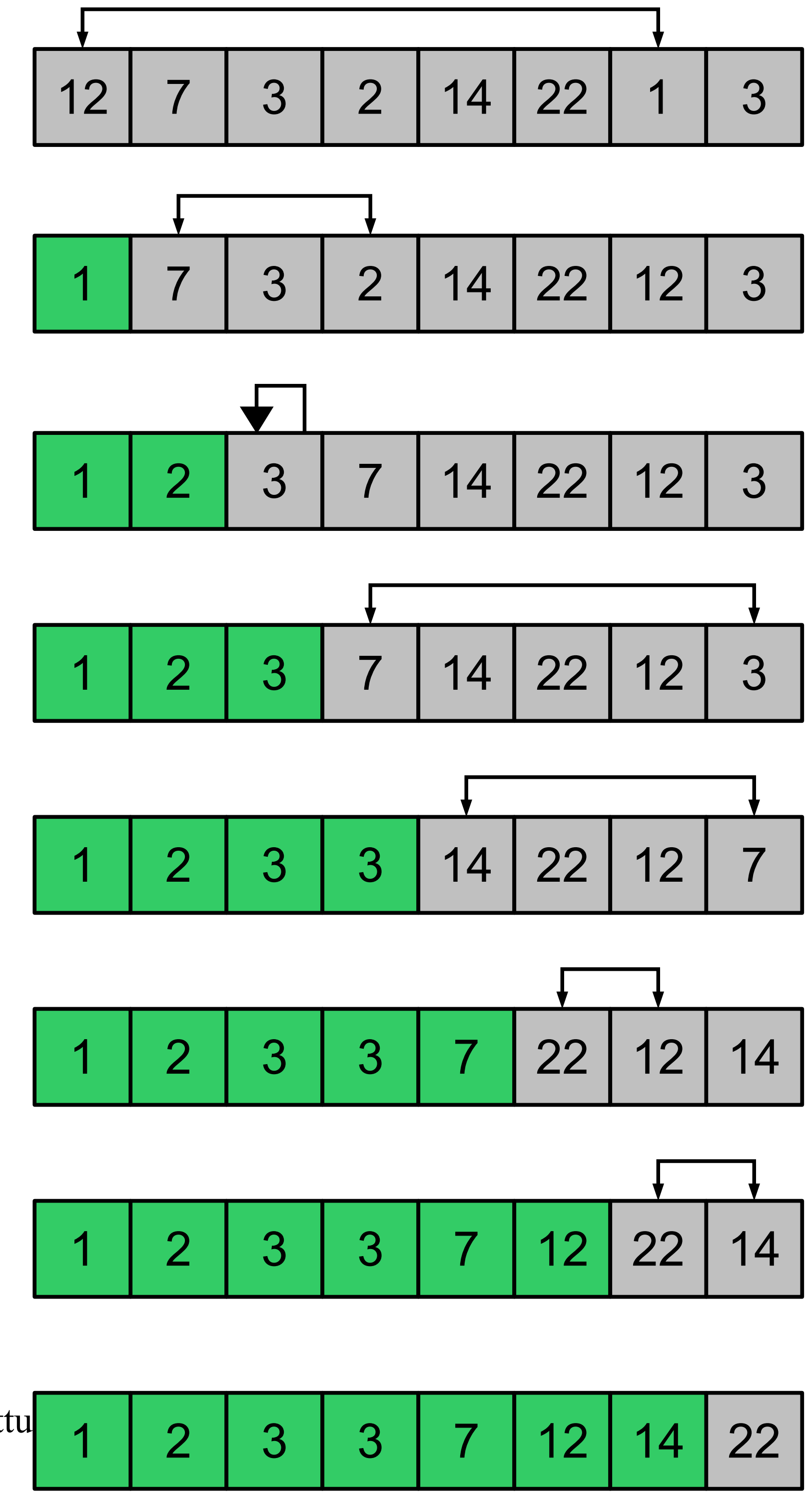
- It is possible to make every algorithm to be stable:
  - It is sufficient ot use the pair (key, position in the unordered array) as the ordering key.
  - (k1, p1) < (k2, p2) in and only if:
    - (k1 < k2), or
    - (k1 == k2) and (p1 < p2)

“incremental” sorting algorithms

- Starting from an ordered prefix A[1..k], they “extend” the ordered part of one additional element: A[1..k+1]
- Selection sort
  - Finds the min in A[k+1..n] and move it in position k+1
- Insertion sort
  - Inserte the element A[k+1] in the correct position inside the ordered prefix A[1..k]

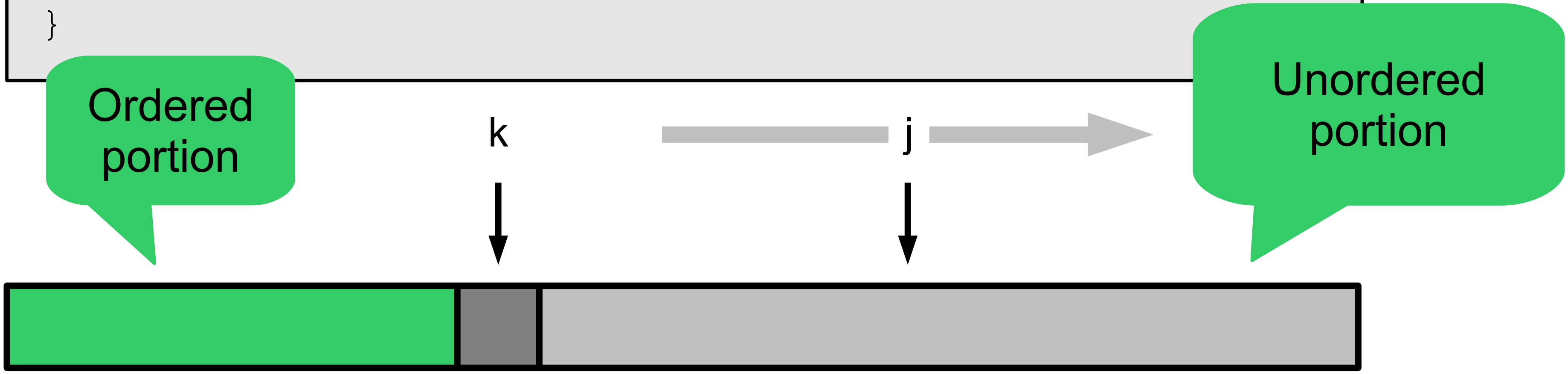
Selection Sort

- Find the min in A[1]...A[n] and swap with A[1]
- Find the min in A[2]...A[n] and swap with A[2]
- ...
- Find the min in A[k]...A[n] and swap with A[k]
- ...



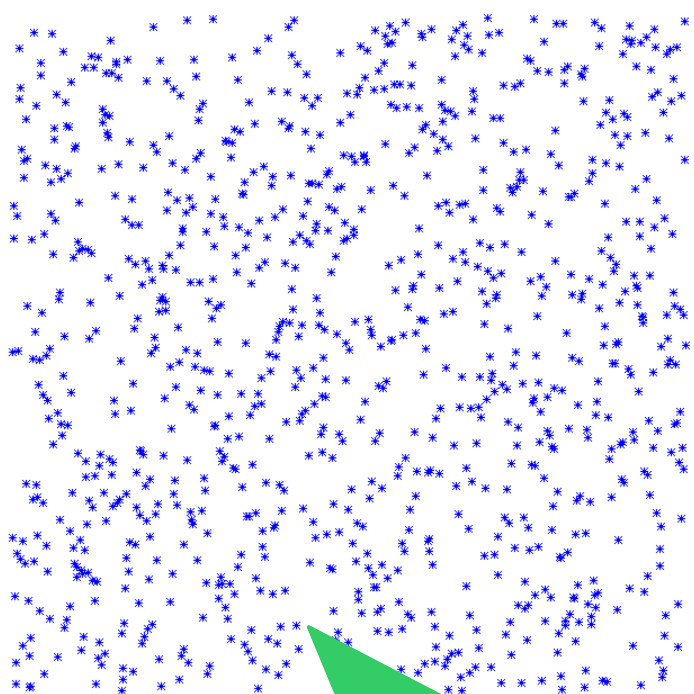
Selection Sort

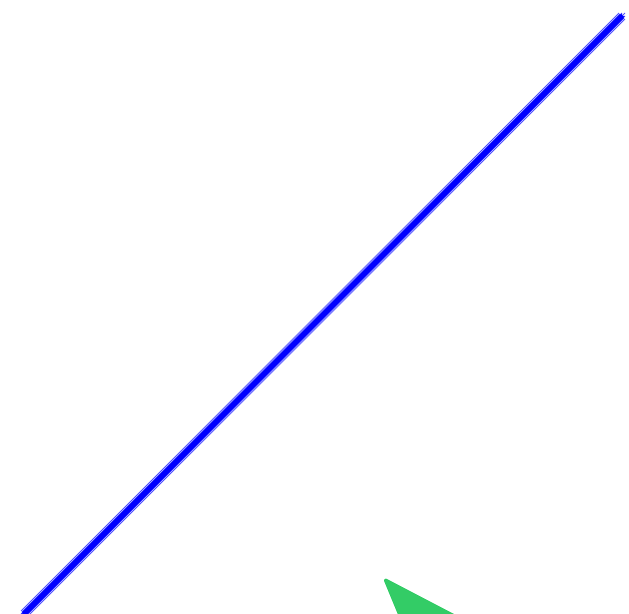
```
public static void selectionSort(Comparable A[]) {
    for (int k = 0; k < A.length - 1; k++) {
        // find min A[m] in A[k..n-1]
        int m = k;
        for (int j = k + 1; j < A.length; j++)
            if (A[j].compareTo(A[m]) < 0)
                m = j;
        // swap A[k] with A[m]
        if (m != k) {
            Comparable temp = A[m];
            A[m] = A[k];
            A[k] = temp;
        }
    }
}
```



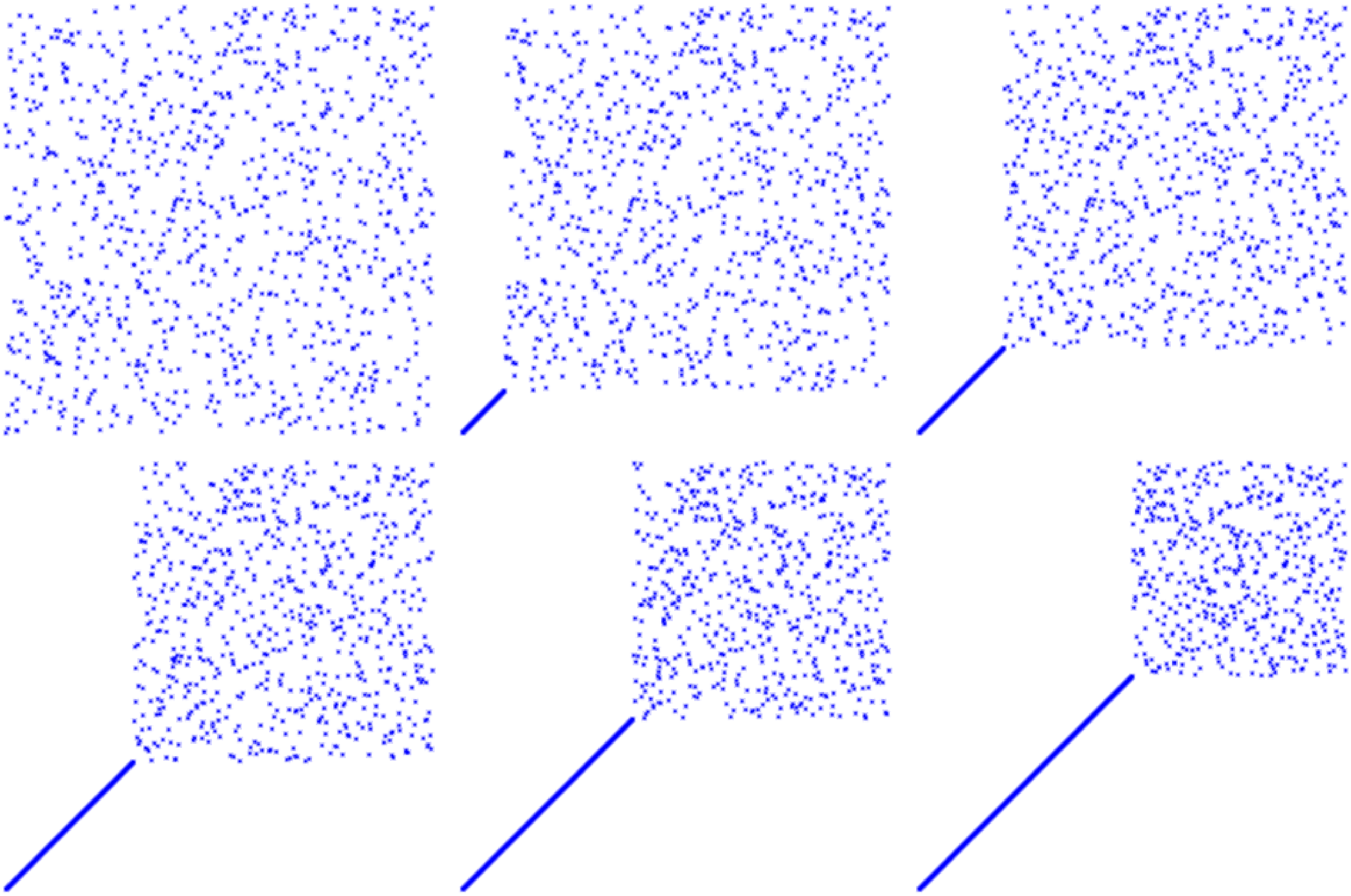
How to visualize the behavior of a sorting algorithm

- Given an array A[] containing all the integers between 1 and N
- We plot each element as the point (i, A[i])





Selection Sort image after image



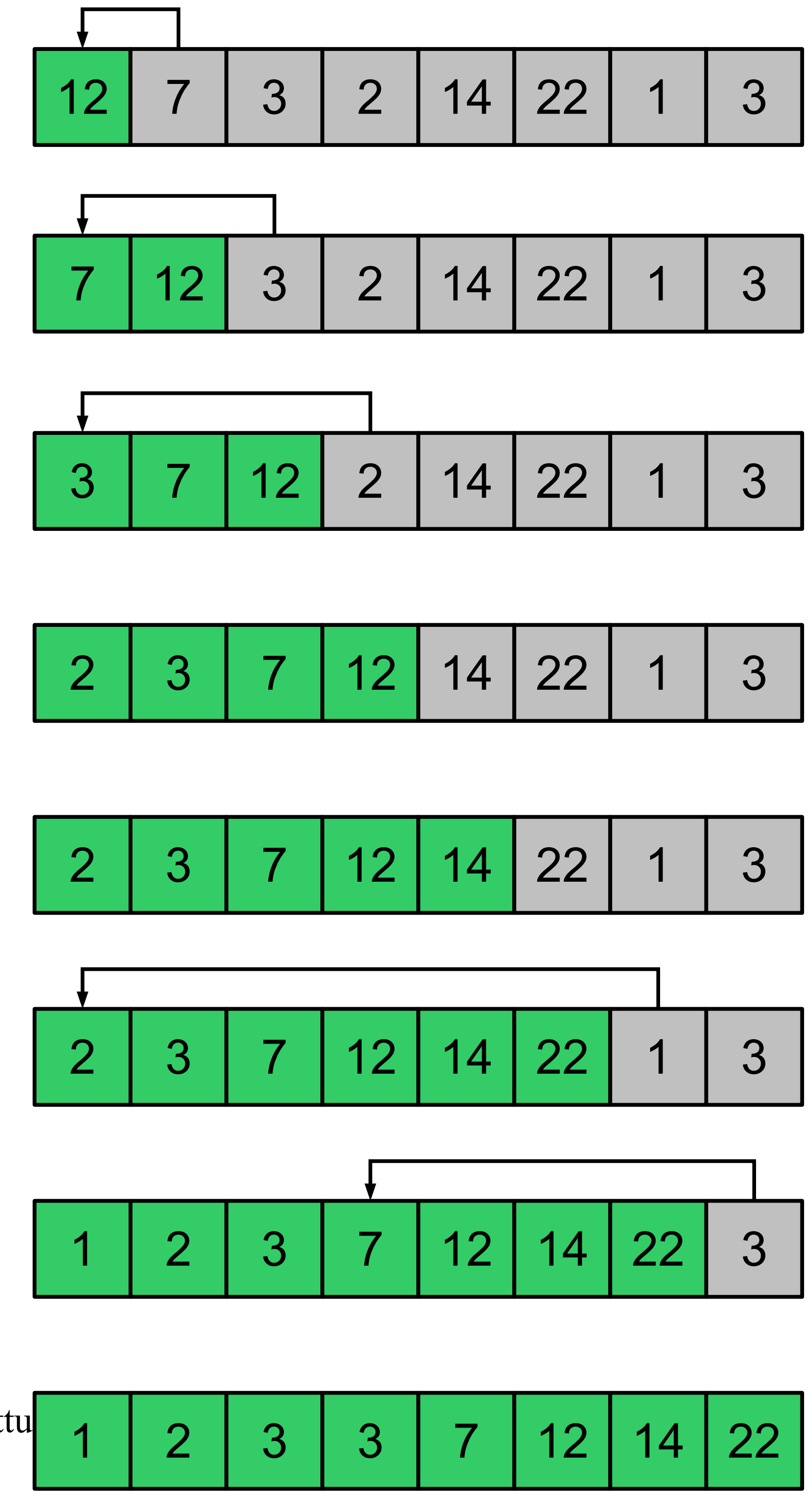
Complexity of Selection Sort

- To extract the k-th min takes *(n-k-1)* comparisons (k=0,1, … n-2)
- The whole cost is

$$\sum_{k=0}^{n-2} (n-k-1) = \sum_{k=1}^{n-1} k = \Theta(n^2)$$

Insertion Sort

- Idea: after step k, the array has the first k elements ordered
- We insert the k+1-th element in the correct position inside the first k ordered elements



Insertion Sort

```
public static void insertionSort(Comparable A[]) {
    for (int k = 1; k <= A.length - 1; k++) {
        int j;
        Comparable x = A[k];
        // find position j where to insert A[k]
        for (j = 0; j < k; j++)
            if (A[j].compareTo(x) > 0) break;
        if (j < k) {
            // Sposta A[j..k-1] in A[j+1..k]
            for (int t = k; t > j; t--)
                A[t] = A[t - 1];
            // Insert A[k] in position j
            A[j] = x;
        }
    }
}
```

Question: is this a stable ordering algorithm?

Insertion Sort

- Insertion of k+1-th element in the correct position given first *k* ordered elements takes *k+1* comparisons in the worst case
- The number of comparisons in the worst case is

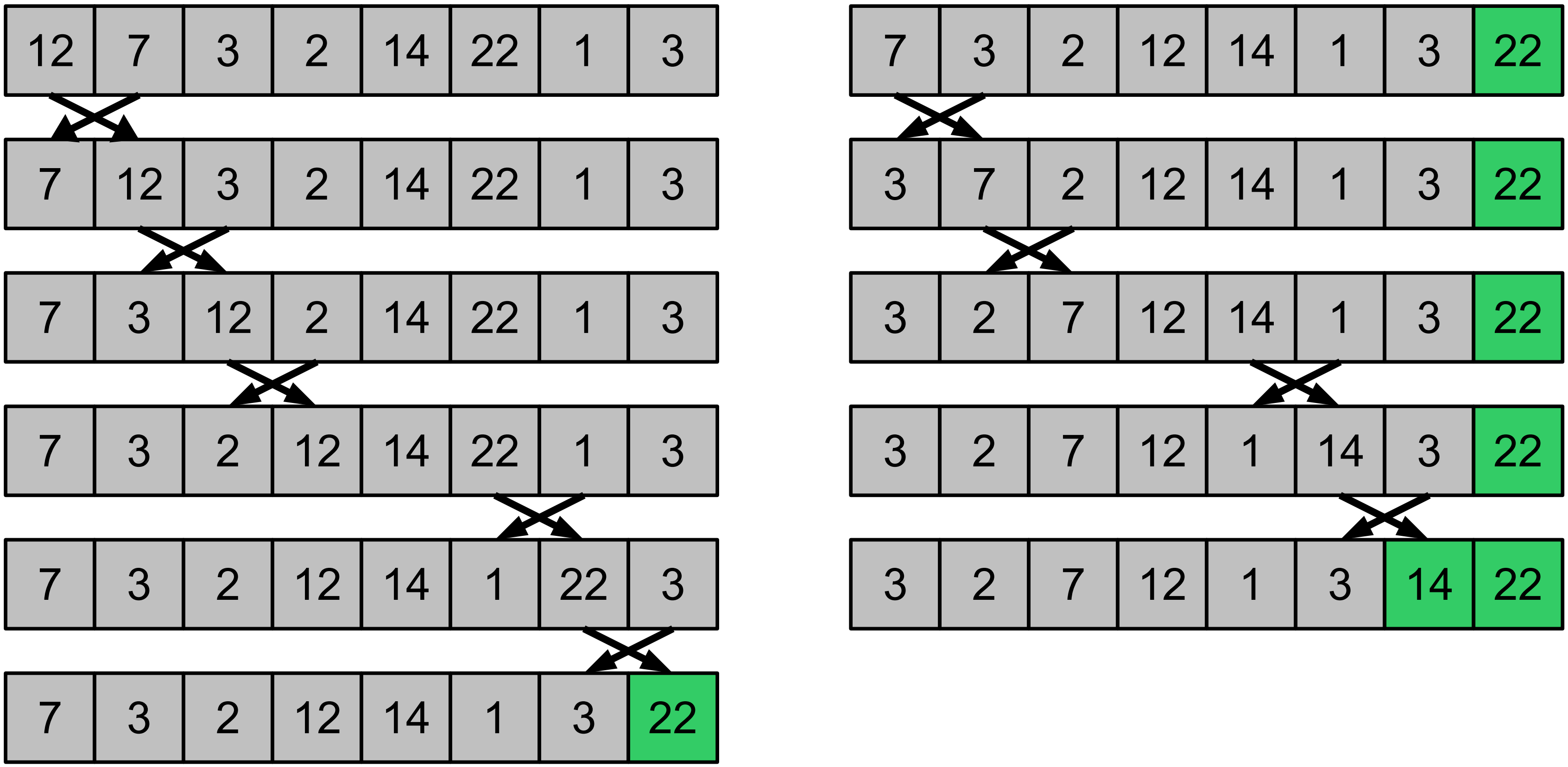
$$\sum_{k=1}^{n-1}(k+1)=\left(\sum_{k=1}^{n-1}k\right)+(n-1)=\Theta(n^2)$$

- Question: how many basic operations are executed by insertion sort algorithm in the worst case? And in the best case?

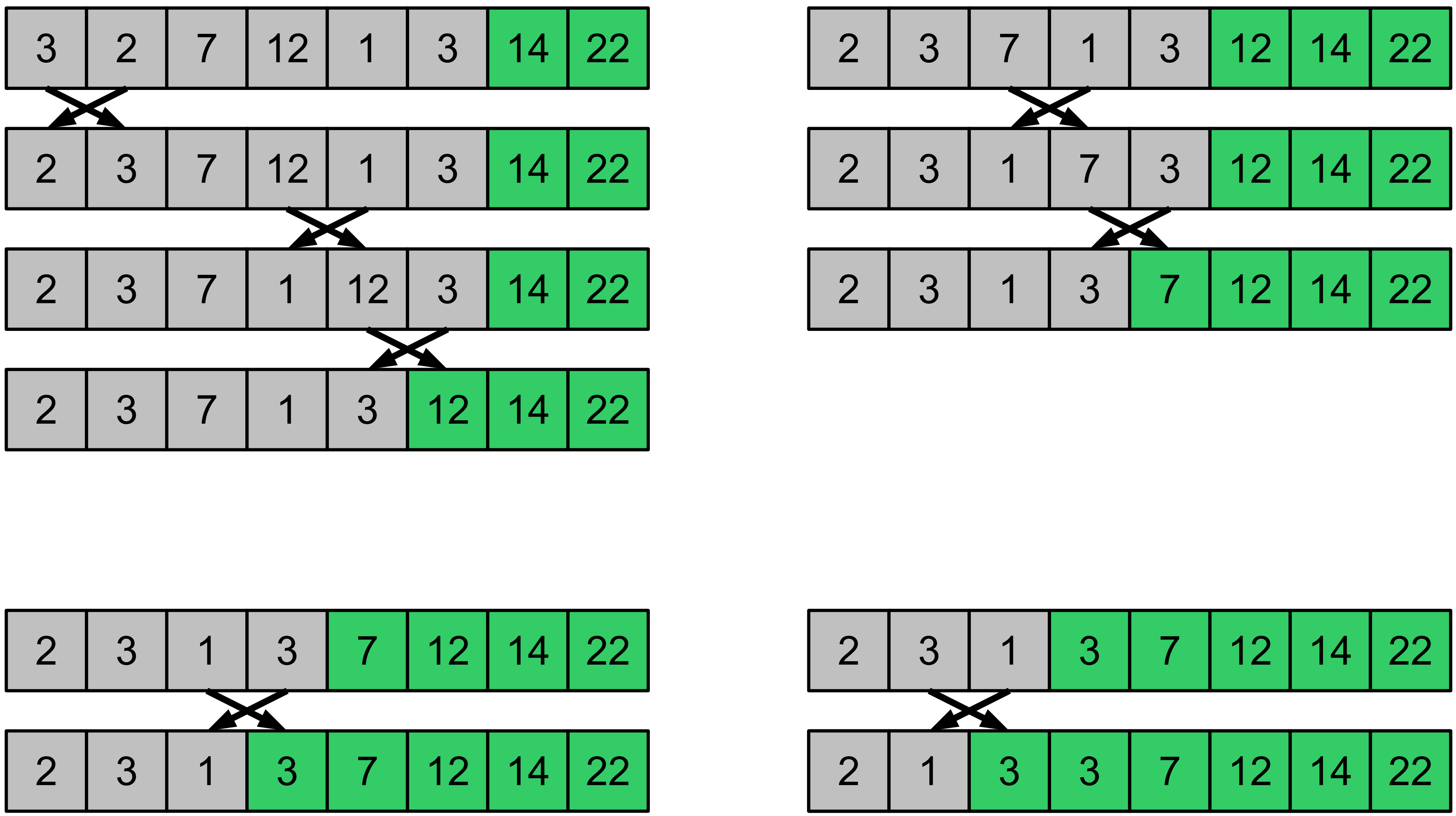
Bubble Sort

- It executes iterative scans of the array
  - In each scan it swaps the unordered pairs of values
  - It ends when at the end of a scan no swaps have been made
- After the first scan the max element occupies the last position
- After the second scan the second max element occupies the penultimate position
- ...after k scans, the k max elements are in the correct position at the end of the array

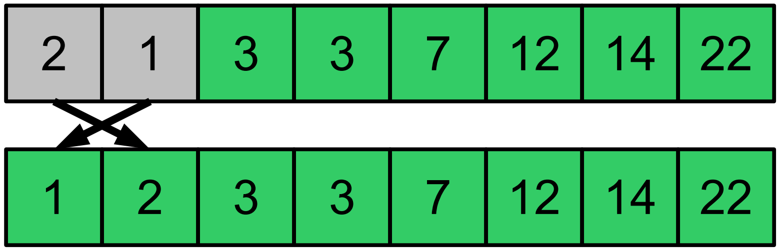
Bubble Sort


12 7 3 2 14 22 1 3
7 12 3 2 14 22 1 3
7 3 12 2 14 22 1 3
7 3 2 12 14 22 1 3
7 3 2 12 14 1 22 3
7 3 2 12 14 1 3 22
7 3 2 12 14 1 3 22
3 7 2 12 14 1 3 22
3 2 7 12 14 1 3 22
3 2 7 12 1 14 3 22
3 2 7 12 1 3 14 22

Bubble Sort


3 2 7 12 1 3 14 22
2 3 7 12 1 3 14 22
2 3 7 1 12 3 14 22
2 3 7 1 3 12 14 22
2 3 7 1 3 12 14 22
2 3 1 7 3 12 14 22
2 3 1 3 7 12 14 22
2 3 1 3 7 12 14 22
2 3 1 3 7 12 14 22
2 3 1 3 7 12 14 22
2 1 3 3 7 12 14 22

Bubble Sort



Bubble Sort

```
public static void bubbleSort(Comparable A[]) {
    for (int i = 1; i < A.length; i++) {
        boolean scambiAvvenuti = false;
        for (int j = 1; j <= A.length - i; j++) {
            // Se A[j-1] > A[j], scambiali
            if (A[j - 1].compareTo(A[j]) > 0) {
                Comparable temp = A[j - 1];
                A[j - 1] = A[j];
                A[j] = temp;
                scambiAvvenuti = true;
            }
        }
        if (!scambiAvvenuti) break;
    }
}
```

Bubble Sort
cyclic invariant

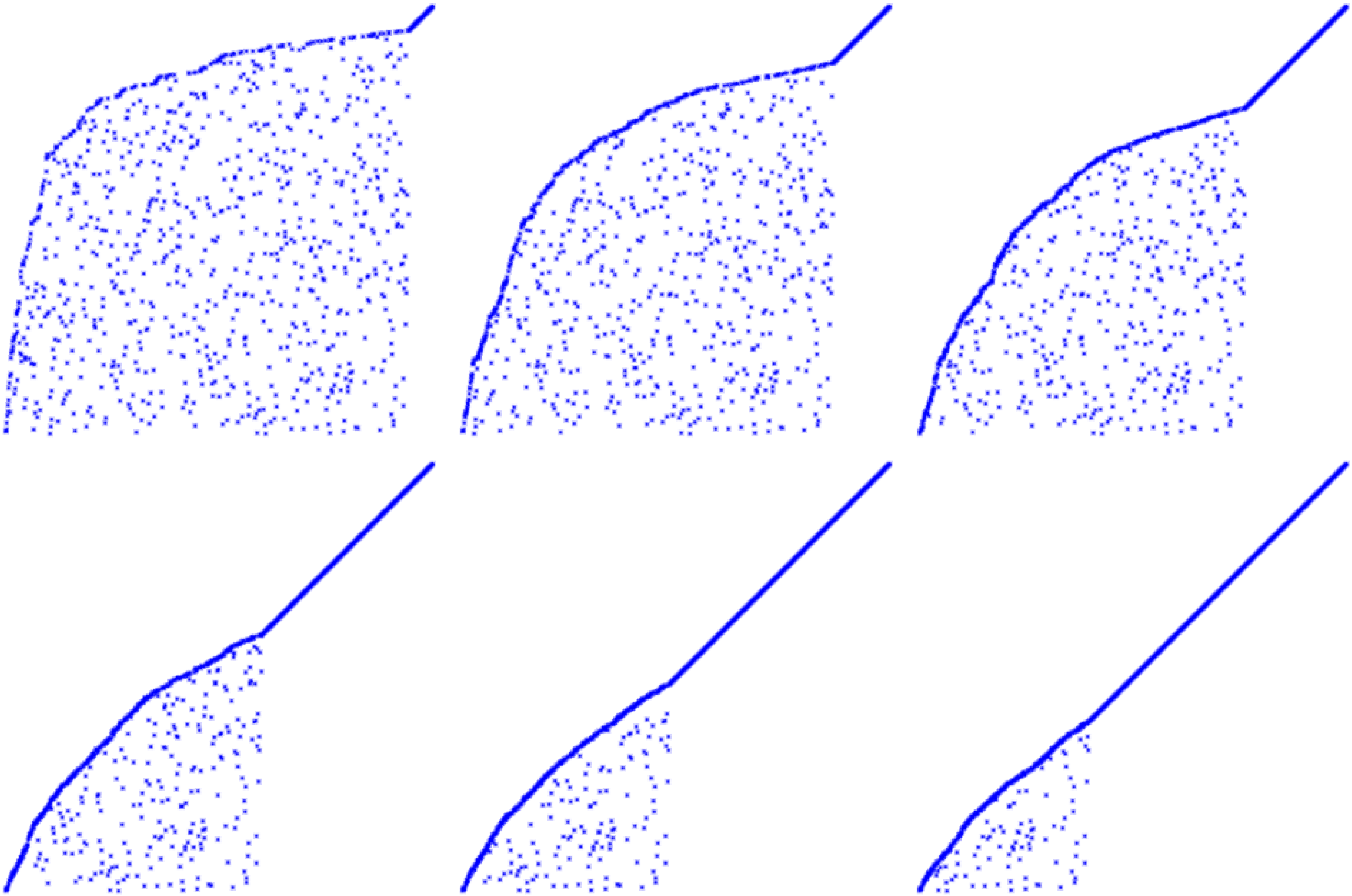
- After the i-th iteration, elements A[n-i]... A[n-1] are ordered and occupy the correct final position in the array

```
public static void bubbleSort(Comparable A[]) {
    for (int i = 1; i < A.length; i++) {
        boolean scambiAvvenuti = false;
        for (int j = 1; j <= A.length - i; j++) {
            if (A[j - 1].compareTo(A[j]) > 0) {
                Comparable temp = A[j - 1];
                A[j - 1] = A[j];
                A[j] = temp;
                scambiAvvenuti = true;
            }
        }
        if (!scambiAvvenuti) break;
    }
}
```

Bubble Sort

- Bubble Sort algorithm has $O(n^2)$ complexity
  - *In the best case the algorithm has cost O(n): only one scan of the array with no swaps.*
- In general, the algorithm has a behavior “almost natural”, meaning that the time for rodering the elements tends to be related to the initial degree of ordering of the array
  - However, how the algorithm behaves in this case? [2 3 4 5 6 7 8 9 1]

Bubble Sort image after image



Can we do better?

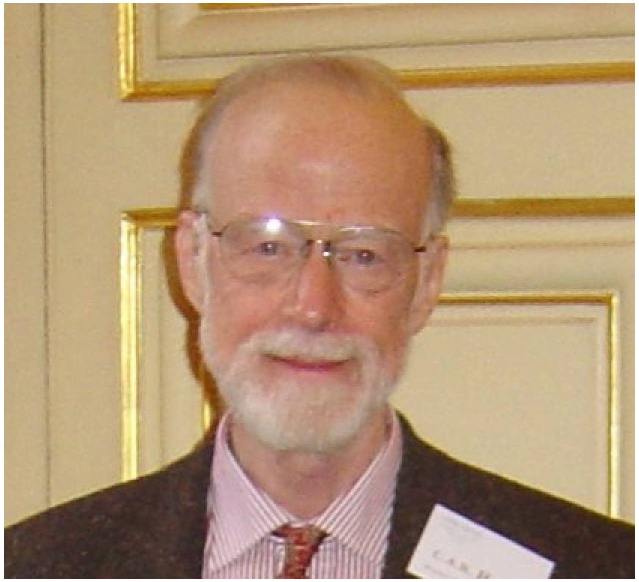
- Algorithms seen so far have cost $O(n^2)$
- Can we do better?
  - How much better?

“divide et impera” algorithms

- General idea
  - Divide: split the problem in subproblems of the same type
  - Resolve (recursively) the subproblems
  - Impera: combine the partial solutions of subproblems to get the general solution
- We will se two divide et impera sorting algorithms
  - Quick Sort
  - Merge Sort

Quick Sort

- Invented in 1962 by Sir Charles Anthony Richard Hoare
  - At that time *exchange student* at Moscow State University
  - Winner of the *Turing Award* (kind of Nobel for computer science) in 1980 for his contribution in the field of programming languages
  - Hoare, C. A. R. *"Quicksort."* Computer Journal 5 (1): 10-15. (1962).



C. A. R. Hoare (1934—)
http://en.wikipedia.org/wiki/C._A._R._Hoare

Quick Sort

- Recursive algorithm “divide et impera”
  - Choose and element x in array v, and split the array in two portions of elements ≤x and >x
  - Recursively order the two portions.
  - Return the result by creating the unique final solution
- R. Sedgewick, “*Implementing Quicksort Programs*”, Communications of the ACM, 21(10):847-857, 1978 http://portal.acm.org/citation.cfm?id=359631

Quick Sort

- Input: Array A[1..n], index i,f such that $1 \le i < f \le n$
- Divide-et-impera
  - Choose a number m in [i, i+1, ... f]
  - Divide: make a permutation of the array A[i..f] in two subarray A[i..m-1] and A[m+1..f] (maybe empty) such that:

$$\forall\ j \in [i \ldots m-1] : A[j] \le A[m]$$
$$\forall\ k \in [m+1 \ldots f] : A[m] < A[k]$$

    - A[m] is called the pivot
  - Impera: order to two subarrays A[i..m-1] and A[m+1..f] returing from quicksort recursive calls
  - Combine:do nothing;the two subarrays and element A[m] are already ordered.

Quick Sort

```
public static void quickSort(Comparable A[]) {
    quickSortRec(A, 0, A.length - 1);
}

public static void quickSortRec(Comparable[] A, int i, int f) {
    if (i >= f) return;
    int m = partition(A, i, f);
    quickSortRec(A, i, m - 1);
    quickSortRec(A, m+1, f);
}
```

Remember that in Java arrays start from 0, not from 1

Quick Sort: partition() explanation of idea

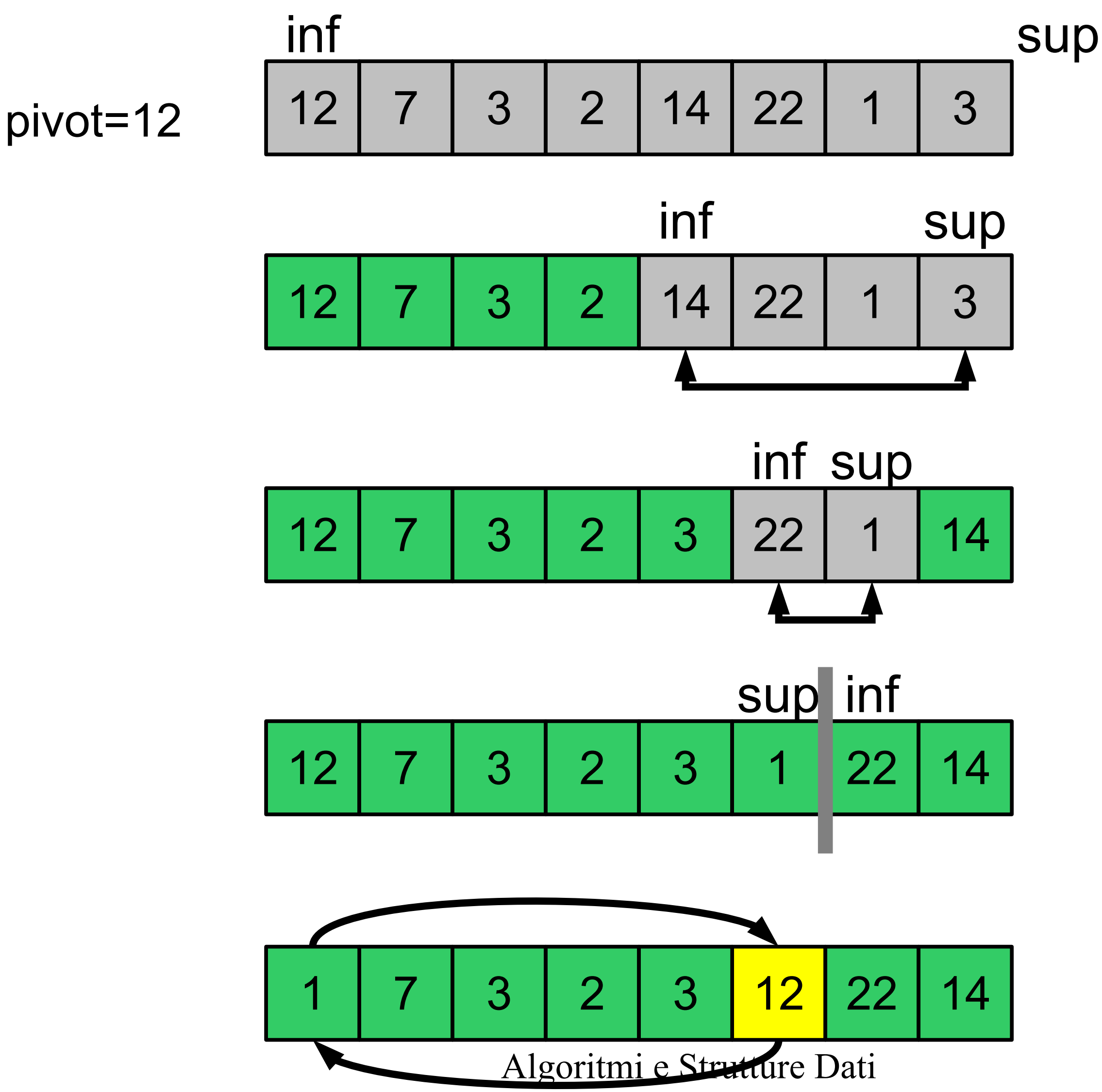
- We maintain two inicesd, inf and sup, shifted from the left and right towards the center
  - subarray A[i..inf-1] contains elements ≤ pivot
  - subarray A[sup+1..f] contains elements > pivot
- When bothi (inf e sup) cannot advance, we swap A[inf] and A[sup]

Quick Sort: partition()

```
private static int partition(Comparable A[], int i, int f) {
    int inf = i, sup = f + 1;
    Comparable temp, x = A[i];
    while (true) {
        do {
            inf++;
        } while (inf <= f && A[inf].compareTo(x) <= 0);
        do {
            sup--;
        } while (A[sup].compareTo(x) > 0);
        if (inf < sup) {
            temp = A[inf];
            A[inf] = A[sup];
            A[sup] = temp;
        } else
            break;
    }
    temp = A[i];
    A[i] = A[sup];
    A[sup] = temp;
    return sup;
}
```

deterministic selection of pivot

Example of partitioning

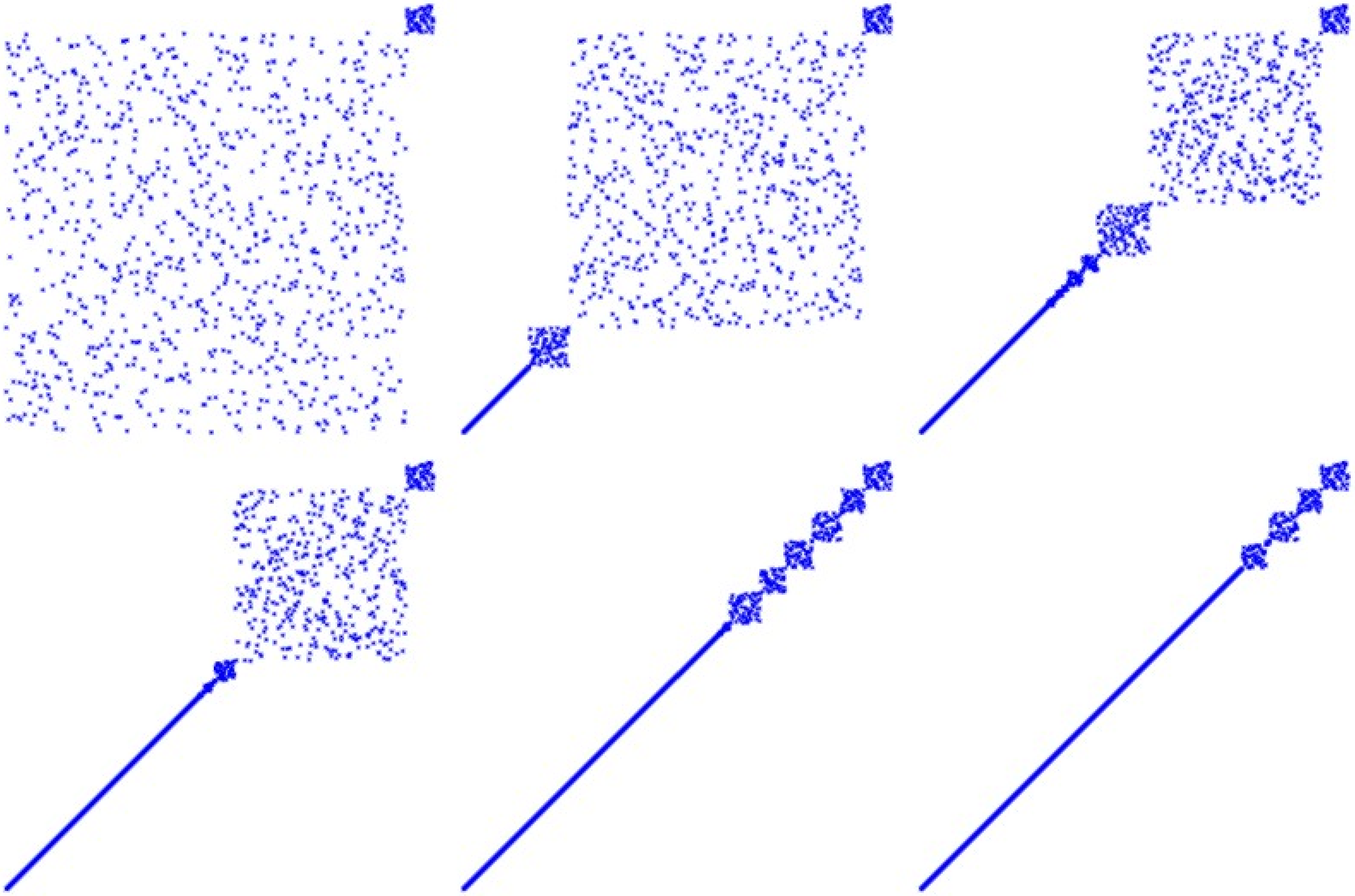


Exercise
(national flag problem)

- We have an array A[1..n] whose elements can have three values: green, white and red. We want to order the array such that all the green values are on the left, whites in center and red to the rigth.
- Algorithm must be O(n) with O(1) additional memory. We can compare and swap elements, we DO NOT use additional arrays, and we cannot use counters to store the number of elements of each colour.
- Algorithm must do a single scan of the array

This algorithm will be used in the algorithm for the selection of k-th element

Quick Sort image by image



Quick Sort: cost analysis

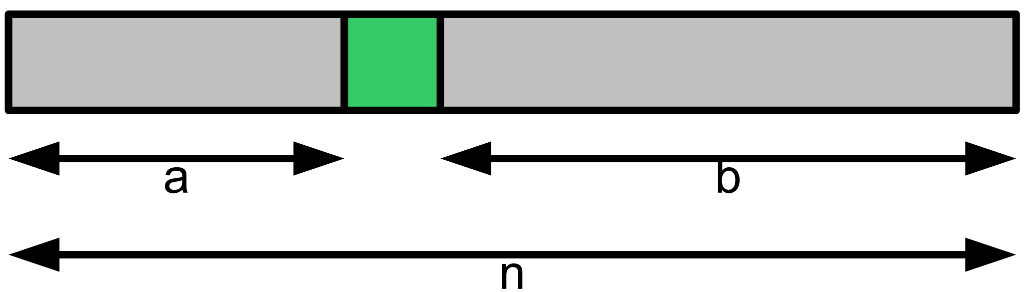
- Cost ofi partition(): $\Theta(f-i)$
- Cost of Quick Sort: Depends on partitioning
- Worst partitioning
  - When given a problem of size n, this is always divided in two subproblems of size 0 e n-1
  - $T(n) = T(n-1)+T(0)+\Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$
- **Question**: when do we have the worst case?
- Best partioning
  - When given a problem of size n, this is always divided in two subproblems of size n/2
  - $T(n) = 2T(n/2)+\Theta(n) = \Theta(n \log n)$ (case 2 Master Theorem)

QuickSort: average case analysis

- In general, we can write the recurrence equation T(n) —indicating the number of comparisons requested— as follows:

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

with $(a+b)=(n-1)$



- Unfortunately a and b could change after every iteration.

QuickSort: average case analysis

- By assuming that all the partitions are with the same probability

$$T(n)=\sum_{a=0}^{n-1}\frac{1}{n}\left(n-1+T(a)+T(n-a-1)\right)$$

- Note tht terms T(a) and T(n-a-1)produce the same summatory, hence we can simplify

$$T(n)=n-1+\frac{2}{n}\sum_{a=0}^{n-1}T(a)$$

QuickSort: average case analysis

- Theorem: the recurrence relation

$$T(n) = n - 1 + \frac{2}{n}\sum_{a=0}^{n-1} T(a)$$

has solution $T(n) \le 2n \log n$

- Proof: we verify by substitution that T(n) satisfies the relation $T(n) \le \alpha\, n \log n$
  - We will see we will obtain $\alpha=2$

QuickSort: average case analysis

$$T(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$$

$$\leq n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} \alpha\, i \log i$$

$$= n - 1 + \frac{2\alpha}{n}\sum_{i=2}^{n-1} i \log i$$

$$\leq n - 1 + \frac{2\alpha}{n}\int_{2}^{n} x \log x \, dx$$

continua...

QuickSort: average case analysis

$$\int f(x)g'(x)\,dx = f(x)g(x) - \int f'(x)g(x)\,dx$$

$$T(n) \leq n - 1 + \frac{2\alpha}{n}\int_2^n x\log x\,dx$$

$$= n - 1 + \frac{2\alpha}{n}\left(\frac{n^2\log n}{2} - \frac{n^2}{4} - 2\log 2 + 1\right)$$

$$= n - 1 + \alpha n\log n - \alpha\frac{n}{2} - O(1)$$

$$\leq \alpha n\log n$$

- Last inequality holds for $\alpha \geq 2$ (and for large values of n), and this prove the theorem.

Quick Sort: randomized version

- Choice of pivot into partition() is crucial to avoid the worst case execution
- We have seen an implementation where the pivot is always selected as the first element of the subarray.
  - Inthis case it is easy to define examples providing the worst case execution
- We can reduce the probability of occurrence of the worst case by adopting a randomization of the pivot
  - We select in a pseudo-random way the pivot into the subarray

Quick Sort: partition()

randomized version

```
private static int partition(Comparable A[], int i, int f) {
    int inf = i, sup = f + 1,
        pos = i + (int) Math.floor((f-i+1) * Math.random());
    Comparable temp, x = A[pos];
    A[pos] = A[i];
    A[i] = x;
    while (true) {
        do {
            inf++;
        } while (inf <= f && A[inf].compareTo(x) <= 0);
        do {
            sup--;
        } while (A[sup].compareTo(x) > 0);
        if (inf < sup) {
            temp = A[inf];
            A[inf] = A[sup];
            A[sup] = temp;
        } else
            break;
    }
    temp = A[i];
    A[i] = A[sup];
    A[sup] = temp;
    return sup;
}
```

pseudorandom selection of pivot

Merge Sort

- Invented by John von Neumann in 1945
- Algorithm *divide et impera*
- Idea:
  - Divide A[] in 2 equal size halves A1[] e A2[] (without permutation);
  - Recursively call Merge Sort on A1[] and A2[]
  - Merge the ordered arrays A1[] and A2[] to obtain the ordered A[].

John von Neumann (1903—1957)
http://en.wikipedia.org/wiki/John_von_Neumann

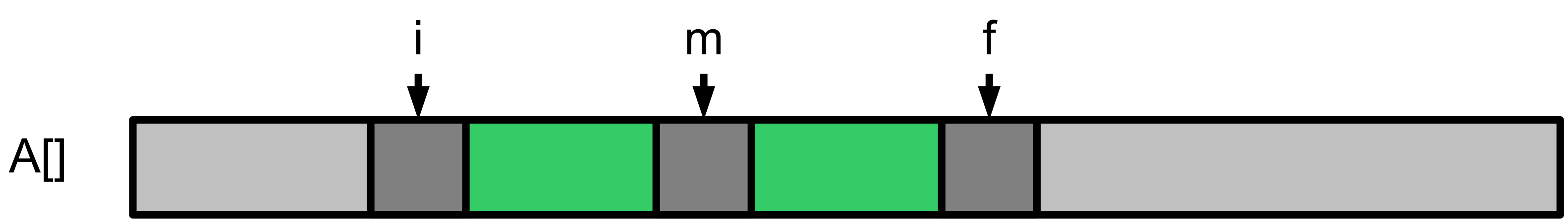
Merge Sort vs Quick Sort

- Quick Sort:
  - Compelx partitioning, trivial merge (in fact no merge operation is needed)
- Merge Sort:
  - Trivial partitioning,merge operation complex

Merge Sort

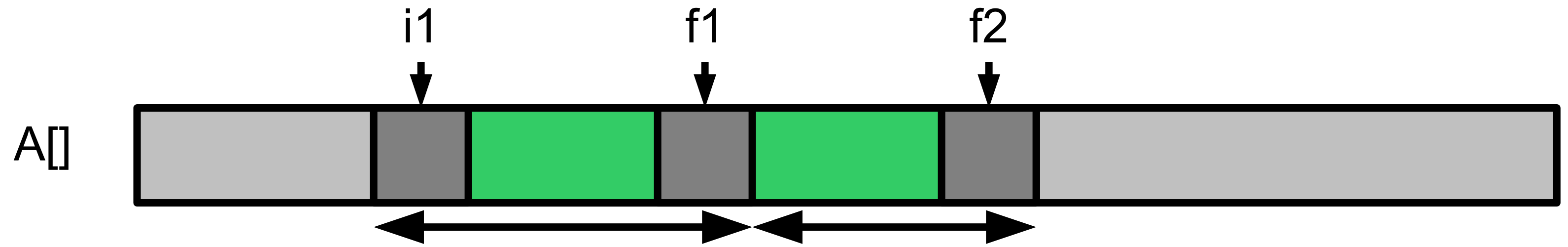
```
public static void mergeSort(Comparable A[]) {
    mergeSortRec(A, 0, A.length - 1);
}

private static void mergeSortRec(Comparable A[], int i, int f) {
    if (i >= f) return;
    int m = (i + f) / 2;
    mergeSortRec(A, i, m);
    mergeSortRec(A, m + 1, f);
    merge(A, i, m, f);
}
```

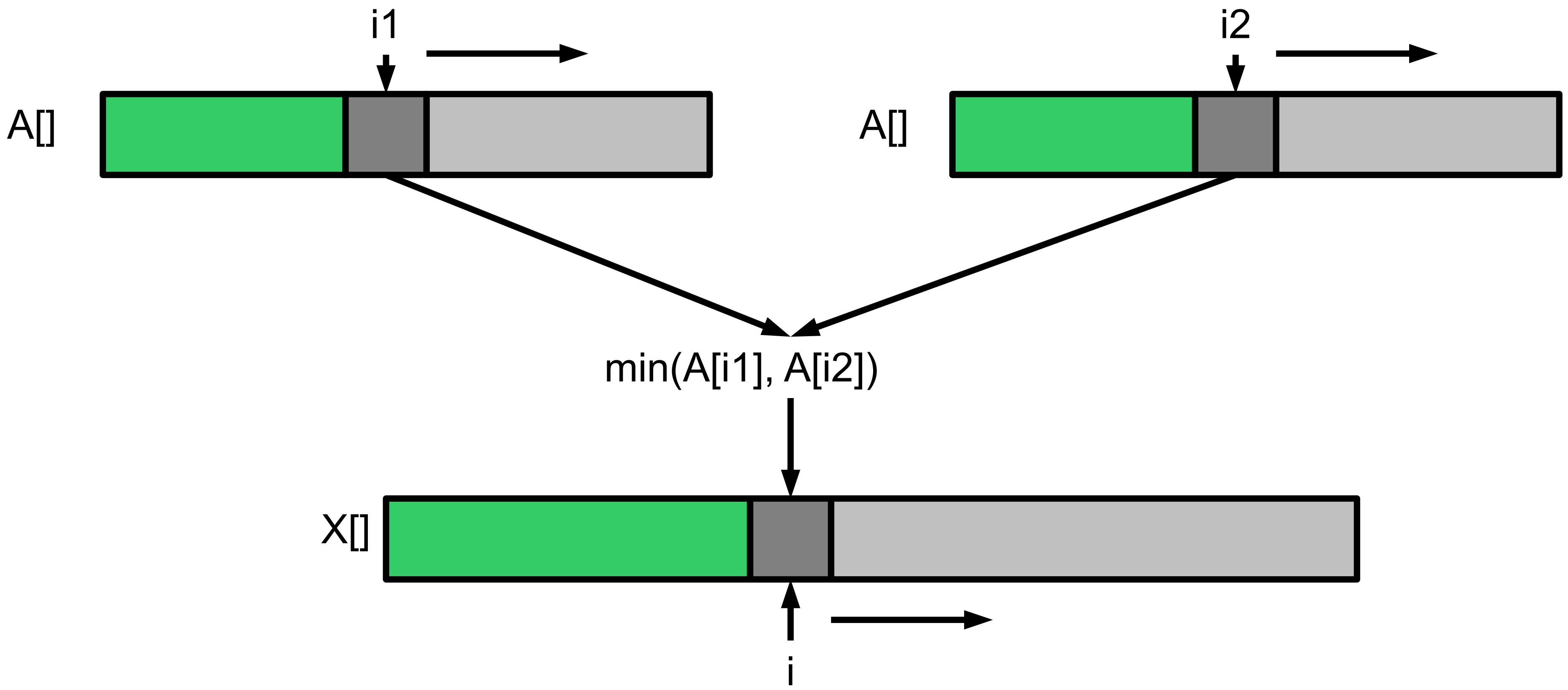


Operation merge()

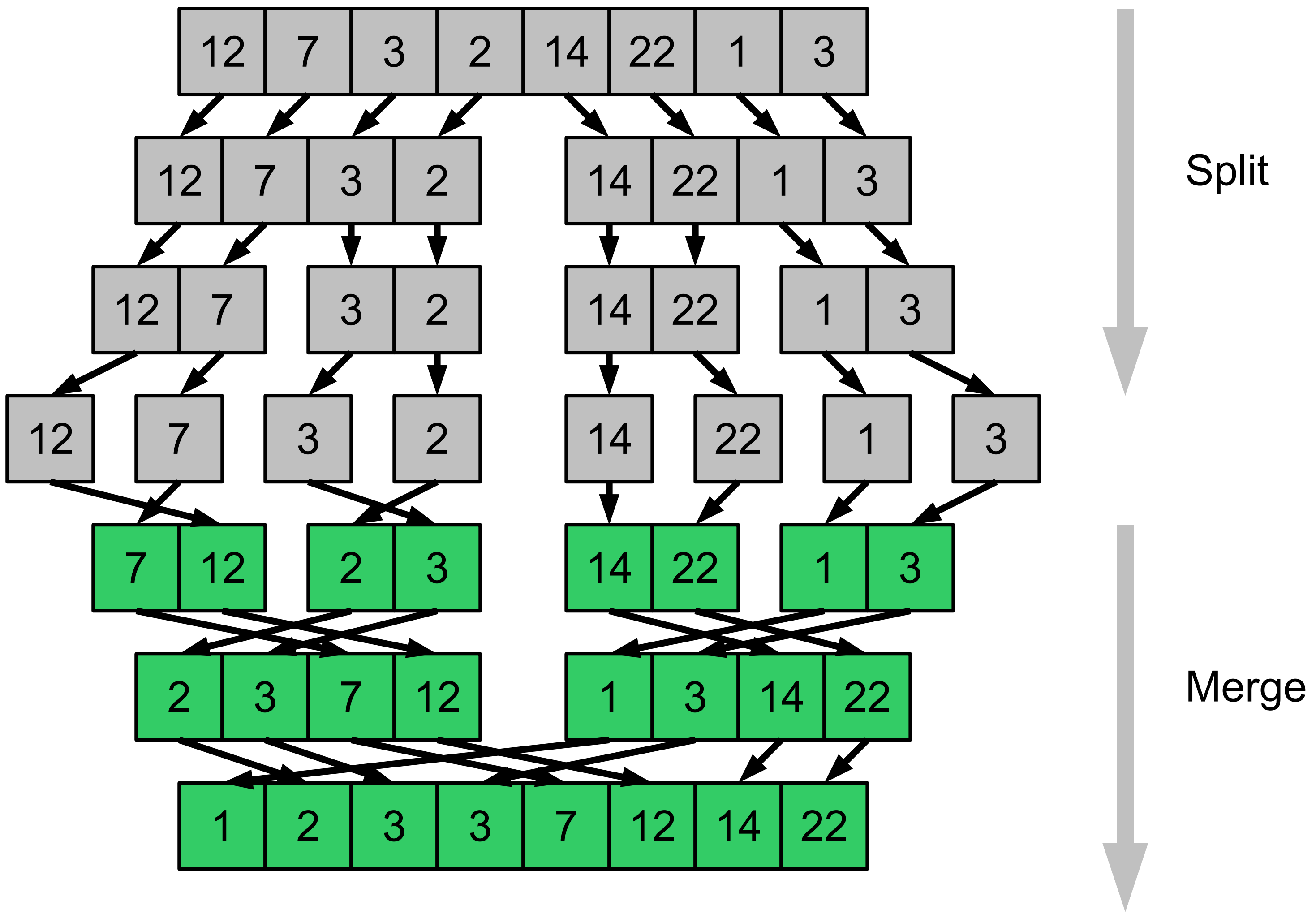
```
private static void merge(Comparable A[], int i1, int f1, int f2)
{
    Comparable[] X = new Comparable[f2 - i1 + 1];
    int i = 0, i2 = f1 + 1, k = i1;
    while (i1 <= f1 && i2 <= f2) {
        if (A[i1].compareTo(A[i2]) < 0)
            X[i++] = A[i1++];
        else
            X[i++] = A[i2++];
    }
    if (i1 <= f1)
        for (int j = i1; j <= f1; j++, i++) X[i] = A[j];
    else
        for (int j = i2; j <= f2; j++, i++) X[i] = A[j];
    for (int t = 0; k <= f2; k++, t++) A[k] = X[t];
}
```



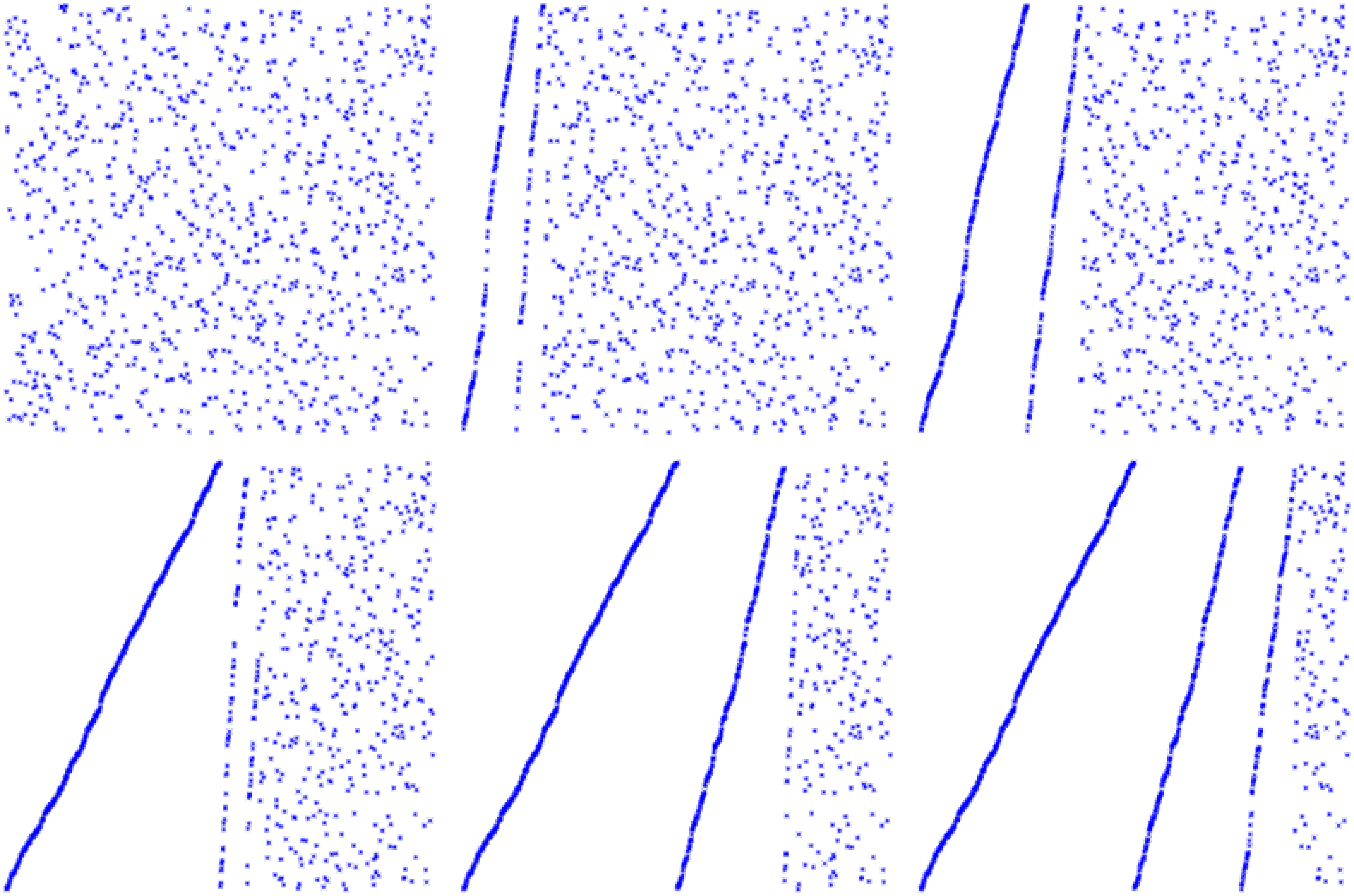
Operation merge()



Merge Sort: example



Merge Sort image by image



Merge Sort: complexity

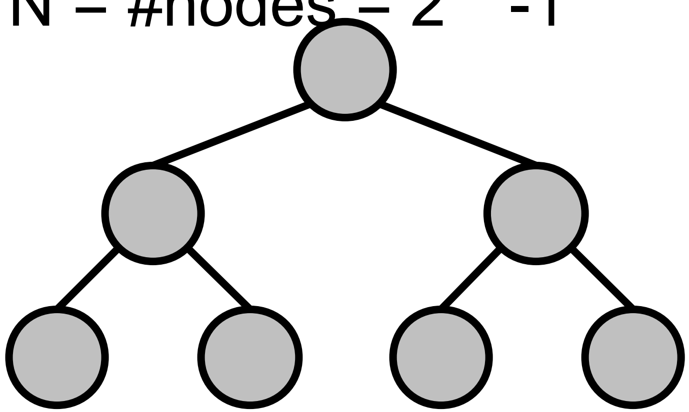
- $T(n) = 2T(n/2) + \Theta(n)$
- Given the Master Theorem (case 2), we get
  $T(n) = \Theta(n \log n)$
- Merge Sort complexity does not depend on the initial ordering of the array
  - Hence the above limit holds in the best/average/worst case
- Disadvantage w.r.t Quick Sort: Merge Sort needs more memory space (not ordering on place).
  - Jyrki Katajainen, Tomi Pasanen, Jukka Teuhola, "*Practical in-place mergesort*", http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.8523

Heapsort

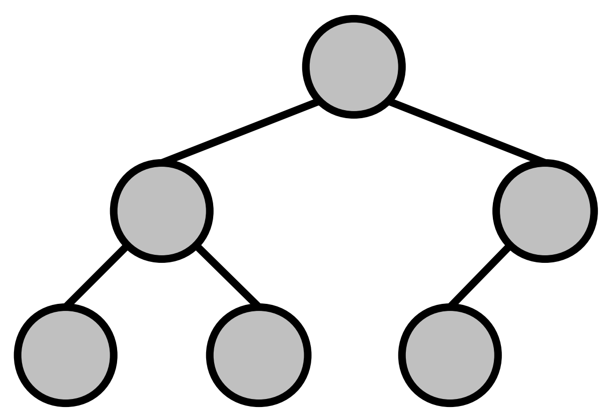
- idea
  - We use a data structure—called heap—for ordering the array
  - Computational Cost: O(n log n)
  - Ordering on place
- Moreover
  - The heap concept can be used to implement priority queues

Binary tree

- Perfect binary tree
  - All leaves have same heigth h
  - Internal nodes have degree 2
- A perfect tree
  - Has heigth h ≈ log N
  - N = #nodes = $2^{h+1}-1$



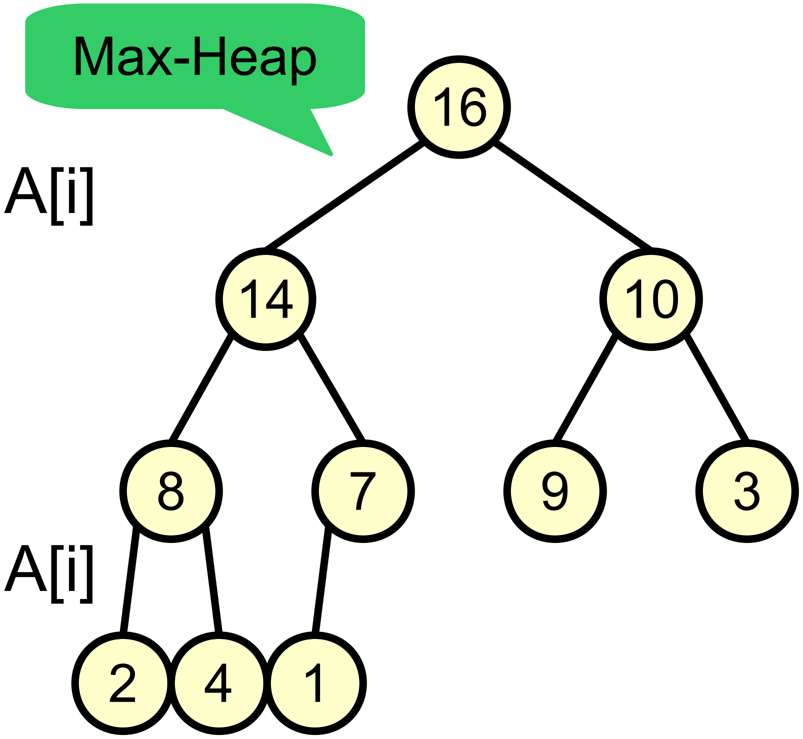
- Complete binary tree
  - All leaves have heigth h or h-1
  - All the nodes at level h are accumulated from the left
  - All internal nodes have degree 2 (maybe but one)



Heap binary tree

- A complete binary tree is a

max-heap iff
  - Every node i has associated value A[i]
  - A[Parent(i)] ≥ A[i]



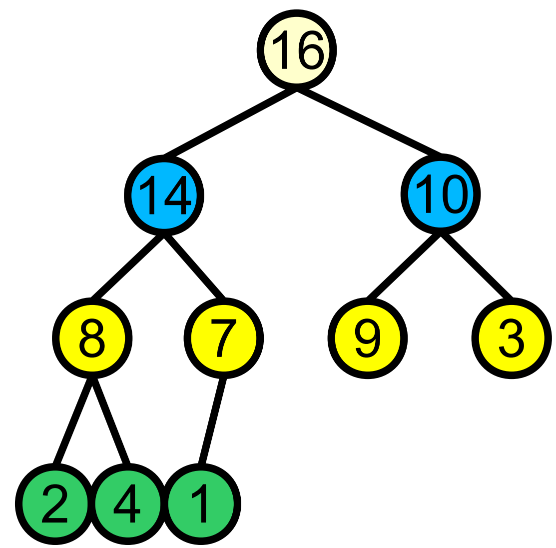
- Un complete binary tree is

a min-heap iff
  - Every node i has associated value A[i]
  - A[Parent(i)] ≤ A[i]

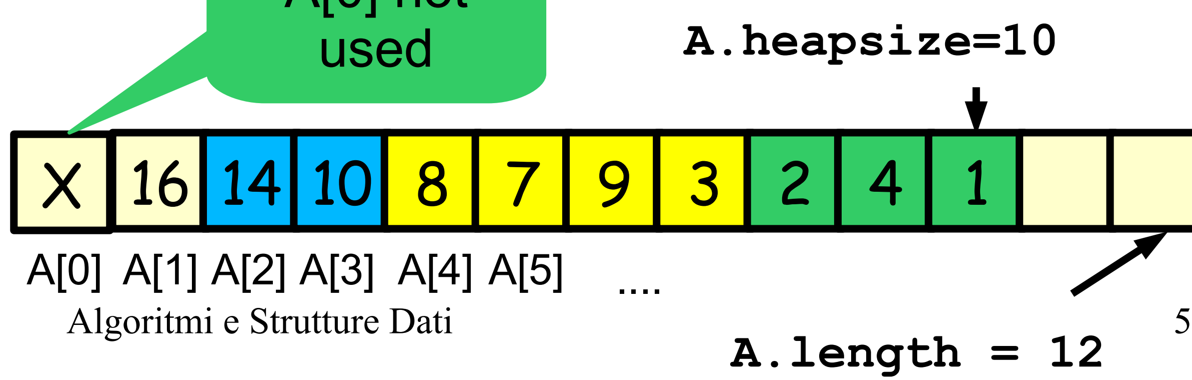
- Of course the definition and the algorithms max-heap and min-heap are similar.

Heap Array

- Is it possible to represent a heap binary treewith a heap array (with pointers)
- What does it contain?
  - Array A, size A.length
  - size A.heapsize ≤ A.length
- How does it work?
  - A[1] contains the root
  - Parent(i) = Math.floor(i/2)
  - Left(i) = 2*i
  - Right(i) = 2*i+1



A[0] not used



Question: elements of the heap array appear as in the visit of the tree ...

Heap array operations

- findMax(): find the max value contained in a heap
  - Max value always in the root, that is A[1]
  - cost O(1)
- fixHeap(): restore max-heap property
  - If we replace the root of A[1] in a max-heap with an arbitrary value
  - We want A[] becomes again a heap
- heapify(): to create a heap starting from a generic unordered array
- deleteMax(): deletes the max element from max-heap A[]

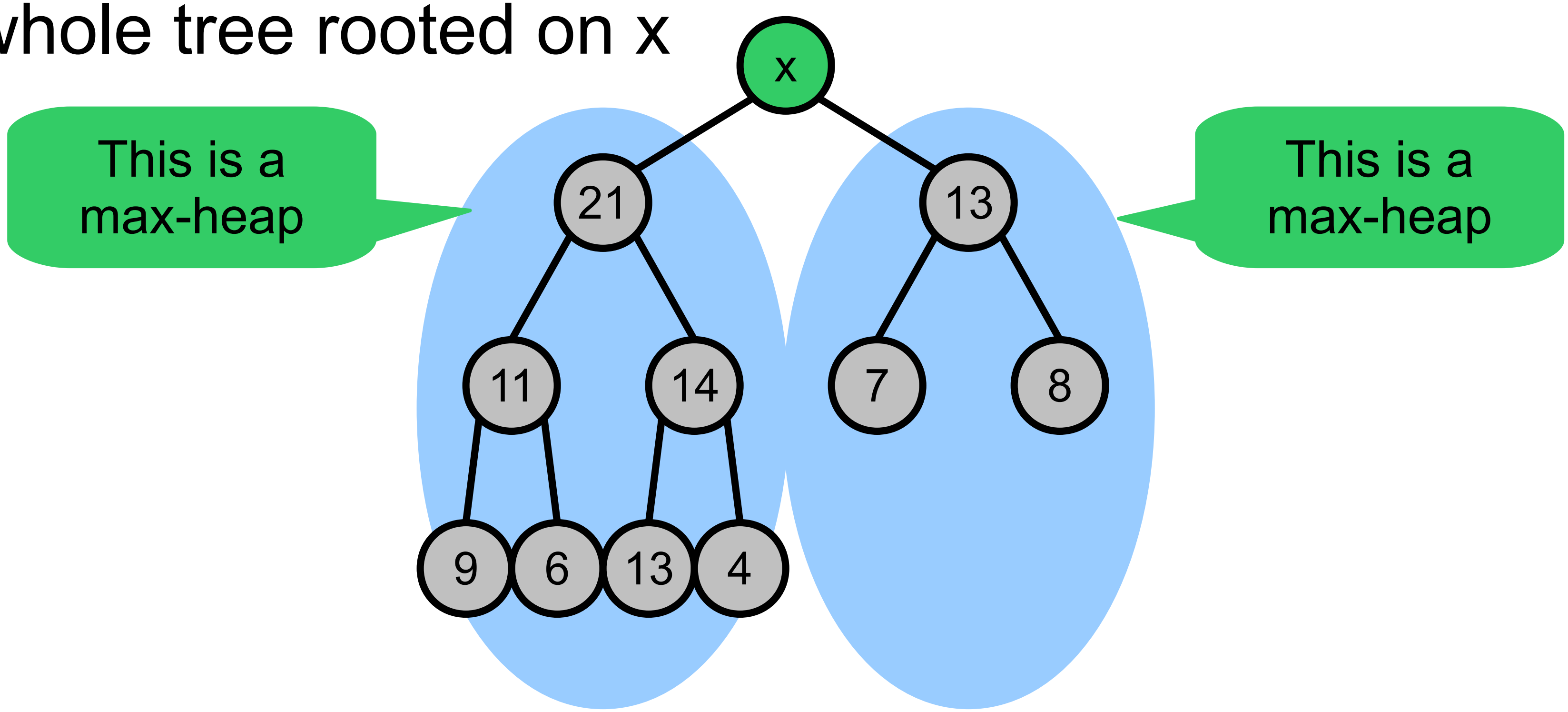
heapify()

- Parameters:
  - S[] is an arbitrary array; we assume the heap has elements S[1], ... S[n] (S[0] not used)
  - i is the index of element that will become the root of the heap (i≥1)
  - n is the index of last element of the heap

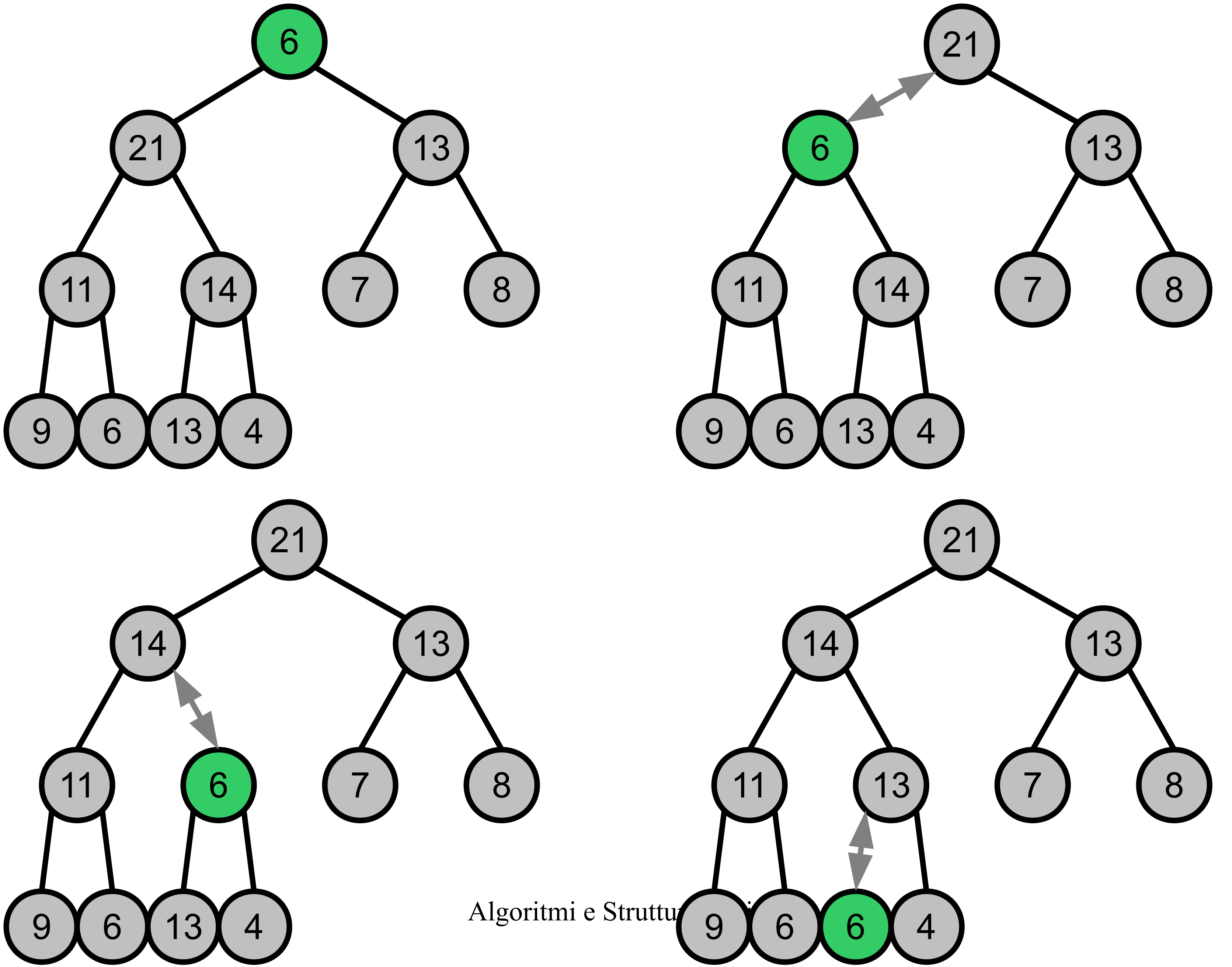
```
private static void heapify(Comparable S[], int n, int i) {
    if (i > n) return;
    heapify(S, n, 2 * i); // create heap rooted in S[2*i]
    heapify(S, n, 2 * i + 1); // create heap rooted in S[2*i+1]
    fixHeap(S, n, i);
}
// transform array S into a heap:
// heapify(S, S.length, 1 );
```

fixHeap()

- Imagine to transform in max-heaps the two left and irgth subtree of a node x
- operation fixHeap()will transform into a max-heap the whole tree rooted on x



Operation fixHeap()



Operation fixHeap()

- Restores the ordering property of a max-heap w.r.t. Root node with index i.
- Recursively compare S[i] against the max of the child nodes and swap each time the order property is not satisfied

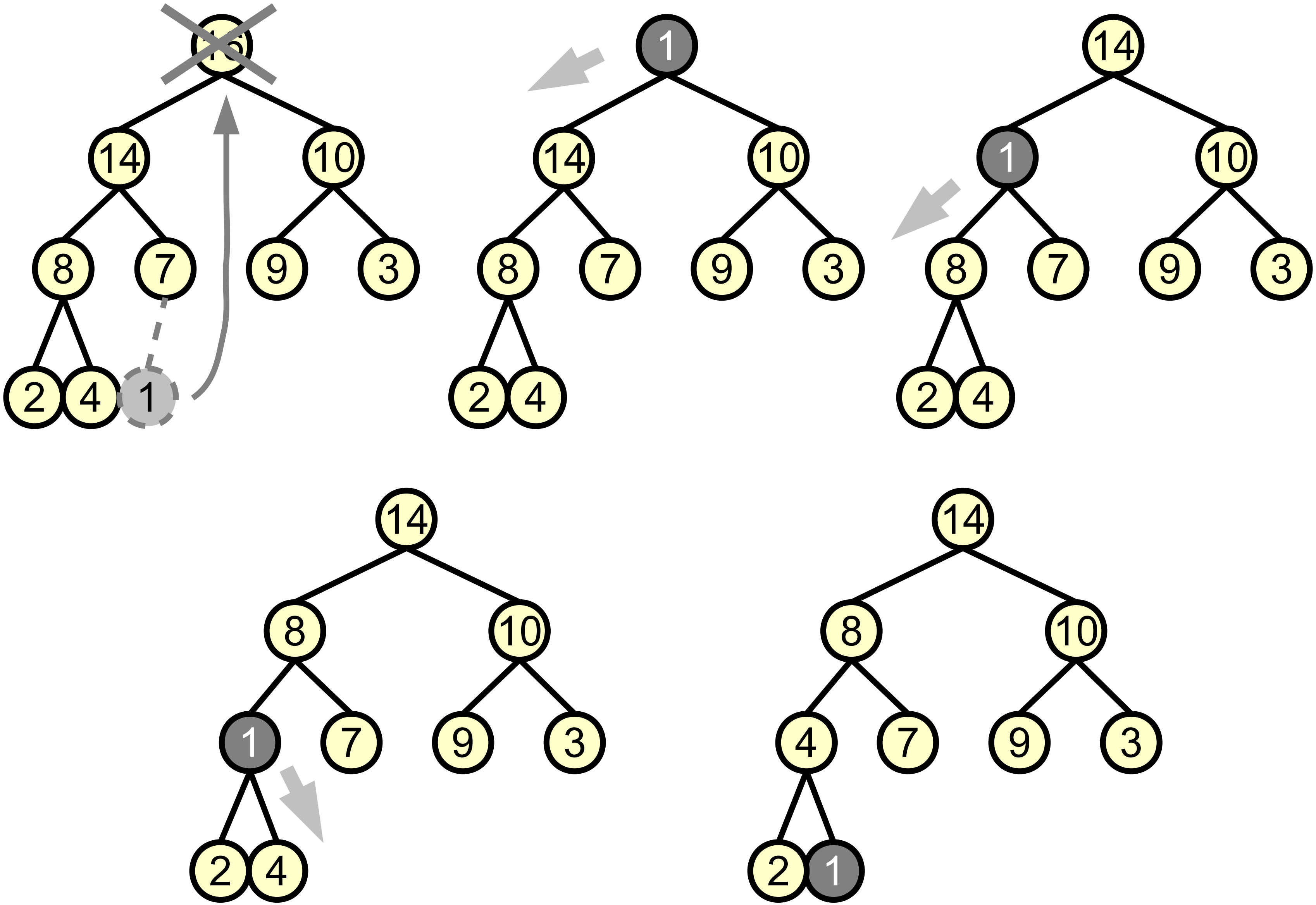
```
private static void fixHeap(Comparable S[], int c, int i) {
    int max = 2 * i; // left child
    if (2 * i > c) return;
    if (2 * i + 1 <= c && S[2 * i].compareTo(S[2 * i + 1]) < 0)
        max = 2 * i + 1; // right child
    if (S[i].compareTo(S[max]) < 0) {
        Comparable temp = S[max];
        S[max] = S[i];
        S[i] = temp;
        fixHeap(S, c, max);
    }
}
```

C is the index of last element of the heap

DeleteMax() operation

- Aim: remove root (i.e. Max value) from the heap, by mintaining the max-heap property.
- Idea
  - In the place of the old value A[1] we put the calue of the last position in the heap array.
  - apply fixHeap() to restore the heap property

Example



14 10 8 7 9 3 2 4 1
1 14 10 8 7 9 3 2 4
14 1 10 8 7 9 3 2 4
14 8 10 1 7 9 3 2 4
14 8 10 4 7 9 3 2 1

Computational cost

- fixHeap()
  - In the worst case the cost is equal to the heigth of the heap
  - O(log n)
- heapify()
  - $T(n) = 2T(n/2) + O(\log n)$
  - So, $T(n) = O(n)$ (case (1) of Master Theorem)
- findMax()
  - O(1)
- deleteMax()
  - Same as fixHeap(), that is O(log n)

Heap Sort

- Idea:
  1. Build a max-heap starting from original array A[] by exploiting the operation heapify()
  2. Extract the maximum (findMax() + deleteMax())
     - Heap shortens one element
  3. Insert the max into last position of A[]
  4. Repeat from 2. as far as heap is empty

Heap Sort

```
public static void heapSort(Comparable S[]) {
    heapify(S, S.length - 1, 1);
    for (int c = (S.length - 1); c > 0; c--) {
        Comparable k = findMax(S);
        deleteMax(S, c);
        S[c] = k;
    }
}
```

O(n)

O(1)

O(log n)

Remember that elements to sort are S[1], ... S[n]

- Computational cost:
  - O(n) for initial heapify()
  - For each 'for' cycle the cost is O(log c)
- Total:

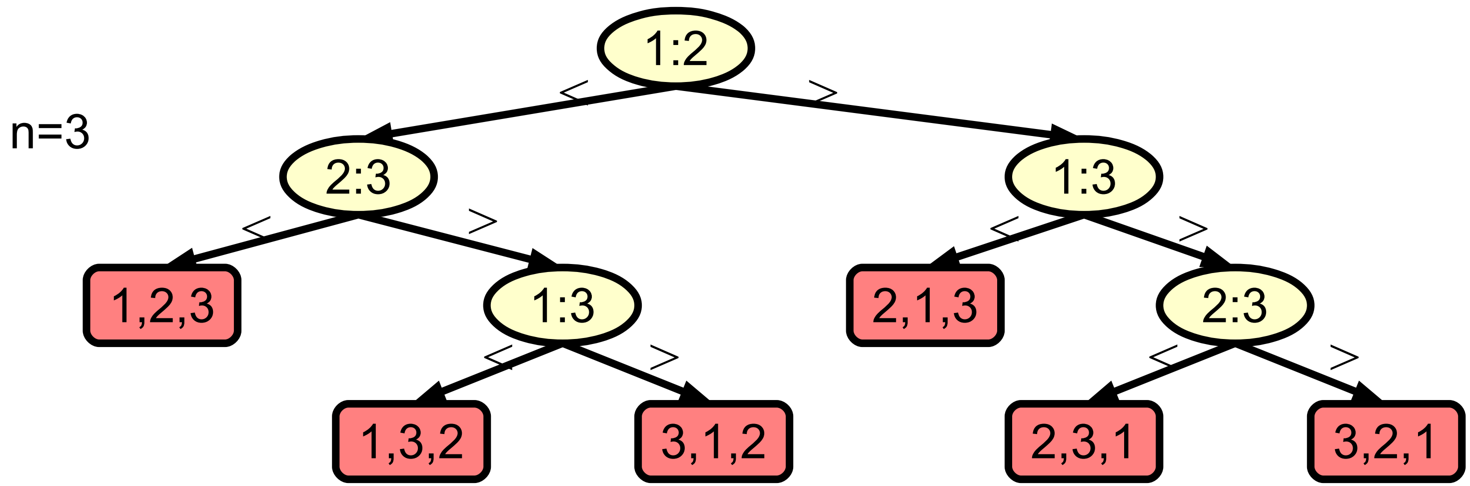
$$T(n) = O(n) + O\left(\sum_{c=n}^{1} \log c\right) = O(n \log n)$$

Sorting algorithms: summary

- We have seen different sorting algorithms:
  - Selection Sort: best/average/worst $\Theta(n^2)$
  - Insertion Sort: best/average/worst $\Theta(n^2)$ (question: how to modify it to have $\Theta(n)$ as best case?)
  - Merge Sort: best/average/worst $\Theta(n \log n)$
  - Heap Sort: $O(n \log n)$
  - Quick Sort: best/average $O(n \log n)$, worst $O(n^2)$
- Note:
  - All these algorithms are based on comparisons
    - Decisions for ordering are taken based on comparison (<,=,>) between two values
- Question
  - Is it possible to have better complexity than $O(n \log n)$?

Lower limit to complexity of a sorting algorithm

- Assumption
  - Let's consider a given algorithm X based on comparison
  - Let's assume all the elements are different values
- algorithm X
  - Can be always represented as a decision tree, which is a binary tree representing all the comparisons made.



Lower limit to complexity of a sorting algorithm

- Idea
  - Every sorting algorithm based on comparisons can be always represented with a decision tree
  - Every decision tree can be thought as an ordering algorithm
- Property
  - Path root-leaf in a decision tree:
    sequence of comparisons executed by corresponding alg.
  - Heigth of a decision tree:
    # of comparisons of the given algorithm in the worst case
  - Average Heigth of a decision tree:
    # of comparisons of the given algorithm in the average case

Lower limit to complexity of a sorting algorithm

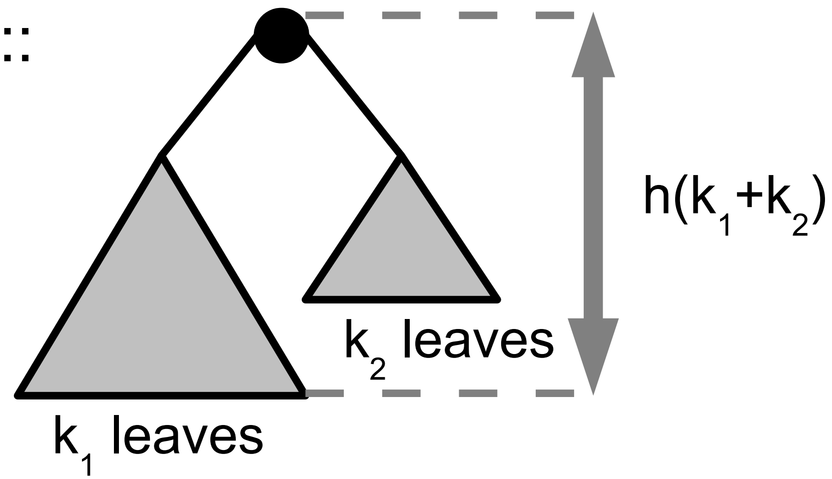
- Lemma 1
  - A decision tree for the sorting of n elements contains at least n! leaves.
- proof
  - Every leaf corresponds to a given solution for the sorting problem
  - A solution for the sorting algorithm consists into a permutation of the input values
  - There are n! possible permutations

Lower limit to complexity of a sorting algorithm

- Lemma 2
  - let T be a binary tree in which every internal node has 2 childs and let k be the number of leaves. The heigth of the tree is at least $log_2 k$
- Proof (by induction on the structure)
  - Consider a tree with only one node::
    $h(1) = 0 \geq \log_2 1 = 0$
  - Inductive step

$$
\begin{aligned}
h(k_1+k_2) &= 1 + \max\{ h(k_1), h(k_2) \} \\
&\geq 1 + h(k_1) \\
&\geq 1 + \log_2 k_1 \text{ (induction)} \\
&= \log_2 2 + \log_2 k_1 = \log_2 (2k_1) \geq \log_2 (k_1 + k_2)
\end{aligned}
$$

assume $k_1 > k_2$



Lower limit to complexity of a sorting algorithm

- Theorem
  - Numer of comparisons neede to sort n elements is *Ω(n log n)*
  - **Question:** prove it.
  - **hints:**
    - Time of execution of a sorting algorithm with n elements based on comparison is proportional to heigth of the corresponding decision tree
    - A decision tree has n! leaves
    - A decision tree with n! Leaves has heigth Ω(log n!)
    - Using the approximated Stirling formula for n!:

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

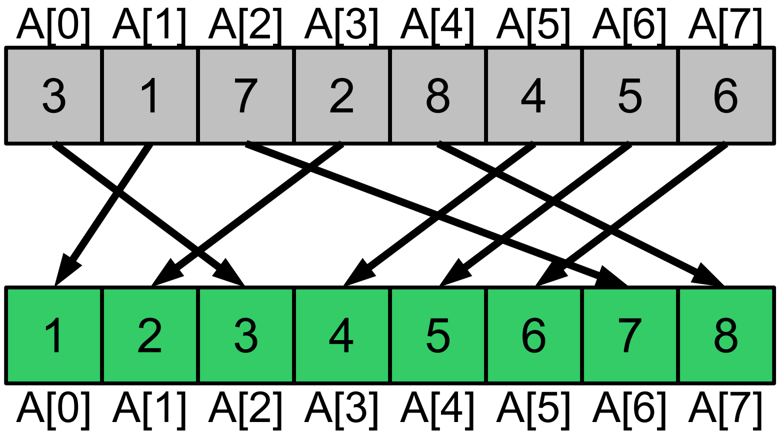
Sorting in linear time!

Linear sorting techniques

- consideration
  - The lower limit of complexity for sorting algorithms applies only on comparison-based algorithms
- Other solutions
  - Counting Sort
  - Bucket Sort
  - Radix Sort

Counting Sort
trivial case

- Given the array A[0, n-1] of n integer values, whose elements are all the values between 1 and n, and every value appears exactly once.
- Which is the correct position for element A[i] in the final ordered array?
  - Of course (A[i]-1)



Counting Sort
less trivial case

- Values in A[0..n-1] belong to interval [0, k-1] (each value can appear zero or more than one times)
  - Build an array Y[0, k-1]; Y[i] counts the number of times value i appears in A[]
  - The we reallocate all the values in A as follows:

```
public static void countingSort(int[] A, int k) {
    int[] Y = new int[k];
    int j = 0;
    for (int i = 0; i < k; i++) Y[i] = 0;
    for (int i = 0; i < A.length; i++) Y[A[i]]++;
    for (int i = 0; i < k; i++) {
        while (Y[i] > 0) {
            A[j] = i;
            j++;
            Y[i]--;
        }
    }
}
```

Couting Sort: Cost

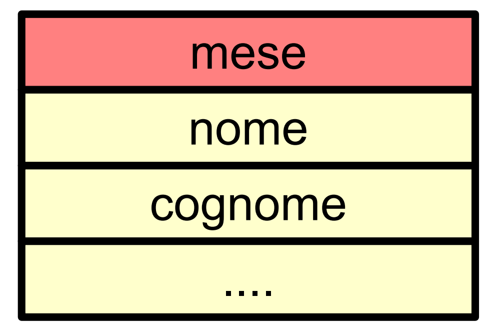
- $O(\max\{n,k\}) = O(n+k)$
- If $k=\Theta(n)$, then the cost is $O(n)$

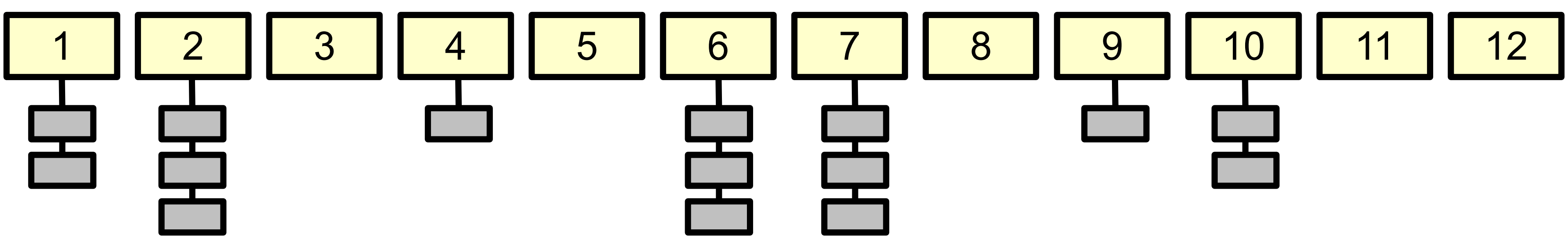
“Pigeonhole Sort” (Bucket Sort)

Pigeon tower
http://www.prolocosalento.it/allistefelline/main.shtml?A=f_alliste

Bucket Sort

- Bucket Sort
  - Imagine the values to order are not integer but values of records associated to a given key.
  - We cannot use counting sort
  - But we can use concatenated lists





Bucket Sort

- sorting n records whose integer keys are in [1,k]

```
Algoritmo bucketSort(array X[1..n], integer k)
   let Y be an array of size k
   for i := 1 to k do
       Y[i]:=empty list
   endfor
   for i := 1 to n do
       Append X[i] to list Y[key(X[i])];
   endfor
   for i := 1 to k do
       Copy in X elements in Y[i]
   endfor
```

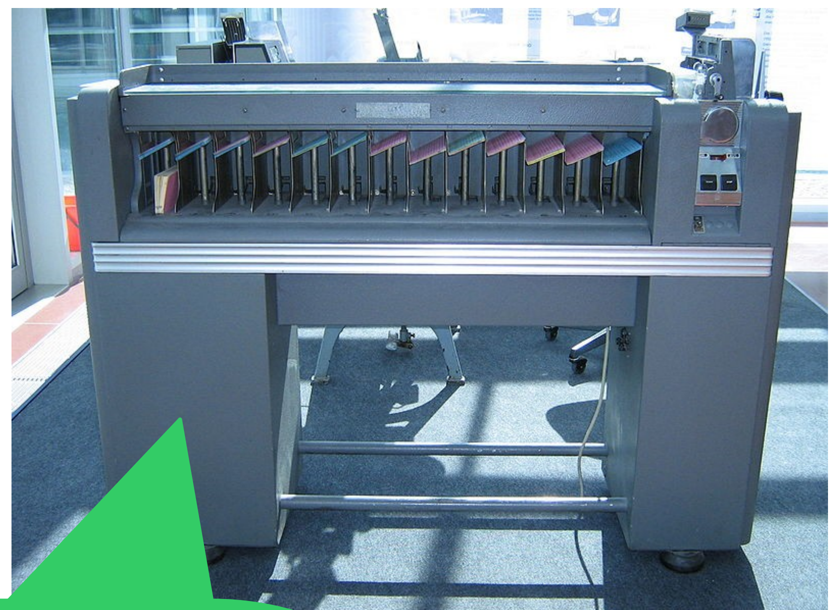
- Cost: O(n+k)

Radix Sort

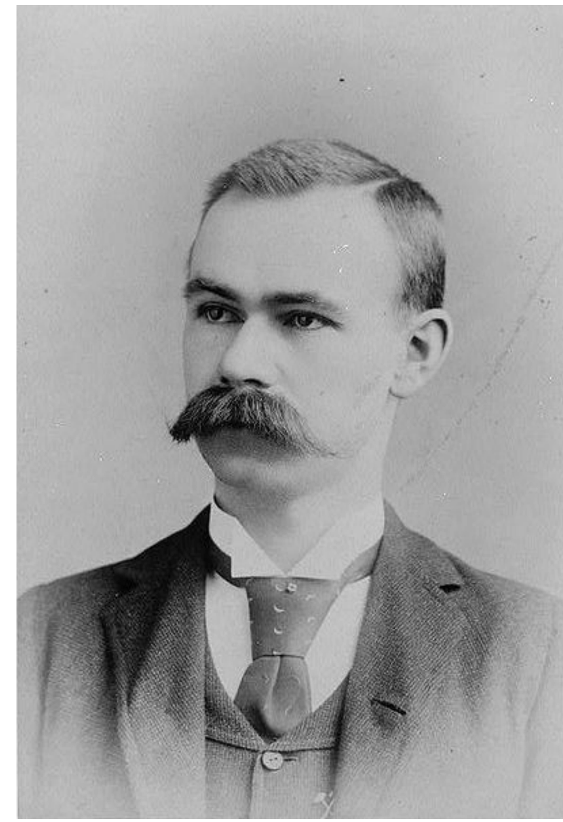
- Bucket Sort is interesting but sometimes the value of K is too big
- Example
  - We want to sort n numbers with 4 decimal digits
  - This woudl take $n+10000$ operations; if $n \log n < n+10000$, this would not be convenient
- Idea
  - Every decimal digit is a ideal candidate for Bucket Sort
  - if Bucket Sort is stable, then we could sort starting from less significant digits.

Radix Sort

- Algorithm designed in 1887 (Herman Hollerith and tabular machines)



Punch cards ordering nachine IBM 082 (13 slots, every card has 12 rows + 1 slot for discarded)



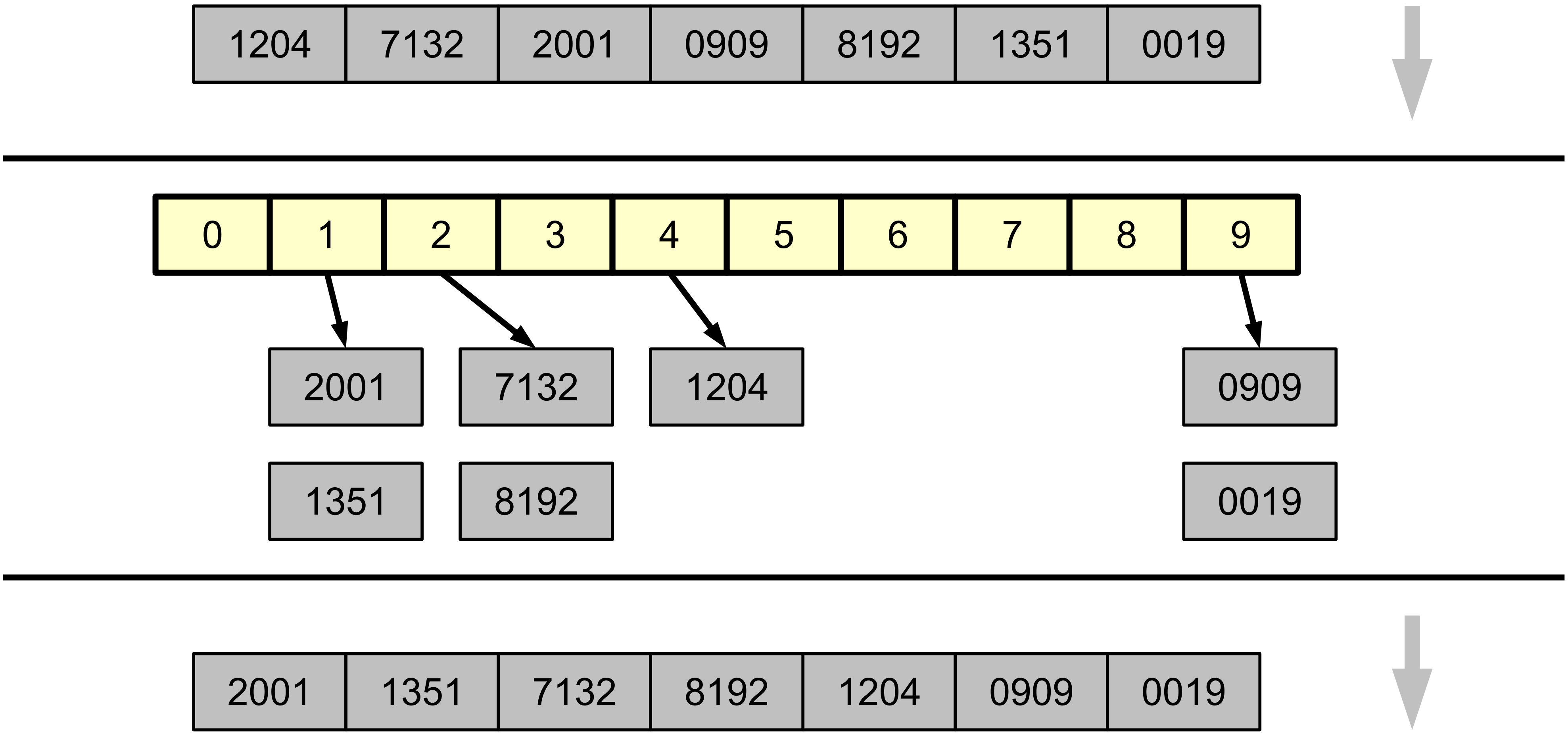
Herman Hollerith (1860—1929)
http://en.wikipedia.org/wiki/Herman_Hollerith

Radix Sort

- Idea:
  - First sort numbers based on unit digit
  - Then order numbers based on tenth digit
  - Then order numbers based on hundreds digit
  - ...
- Important: in every step it is fundamental to use a stabile sorting algorithm

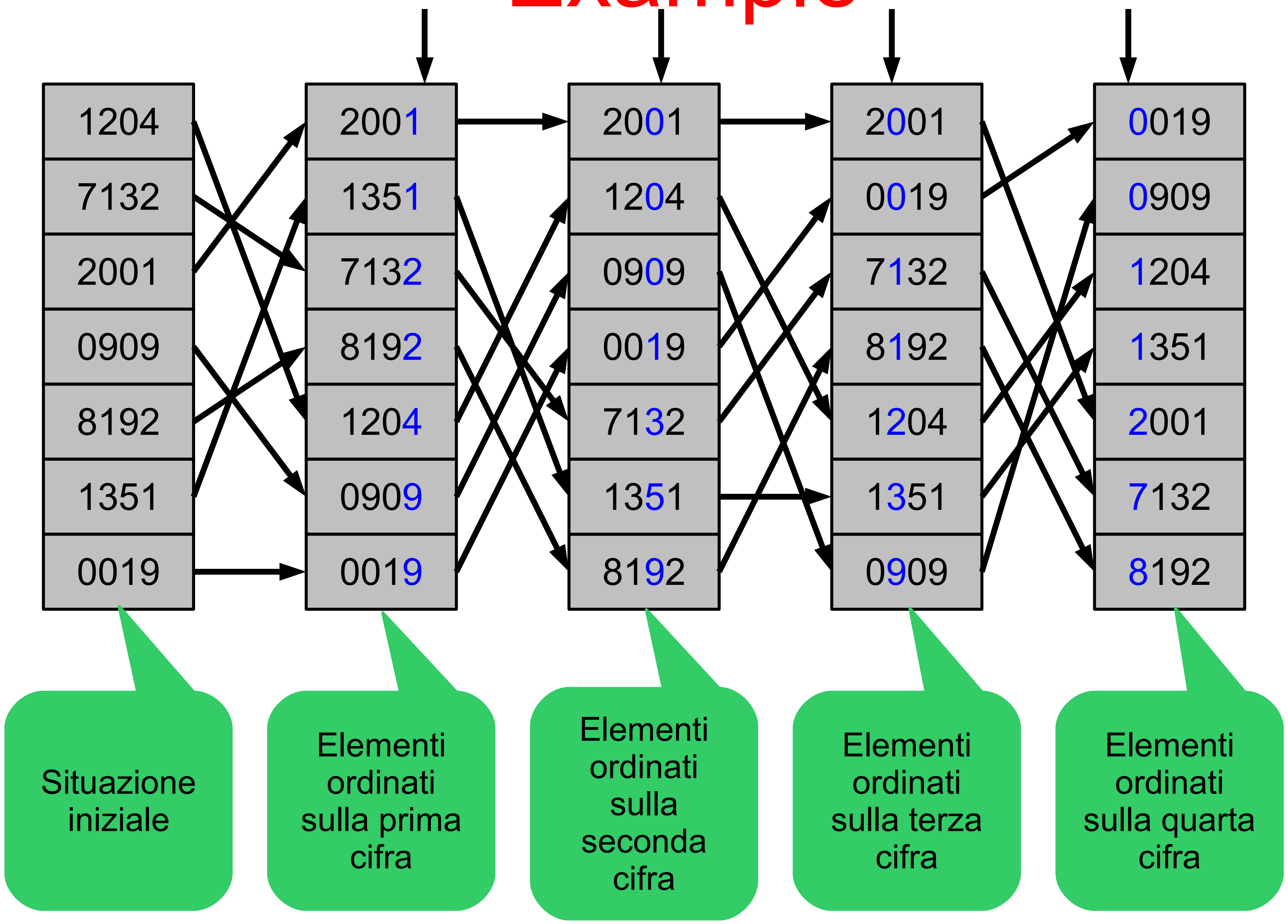
Example

Initial Array



Ordered array based on unit digit on the right

Example


1204
7132
2001
0909
8192
1351
0019
2001
1351
7132
8192
1204
0909
0019
2001
1204
0909
0019
7132
1351
8192
2001
0019
7132
8192
1204
1351
0909
0019
0909
1204
1351
2001
7132
8192
Situazione iniziale
Elementi ordinati sulla prima cifra
Elementi ordinati sulla seconda cifra
Elementi ordinati sulla terza cifra
Elementi ordinati sulla quarta cifra

Radix Sort

- Assume that elements in array A have values in interval [0, k–1]
- Sorting algorithm applies Bucket Sort on the digits realizing the base b representation of element of A

```
public static void radixSort(int[] A, int k, int b) {
    int t = 0;
    while (t <= Math.ceil(Math.log(k) / Math.log(b))){
        sortByDigit(A, b, t);
        t++;
    }
}
```

Sorting (stable) w.r.t. digit t
(t=0 is the less significant)

Number of base b digits
composing the integer k

sortByDigit(A, b, t)

- A specialized version of Bucket Sort to sort integer numbers based on the t-th digit (from the left) in base b

```
public static void sortByDigit(int[] A, int b, int t) {
    List[] Y = new List[b];
    int temp, c, j;
    for (int i = 0; i < b; i++) Y[i] = new LinkedList();
    for (int i = 0; i < A.length; i++) {
        temp = A[i] % ((int) (Math.pow(b, t + 1)));
        c = (int) Math.floor(temp / (Math.pow(b, t)));
        Y[c].add(new Integer(A[i]));
    }
    j=0;
    for (int i = 0; i < b; i++) {
        while (Y[i].size() > 0) {
            A[j] = ((Integer) Y[i].get(0)).intValue();
            j++;
        }
    }
}
```

Radix Sort

- Theorem
  - given n numbers withi d digits, where every digit can have b different values, then Radix Sort order with time O(d(n+b))
- proof:
  - inductive: after *i* calls of sortByDigit,numbers are sorted based on first *i* less significant digits.
  - **question**: prove it.
- proof (complexity):
  - d calls of sortByDigit, every call has cost O(n+b)

Radix Sort

- Theorem
  - Using value $b=\Theta(n)$ as the base,algorithm Radix Sort sorts n integer values in [0, k-1] with time complexity

$$O\left(n\left(1+\frac{\log k}{\log n}\right)\right)$$

  - **question**: prove it
  - Example:
    - 1.000.000 numbers with 32 bit, base $b=2^{16}$, 2 scan in linear time are sufficient
    - Warning : additional memory need O(b+n)

Sorting algs—summary

Algorithm	Stable?	In loco?	best	worst	average
Insertion Sort	Yes	Yes	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	Yes	Yes	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	Yes	No	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quick Sort	No	Yes	$\Theta(n \log n)$	$O(n^2)$	$\Theta(n \log n)$
Heap Sort	No	Yes	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Couting Sort	N.A.	No	$O(n+k)$	$O(n+k)$	$O(n+k)$
Bucket Sort	Yes	No	$O(n+k)$	$O(n+k)$	$O(n+k)$
Radix Sort	Yes	No	$O(d(n+b))$	$O(d(n+b))$	$O(d(n+b))$

Sorting algs—summary

- Insertion Sort / Selection Sort
  - $\Theta(n^2)$, stable, in loco, iterative.
- Merge Sort
  - $\Theta(n \log n)$, stable, needs $O(n)$ additional space, recursive (needs $O(\log n)$ stack space for pending recursive calls).
- Heap Sort
  - $O(n \log n)$, not stable, in loco, iterative.
- Quick Sort
  - $\Theta(n \log n)$ on the average, $\Theta(n^2)$ worst case, not stable, recursive (needs $O(\log n)$ space in stack).

Sorting algs—summary

- Counting Sort
  - O(n+k), needs O(k) additional memory, iterative. Convenient when k=O(n)
- Bucket Sort
  - O(n+k), stable, needs O(n+k) additional memory, iterative. Convenient when k=O(n)
- Radix Sort
  - O(d(n+b)), needs O(n+b) additional memory. Convenient when b=O(n).

Sorting algs—summary

- Divide-et-impera
  - Merge Sort: “divide” simple, “combine” complex
  - Quick Sort: “divide” complex, “combine” null
- Use of efficient data structures as service structures
  - Heap Sort based on Heap
- Randomization
  - Technique to avoid the risk to incur in the worst case
- Guideline: dependency from the model
  - By changing the set of assumptions we can define more efficient algorithms for the sorting problem (and for any problem in general).