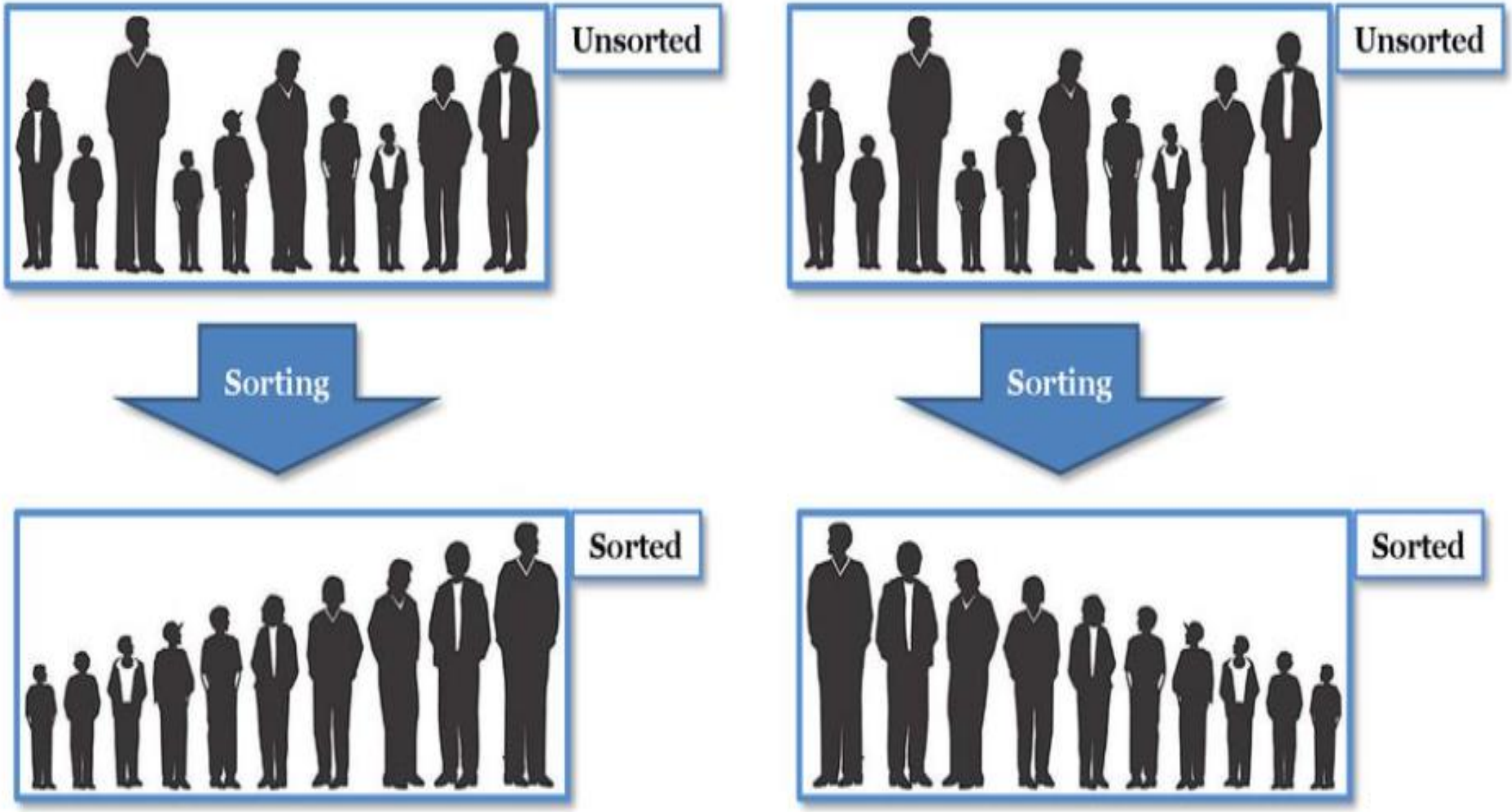
Algorithms & Data Structure
Sorting

Kiran Waghmare


Unsorted
Unsorted
Sorting
Sorting
Sorted
Sorted

Sorting Techniques

Introduction

- Sorting refers to arranging a set of data in some logical order
- For ex. A telephone directory can be considered as a list where each record has three fields - name, address and phone number.
- Being unique, phone number can work as a key to locate any record in the list.

https://visualgo.net/en/sorting
https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Introduction

- Sorting is among the most basic problems in algorithm design.

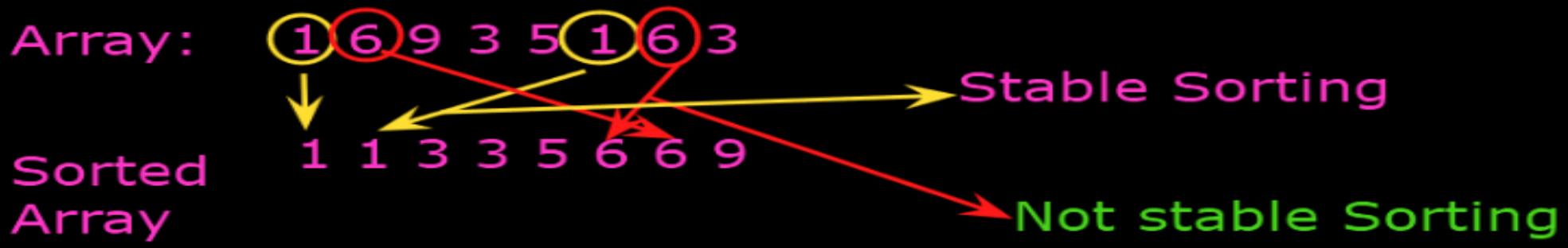
- We are given a sequence of items, each associated with a given key value.
- And the problem is to rearrange the items so that they are in an increasing(or decreasing) order by key.

- The methods of sorting can be divided into two categories:
  - Internal Sorting
  - External Sorting

-External Sorting

-data to be sorted cann't be accoumodate in the memory at the time of sortig and some data has to be kept in additional memeory.
-e.g.,

Stable and Not stable Sorting:



Stable:does not change the sequence of similar elements.

Not Stable: changes the sequence of similar content

Efficiency of Sorting Algorithm:

1.Complexity: Running time
-Length of code
-execution time
-Amount of memory taken by algorithm

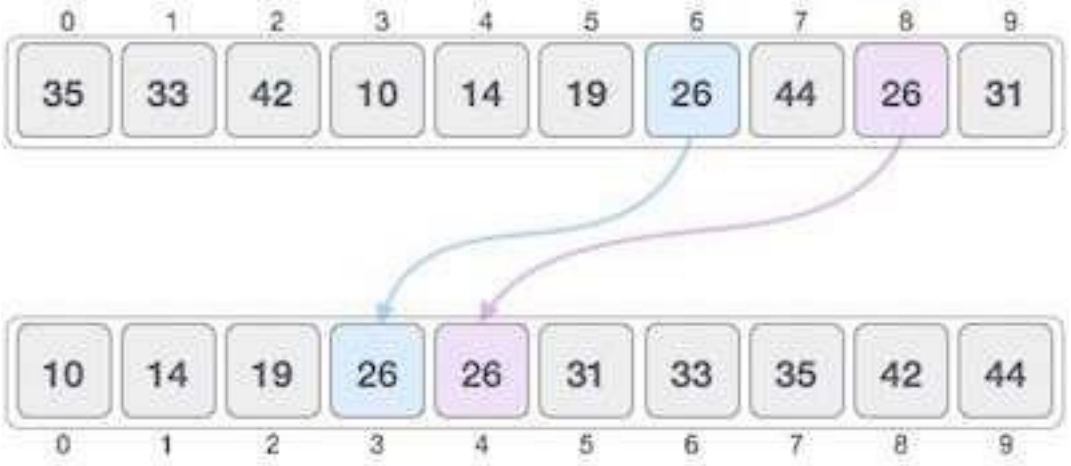
- Internal Sorting
  - ✓If all the data that is to be sorted can be adjusted at a time in main memory, then internal sorting methods are used

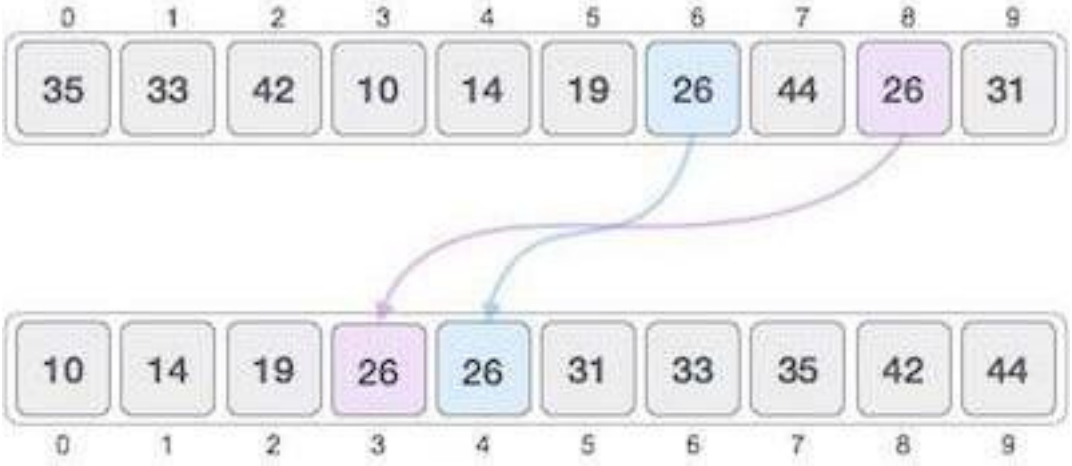
- External Sorting
  - ✓When the data to be sorted can't be accommodated in the memory at the same time and some has to be kept in auxiliary memory, then external sorting methods are used.

❖ NOTE: We will only consider internal sorting

Stable and Not Stable Sorting

- If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.





- If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.

Efficiency of Sorting Algorithm

- The complexity of a sorting algorithm measures the running time of a function in which n number of items are to be sorted.

- The choice of sorting method depends on efficiency considerations for different problems.

- Tree most important of these considerations are:
  - The length of time spent by programmer in coding a particular sorting program
  - Amount of machine time necessary for running the program
  - The amount of memory necessary for running the program

Efficiency of SortingAlgorithm

- Various sorting methods are analyzed in the cases like – best case, worst case or average case.

- Most of the sort methods we consider have requirements that range from 0(nlogn) to $0(n^2)$.

- A sort should not be selected only because its sorting time is 0(nlogn); the relation of the file size n and the other factors affecting the actual sorting time must be considered

Efficiency of SortingAlgorithm

- Determining the time requirement of sorting technique is to actually run the program and measure its efficiency.

- Once a particular sorting technique is selected the need is to make the program as efficient as possible.

- Any improvement in sorting time significantly affect the overall efficiency and saves a great deal of computer time.

Efficiency of SortingAlgorithm

- Space constraints are usually less important than time considerations.
- The reason for this can be, as for most sorting programs, the amount of space needed is closer to 0(n) than to $0(n^2)$
- The second reason is that, if more space is required, it can almost always be found in auxiliary storage.

Things to remember

- An ideal sort is an in-place sort where the algorithm uses no additional array storage, and hence it is possible to sort very large lists without the need to allocate additional working storage.

- A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

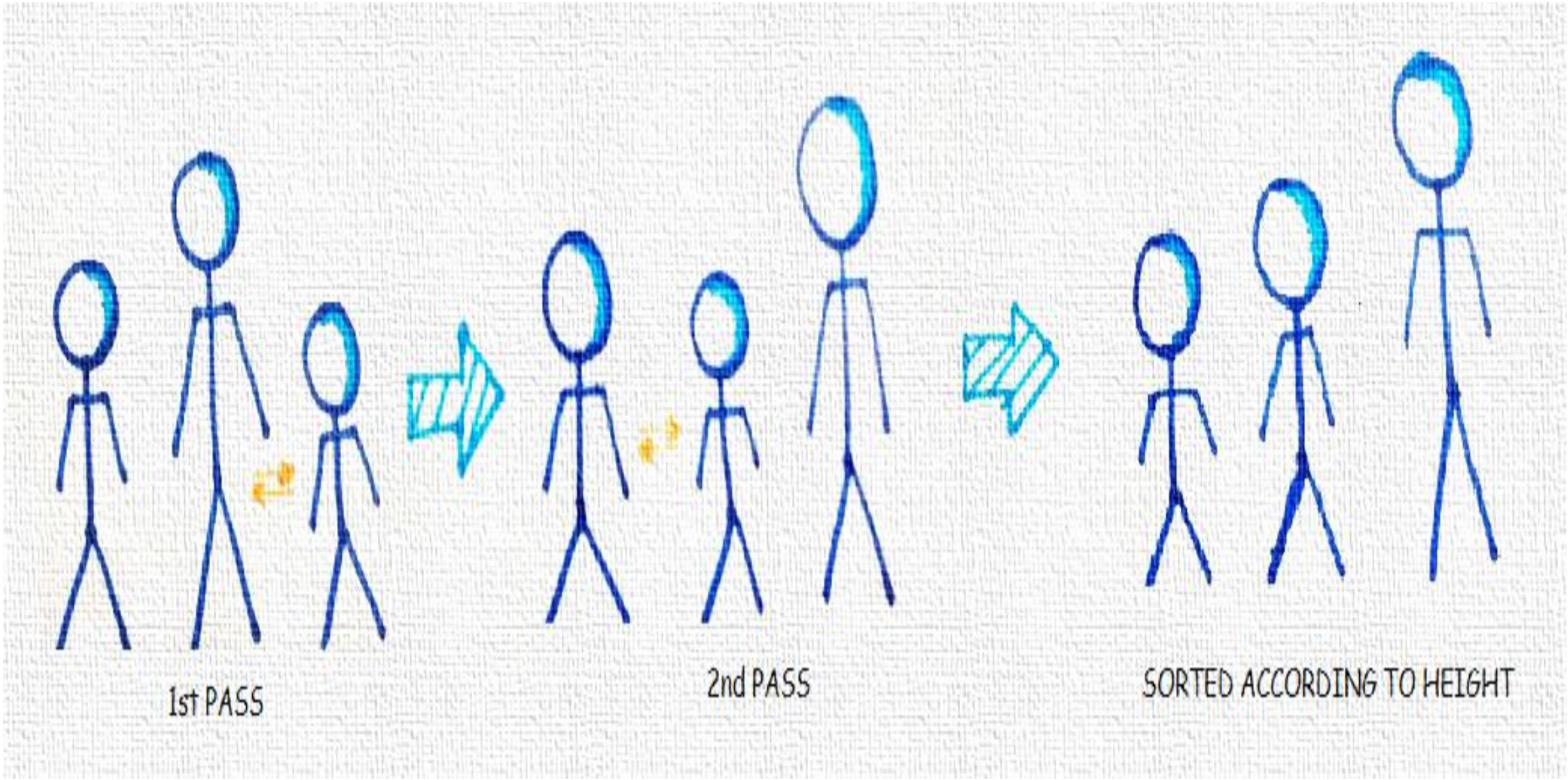
- Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. And some sorting algorithms are not, like Heap Sort, Quick Sort, etc.

Things to remember

- Sorting can be performed in many ways.

- Over a time several methods (or algorithms) are being developed to sort data(s).
  - Bubble sort, Selection sort, Quick sort, Merge sort, Insertion sort are the few sorting techniques discussed in this chapter.

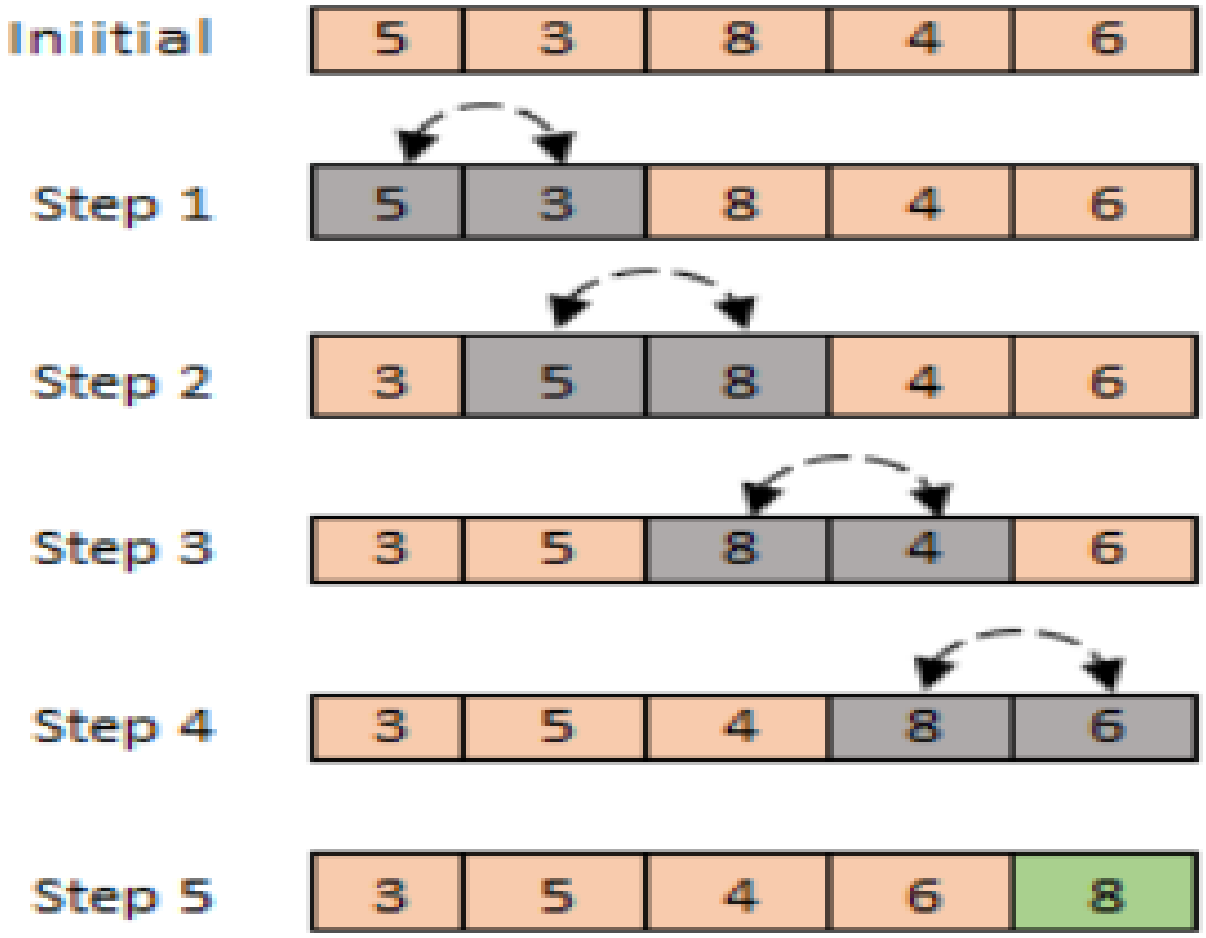
- It is very difficult to select a sorting algorithm over another. And there is no sorting algorithm better than all others in all circumstances.


1st PASS
2nd PASS
SORTED ACCORDING TO HEIGHT

BUBBLE SORT

- In bubble sort, each element is compared with its adjacent element.
- We begin with the 0th element and compare it with the 1st element.
- If it is found to be greater than the 1st element, then they are interchanged.
- In this way all the elements are compared (excluding last) with their next element and are interchanged if required
- On completing the first iteration, largest element gets placed at the last position. Similarly in second iteration second largest element gets placed at the second last position and so on.

Bubble sort example



TIME COMPLEXITY

- Best Case
  - sorting a sorted array by bubble sort algorithm
  - In best case outer loop will terminate after one iteration, i.e it involves performing one pass which requires n-1 comparison

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

- Worst Case
  - Suppose an array [5,4,3,2,1], we need to move first element to end of an array
  - n-1 times the swapping procedure is to be called

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

- Average Case
  - Difficult to analyse than the other cases
  - Random inputs, so in general

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

- Space Complexity
  - O(n)

Algorithm 1: Bubble sort

```
Data: Input array A[]
Result: Sorted A[]
int i, j, k;
N = length(A);
for j = 1 to N do
    for i = 0 to N-1 do
        if A[i] > A[i+1] then
            temp = A[i];
            A[i] = A[i+1];
            A[i+1] = temp;
        end
    end
end
```

SELECTION SORT

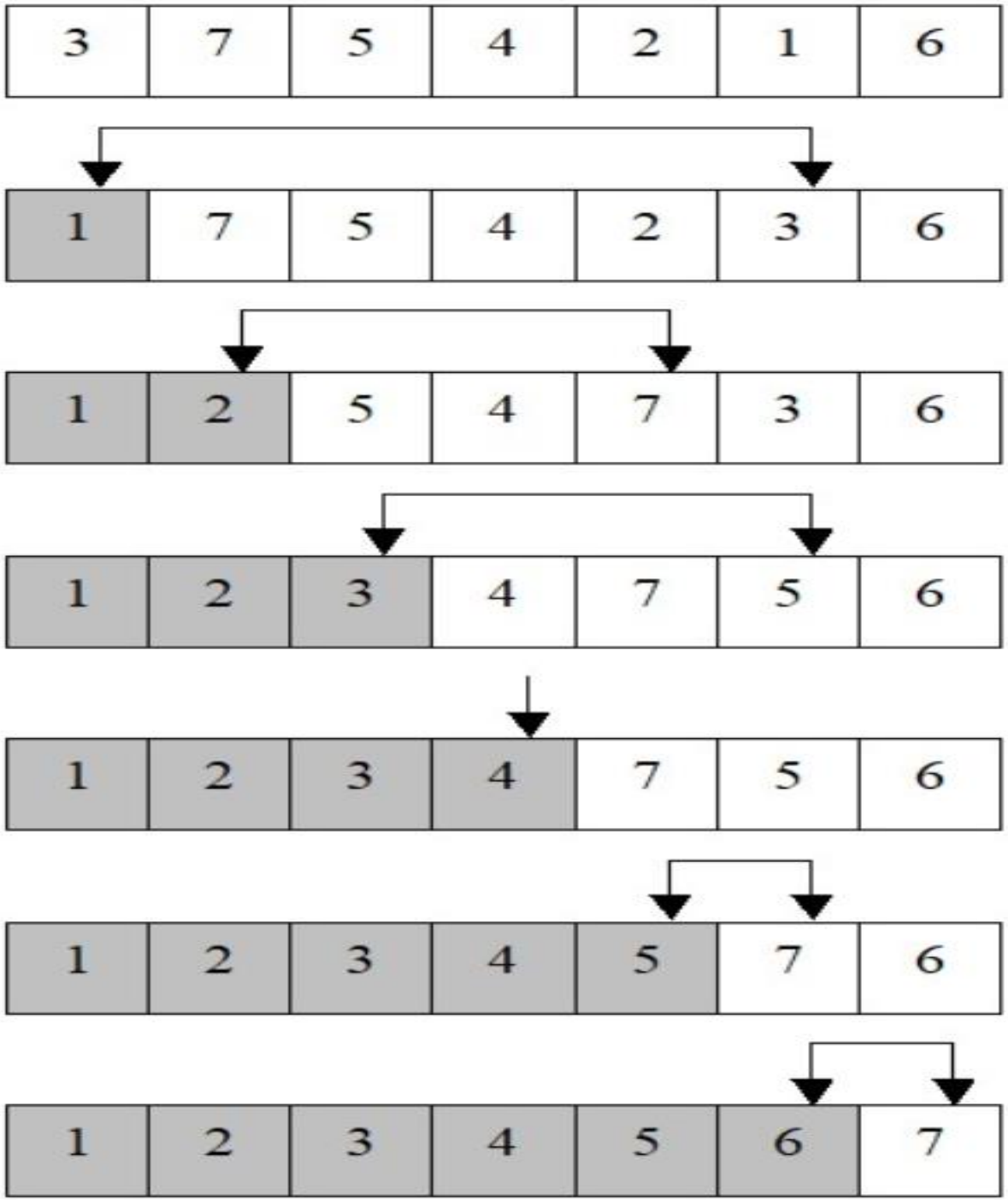
- Find the least(or greatest) value in the array, swap it into the leftmost(or rightmost) component, and then forget the leftmost component, Do this repeatedly.

- Let a[n] be a linear array of n elements. The selection sort works asfollows:

- Pass 1: Find the location loc of the smallest element in the list of n elements a[0], a[1], a[2], a[3], ...,a[n-1] and then interchange a[loc] and a[0].

- Pass 2: Find the location loc of the smallest element int the sub-list of n-1 elements a[1], a[2], a[3], ...,a[n-1] and then interchange a[loc] and a[1] such that a[0], a[1] are sorted.

- Then we will get the sorted list
  a[0]<=a[2]<=a[3]…..<=a[n-1]


3 7 5 4 2 1 6
Original sequence
1 7 5 4 2 3 6
Step 0: 1 ↔ 3
1 2 5 4 7 3 6
Step 1: 2 ↔ 7
1 2 3 4 7 5 6
Step 2: 3 ↔ 5
1 2 3 4 7 5 6
Step 3: 4 ↔ 4
1 2 3 4 5 7 6
Step 4: 5 ↔ 7
1 2 3 4 5 6 7
Step 5: 6 ↔ 7

Algorithm:

```
SelectionSort(A)
{
        for( i = 0;i < n ;i++)
        {
                least=A[i];
                p=i;
                for ( j = i + 1;j < n ;j++)
                {
                        if (A[j] < A[i])
                        least= A[j]; p=j;
                }
        }
        swap(A[i],A[p]);
}
```

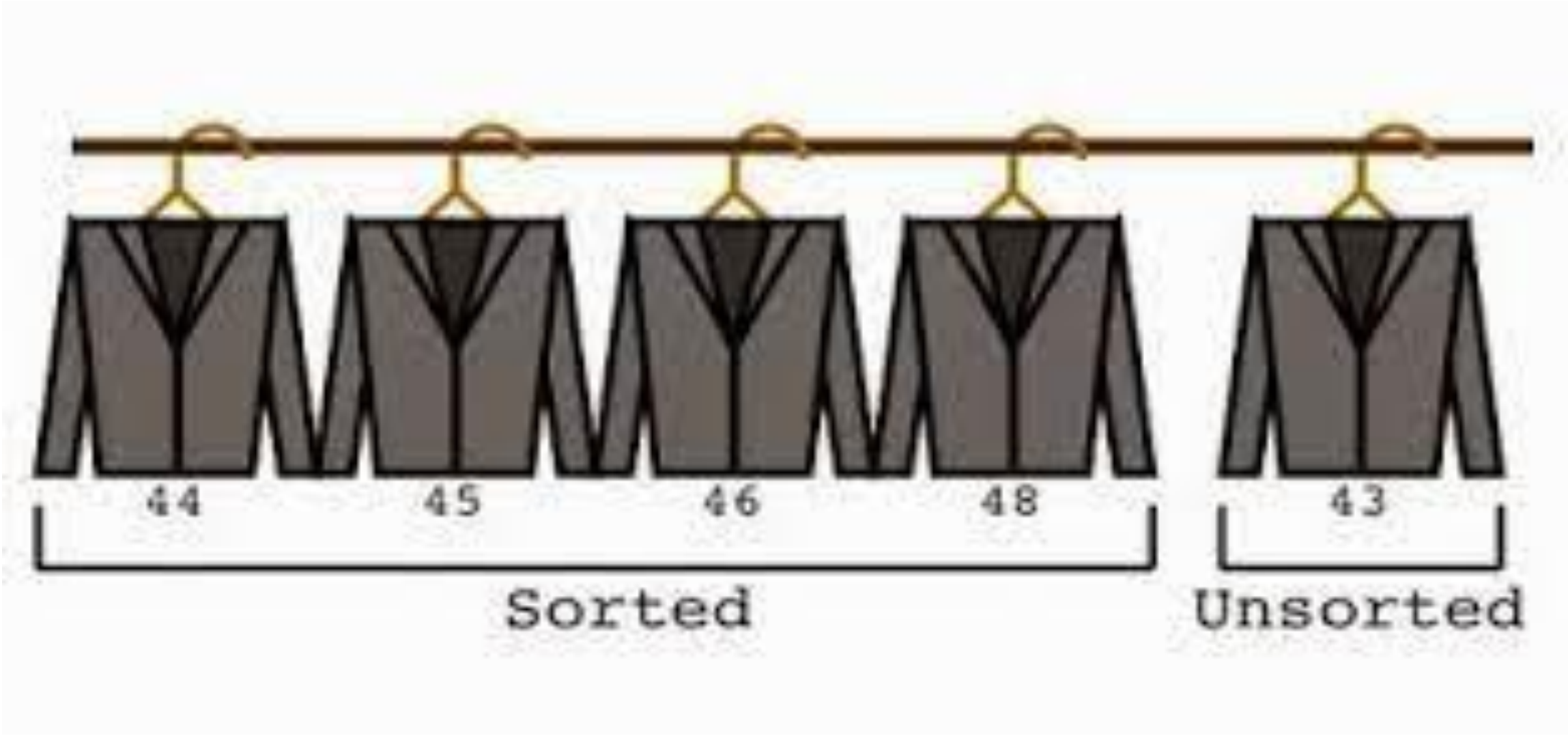
Time Complexity

- Inner loop executes (n-1) times when i=0, (n-2) times when i=1 and so on:
- Time complexity = (n-1) + (n-2) + (n-3) + …..... +2+1

  = $O(n^2)$

Space Complexity

- Since no extra space beside n variables is needed for sorting so
- O(n)

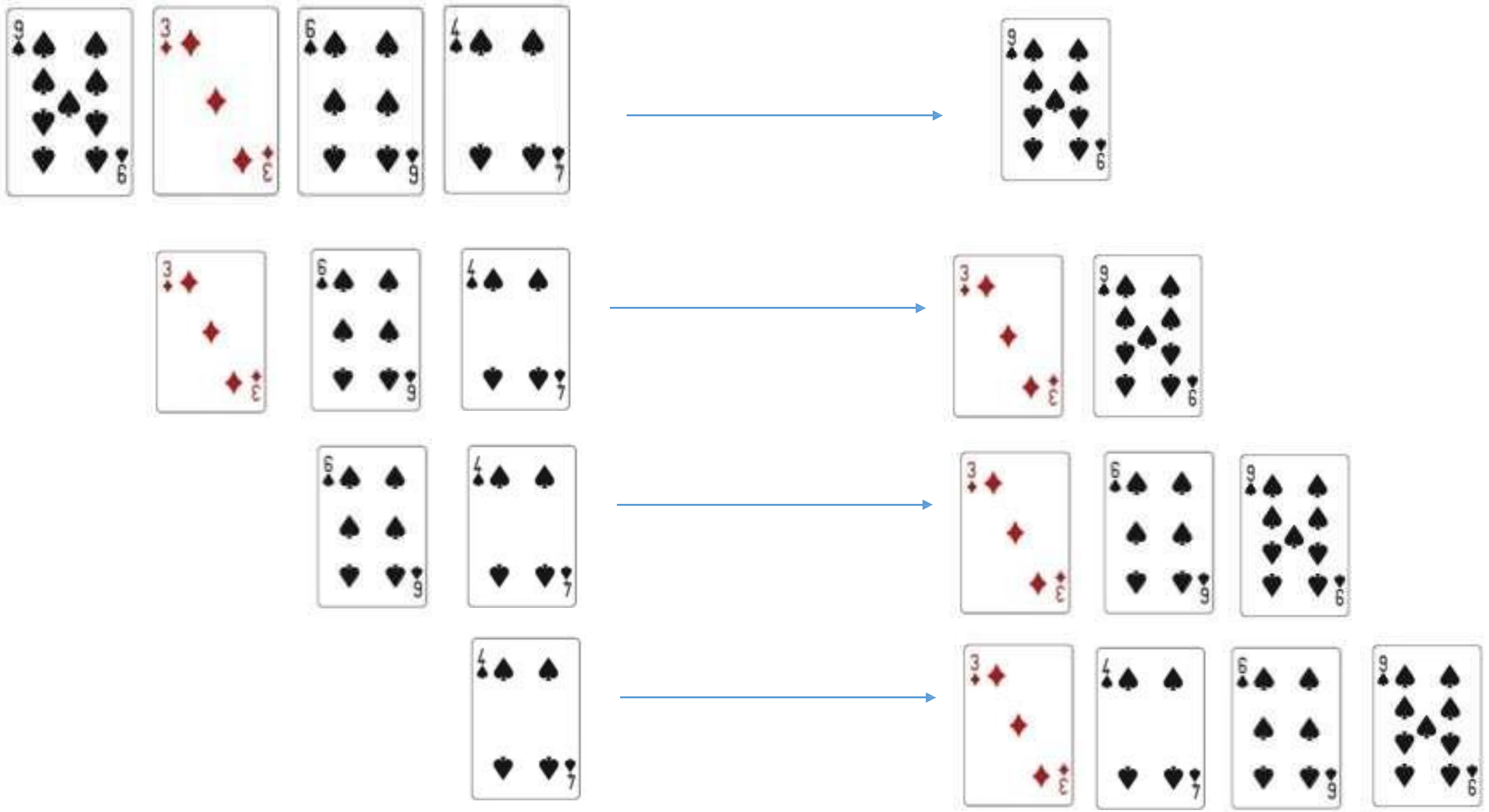

44
45
46
48
43
Sorted
Unsorted

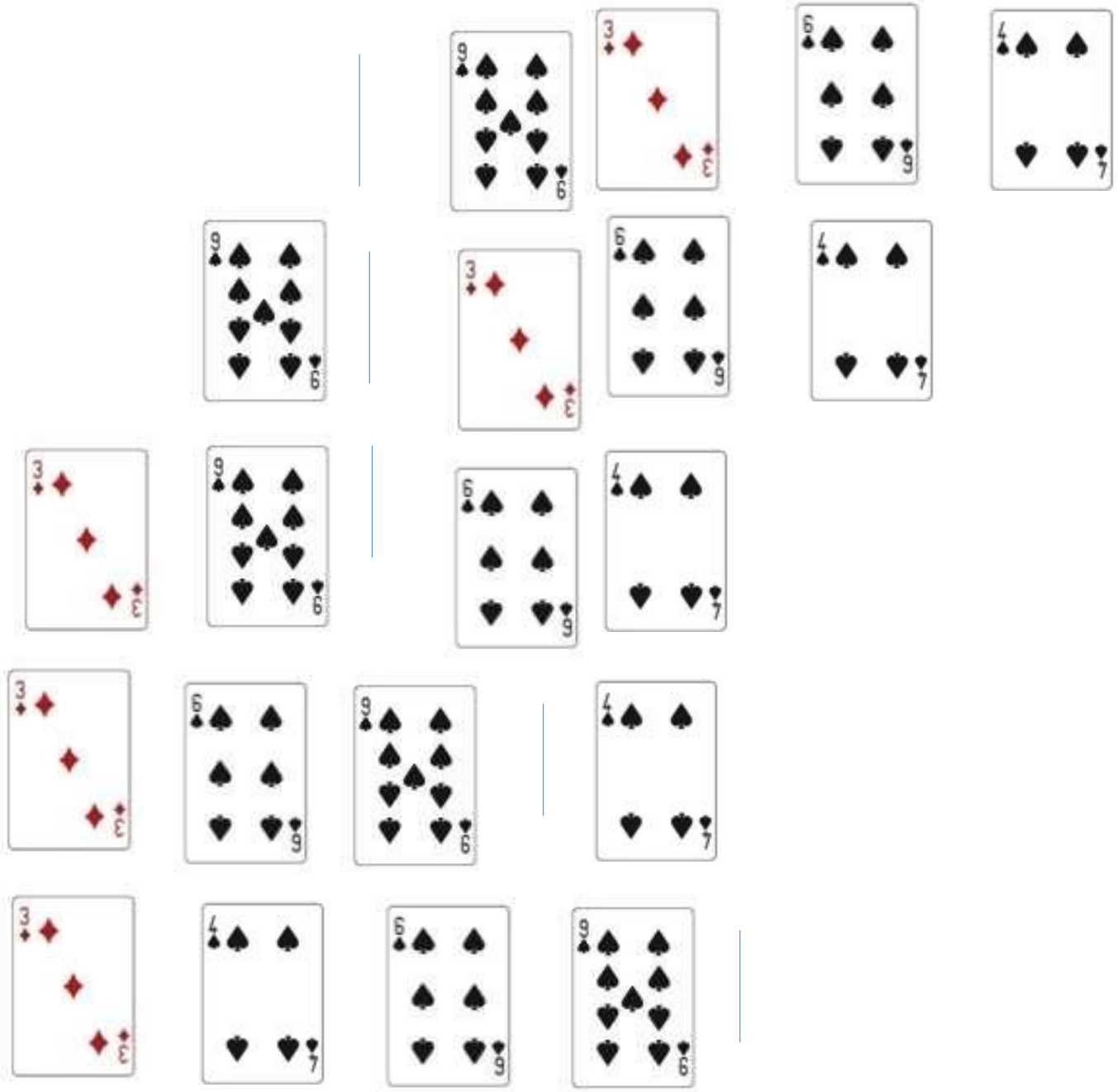
Insertion Sort

- Like sorting a hand of playing cards start with an empty hand and the cards facing down the table.
- Pick one card at a time from the table, and insert it into the correct position in the left hand.
- Compare it with each of the cards already in the hand, from right to left
- The cards held in the left hand are sorted.

Characteristics of Insertion Sort:

- This algorithm is one of the simplest algorithm with simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.





Insertion Sort

- Suppose an array a[n] with n elements. The insertion sort works asfollows:

Pass 1: a[0] by itself is trivially sorted.
Pass 2: a[1] is inserted either before or after a[0] so that a[0], a[1] is sorted.

Pass 3: a[2] is inserted into its proper place in a[0],a[1] that is before a[0], between a[0] and a[1], or after a[1] so that a[0],a[1],a[2] is sorted.

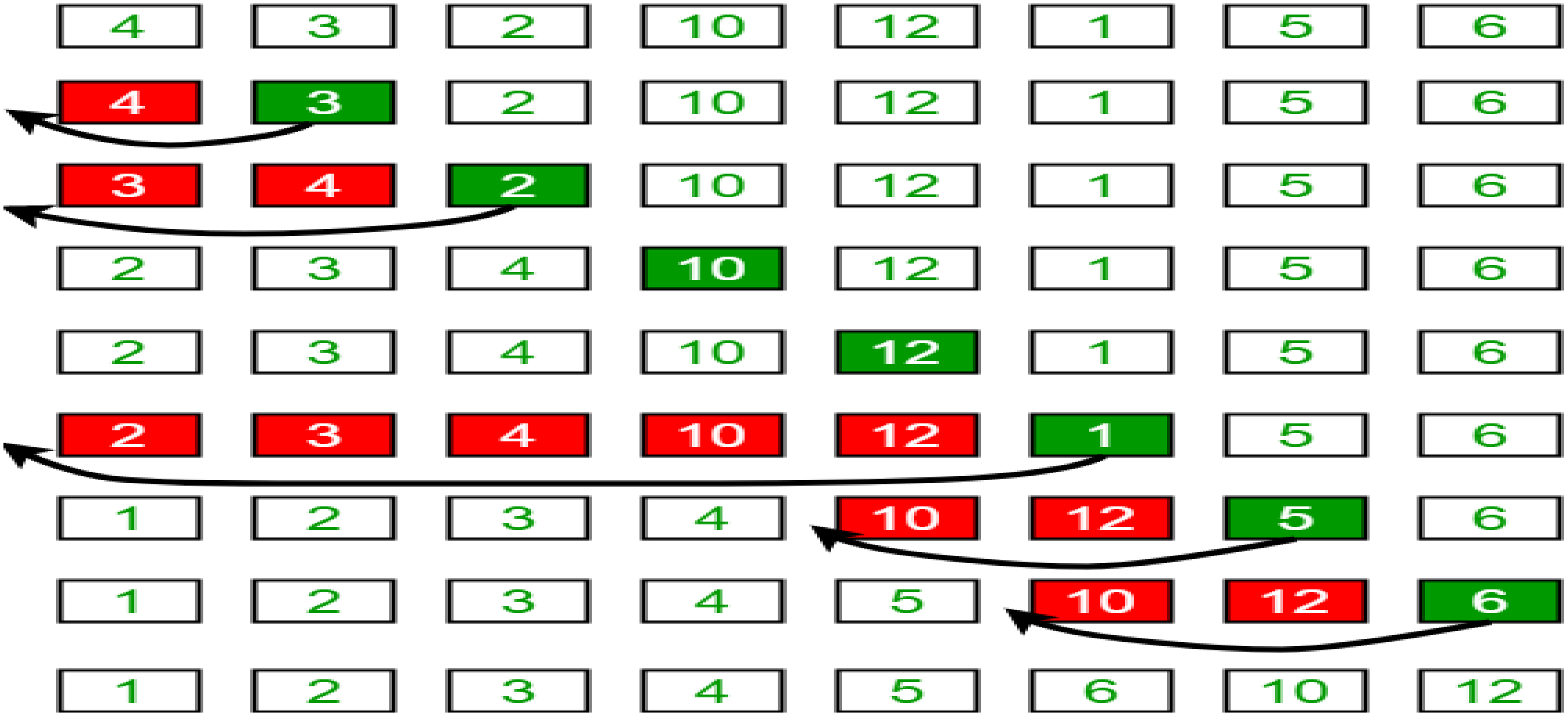
pass N: a[n-1] is inserted into its proper place in a[0],a[1],a[2],.....,a[n-2] so that a[0],a[1],a[2],.....,a[n-1] is sorted with n elements.

INSERTION-SORT(*A*)	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
▷Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i — 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

t_j: # of times the while statement is executed at iteration j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Insertion Sort Execution Example



Time Complexity

- Best Case:
  - If the array is all but sorted then
  - Inner Loop wont execute so only some constant time the statements will run
  - So Time complexity= O(n)

- Worst Case:
  - Array element in reverse sorted order
  - Time complexity=$O(n^2)$

- Space Complexity
  - Since no extra space beside n variables is needed for sorting so
  - Space Complexity = O(n)

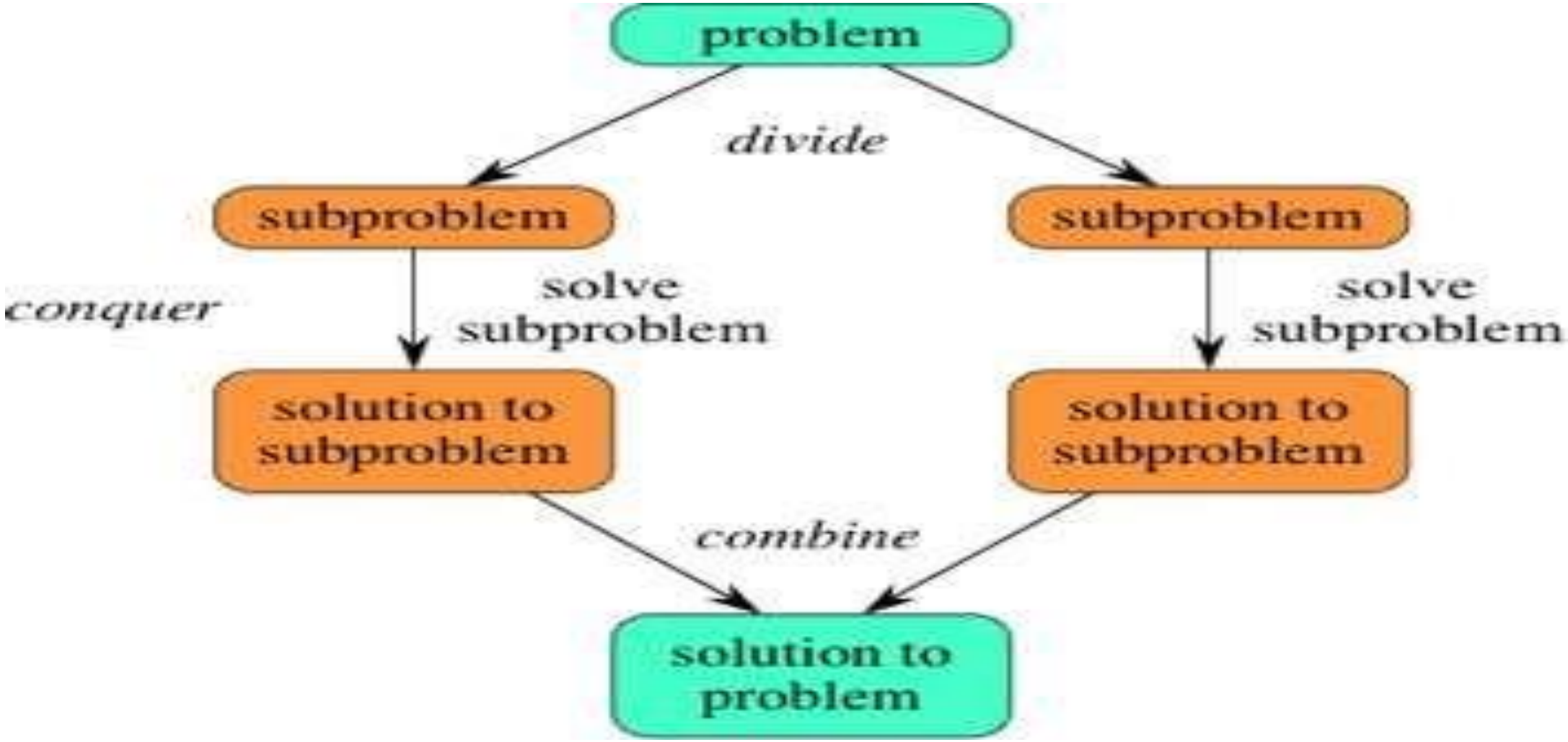
Divide and conquer algorithms

- The sorting algorithms we've seen so far have worst-case running times of $O(n^2)$

- When the size of the input array is large, these algorithms can take a long time to run.

- Now we will discuss two sorting algorithms whose running times are better
  - Merge Sort
  - Quick Sort

Divide-and-conquer



Merge Sort

- Merge sort is a sorting technique based on divide and conquer technique.
- Merge sort first divides the array into equal halves and then combines them in a sorted manner.
- With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge Sort

- Because we're using divide-and-conquer to sort, we need to decide what our sub problems are going to be.

- Full Problem: Sort an entire Array
- Sub Problem: Sort a sub array

- Lets assume array[p..r] denotes this subarray of array.
- For an array of n elements, we say the original problem is to sort array[0..n-1]

Merge Sort

- Here's how merge sort uses divide and conquer
  1. *Divide* by finding the number q of the position midway between p and r. Do this step the same way we found the midpoint in binary search: add p and r, divide by 2, and round down.

  2. *Conquer* by recursively sorting the subarrays in each of the two sub problems created by the divide step. That is, recursively sort the subarray array[p..q] and recursively sort the subarray array[q+1..r].

  3. *Combine* by merging the two sorted subarrays back into the single sorted subarray array[p..r].

Merge Sort

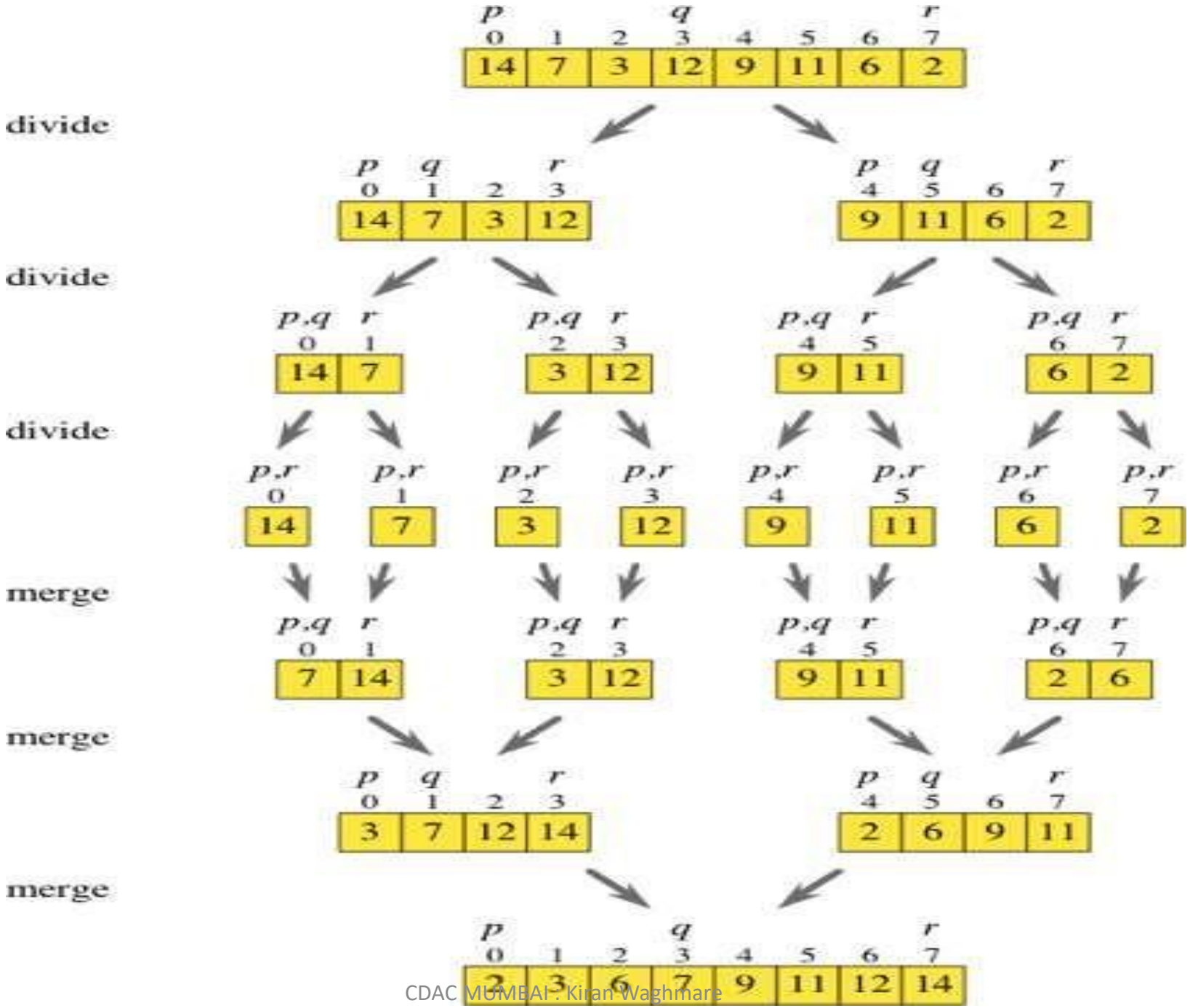
- Let's start with array holding [14,7,3,12,9,11,6,2]
- We can say that array[0..7] where p=0 and r=7
- In the divide step we compute q=3
- The conquer step has us sort the two subarrays
  - array[0..3] = [14,7,3,12]
  - array[4..7]= [9,11,6,2]
  - When we comeback from the conquer step, each of the two subarrays is sorted i.e.
  - array[0..3] = [3,7,12,14]
  - array[4..7]= [2,6,9,11]
- Finally, the combine step merges the two sorted subarrays in first half and the second half, producing the final sorted array [2,3, 6,7,9, 11, 12,14]

How did the subarray array[0..3] become sorted?

- It has more than two element so it's not a base case.
- So with p=0 and r=3, compute q=1, recursively sort array[0..1] and array[2..3], resulting in array[0..3] containing [7,14,3,12] and merge the fist half with the second half, producing [3,7,12,14]

How did the subarray array[0..1] become sorted?

- With p=0 and r=1, compute q=0, recursively sort array[0..0] ([14]) and array[1..1] ([7]), resulting in array[0..1] still containing [14, 7], and merge the first half with the second half, producing [7, 14].


divide
divide
divide
merge
merge
merge
p q r
0 1 2 3 4 5 6 7
14 7 3 12 9 11 6 2
p q r
0 1 2 3
14 7 3 12
p q r
4 5 6 7
9 11 6 2
p,q r
0 1
14 7
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
6 2
p,r
0
14
p,r
1
7
p,r
2
3
p,r
3
12
p,r
4
9
p,r
5
11
p,r
6
6
p,r
7
2
p,q r
0 1
7 14
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
2 6
p q r
0 1 2 3
3 7 12 14
p q r
4 5 6 7
2 6 9 11
p q r
0 1 2 3 4 5 6 7
2 3 6 7 9 11 12 14

Divide-and-Conquer

- **Divide the problem into a number of sub-problems**
  - Similar sub-problems of smaller size
- **Conquer the sub-problems**
  - Solve the sub-problems recursively
  - Sub-problem size small enough $\Rightarrow$ solve the problems in straightforward manner
- **Combine the solutions of the sub-problems**
  - Obtain the solution for the original problem

Merge Sort:

Here is the pseudocode for Merge Sort, modified to include a counter:

```
count ← 0
Merge_Sort(A, p, r)
1       if p < r
2               then  q ← ⌊(p + r)/2⌋
3                     Merge-Sort (A, p, q)
4                     Merge-Sort (A, q+1, r)
5                     Merge (A, p, q, r)
```

And here is the modified algorithm for the Merge function used by Merge Sort:

```
Merge (A, p, q, r)
1       n1 ← (q — p) + 1
2       n2 ← (r — q)
3       create arrays L[1..n1+1] and R[1..n2+1]
4       for i ← 1 to n1 do
5               L[i] ← A[(p + i) -1]
6       for j ← 1 to n2 do
7               R[j] ← A[q + j]
8       L[n1 + 1] ← ∞
9       R[n2 + 1] ← ∞
10      i ← 1
11      j ← 1
12      for k ← p to r do
12.5            count ← count + 1
13              if L[I] <= R[j]
14                      then  A[k] ← L[i]
15                            i ←i + 1
16                      else A[k] ← R[j]
17                            j ← j + 1
```

Analysis of merge Sort

- We can view merge sort as creating a tree of calls, where each level of recursion is a level in the tree.
- Since number of elements is divided in half each time, the tree is balanced binary tree.
- The height of such a tree tend to be log n

Analysis of merge Sort

- Divide and conquer
- Recursive
- Stable
- Not In-place
- 0(n) space complexity
- 0(nlogn) time complexity

Analysis of merge Sort

- The time to merge sort n numbers is equal to the time to do two recursive merge sorts of size n/2 plus the time to merge, which is linear.
- We can express the number of operations involved using the following recurrence relations

$$T(1)=1$$

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

- Going further down using the same logic

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

- Continuing in this manner, we can write

$$T(n)=nT(1)+nlogn$$

$$=n+nlogn$$

$$T(n)=0(nlogn)$$

Analysis of mergeSort

- Although merge sort's running time is very attractive it is not preferred for sorting data in main memory.
- Main problem arises when the it uses linear extra memory as we need to copy the original array into two arrays of half the size and the additional work spent on copying to the temporary array and back
- This might considerably slow down the time to sort
- However merge sort can work very well with linked list as it doesn't require additional space. Since we only need to change pointer links rather than shifting the element.

Quick Sort

- Quick sort is one of the most popular sorting techniques.
- As the name suggests the quick sort is the fastest known sorting algorithm in practice.
- It has the best average time performance.
- It works by partitioning the array to be sorted and each partition in turn sorted recursively. Hence also called partition exchange sort.

Quick Sort

- In partition one of the array elements is choses as a pivot element

- Choose an element pivot=a[n-1]. Suppose that elements of an array a are partitioned so that pivot is placed into position I and the following condition hold:
  - Each elements in position 0 through i-1 is less than or equal to pivot
  - Each of the elements in position i+1 through n-1 is greater than or equal to key

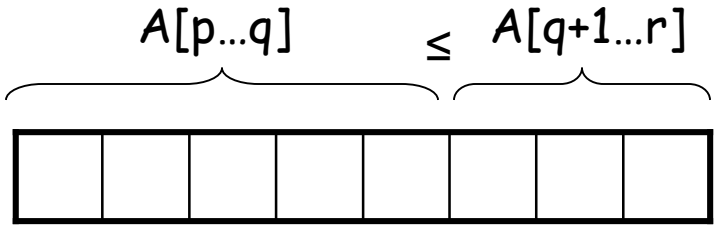
- The pivot remains at the i^{th} position when the array is completely sorted. Continuously repeating this process will eventually sort an array.

Algorithm

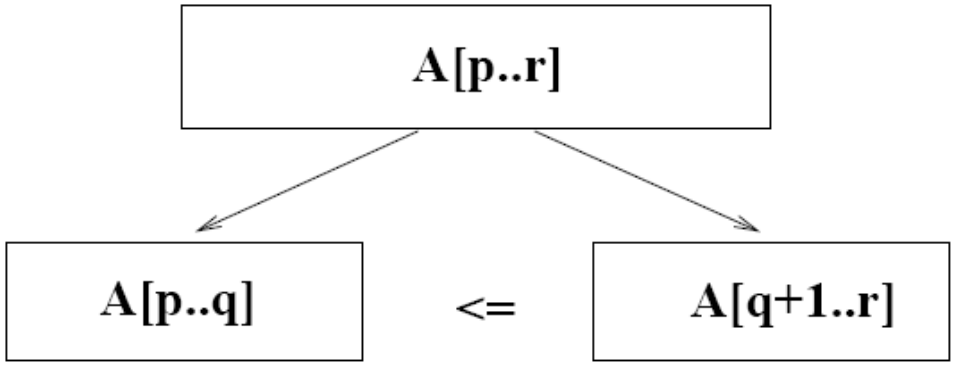
- Choosing a pivot
  - Topartition the list we first choose a pivot element
- Partitioning
  - Then we partition the elements so that all those with values less than pivot are placed on the left side and the higher vale on the right
  - Check if the current element is less than the pivot.
    - If lesser replace it with the current element and move the wall up one position
    - else move the pivot element to current element and vice versa
- Recur
  - Repeat the same partitioning step unless all elements are sorted

Quicksort

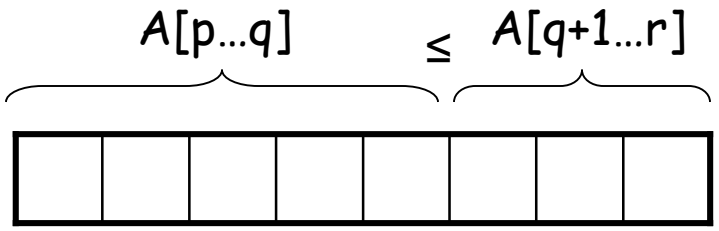
- **Sort an array $A[p...r]$**



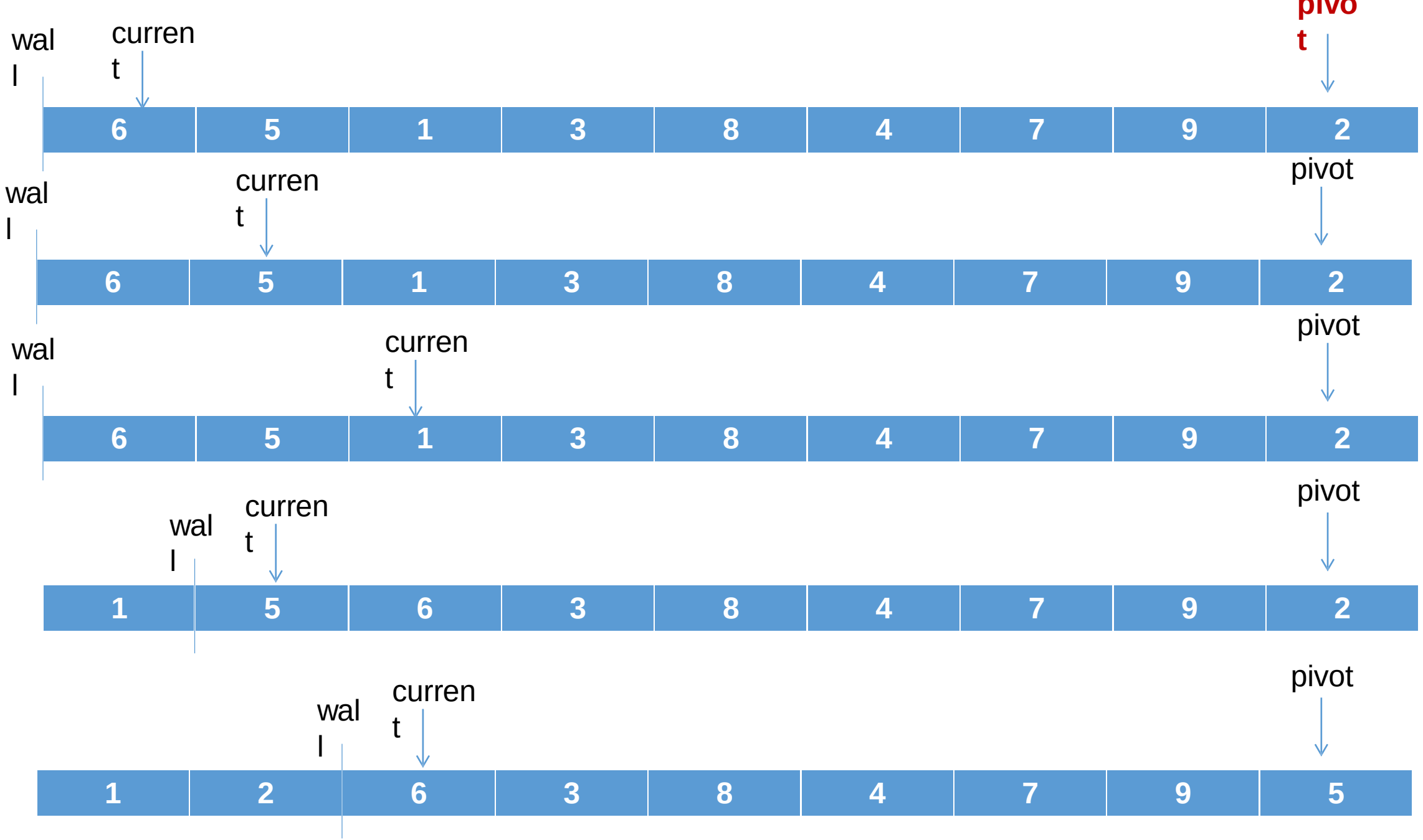
- **Divide**
  - Partition the array A into 2 subarrays $A[p..q]$ and $A[q+1..r]$, such that each element of $A[p..q]$ is smaller than or equal to each element in $A[q+1..r]$
  - Need to find index q to partition the array



Quicksort



- **Conquer**
  - Recursively sort $A[p..q]$ and $A[q+1..r]$ using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted


wall
current
pivot
6 5 1 3 8 4 7 9 2
wall
current
pivot
6 5 1 3 8 4 7 9 2
wall
current
pivot
6 5 1 3 8 4 7 9 2
wall
current
pivot
1 5 6 3 8 4 7 9 2
wall
current
pivot
1 2 6 3 8 4 7 9 5

The following procedure implements quicksort:

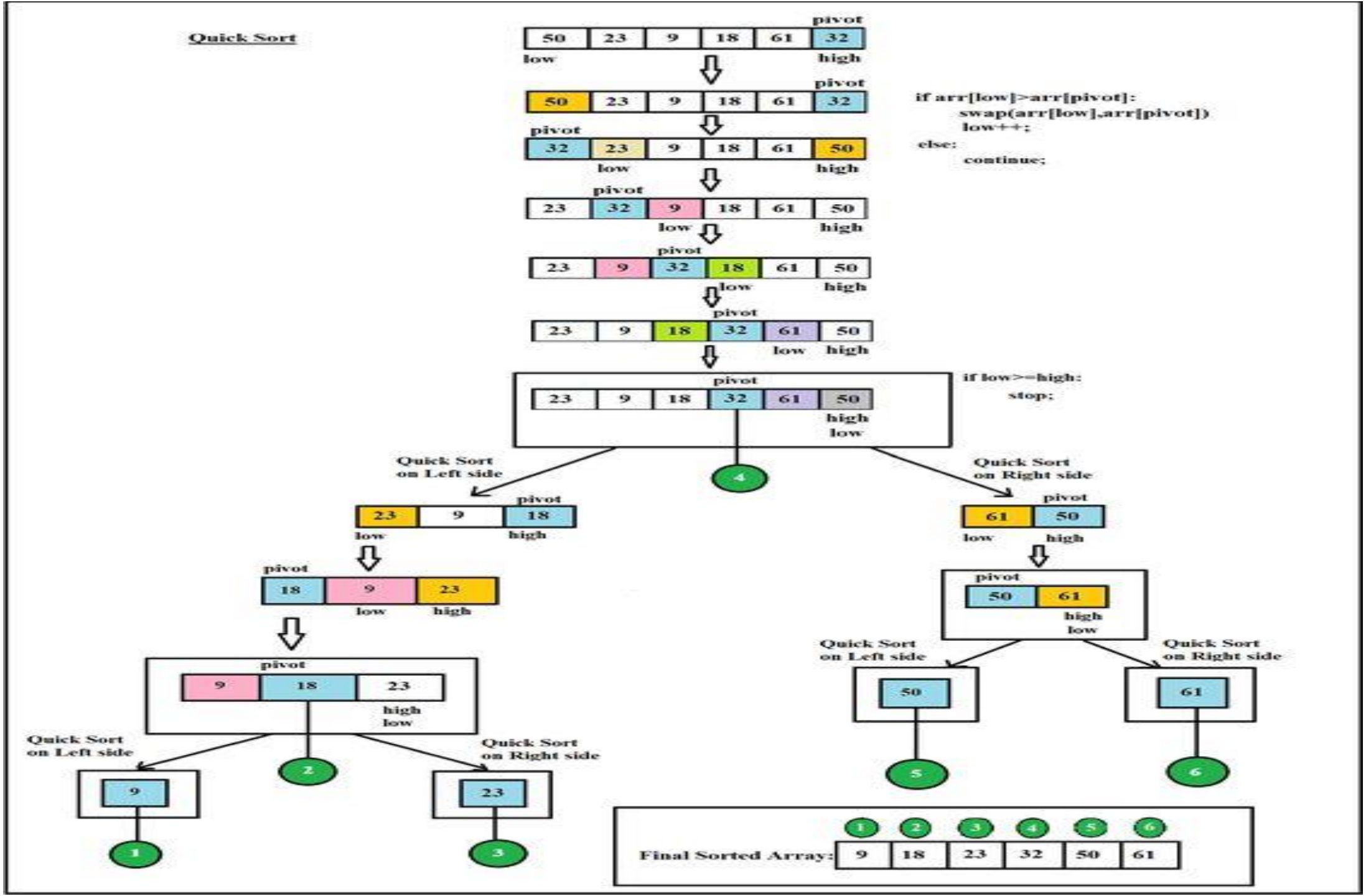
```
QUICKSORT(A, p, r)
1   if p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT$(A, 1, A.length)$.

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p\,..\,r]$ in place.

```
PARTITION(A, p, r)
1   x = A[r]
2   i = p − 1
3   for j = p to r − 1
4       if A[j] ≤ x
5           i = i + 1
6           exchange A[i] with A[j]
7   exchange A[i + 1] with A[r]
8   return i + 1
```


Quick Sort
50 23 9 18 61 32
pivot
low
high
if arr[low]>arr[pivot]:
swap(arr[low],arr[pivot])
low++;
else:
continue;
32 23 9 18 61 50
23 32 9 18 61 50
23 9 32 18 61 50
23 9 18 32 61 50
if low>=high:
stop;
Quick Sort on Left side
Quick Sort on Right side
23 9 18
18 9 23
61 50
50 61
9 18 23
9
23
50
61
1
2
3
4
5
6
Final Sorted Array:
9 18 23 32 50 61

QUICKSORT

Best	Average	Worst
$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Recursion | Divide and Conquer | Array

sort (A)
1. quickSort (A, 0, n − 1)

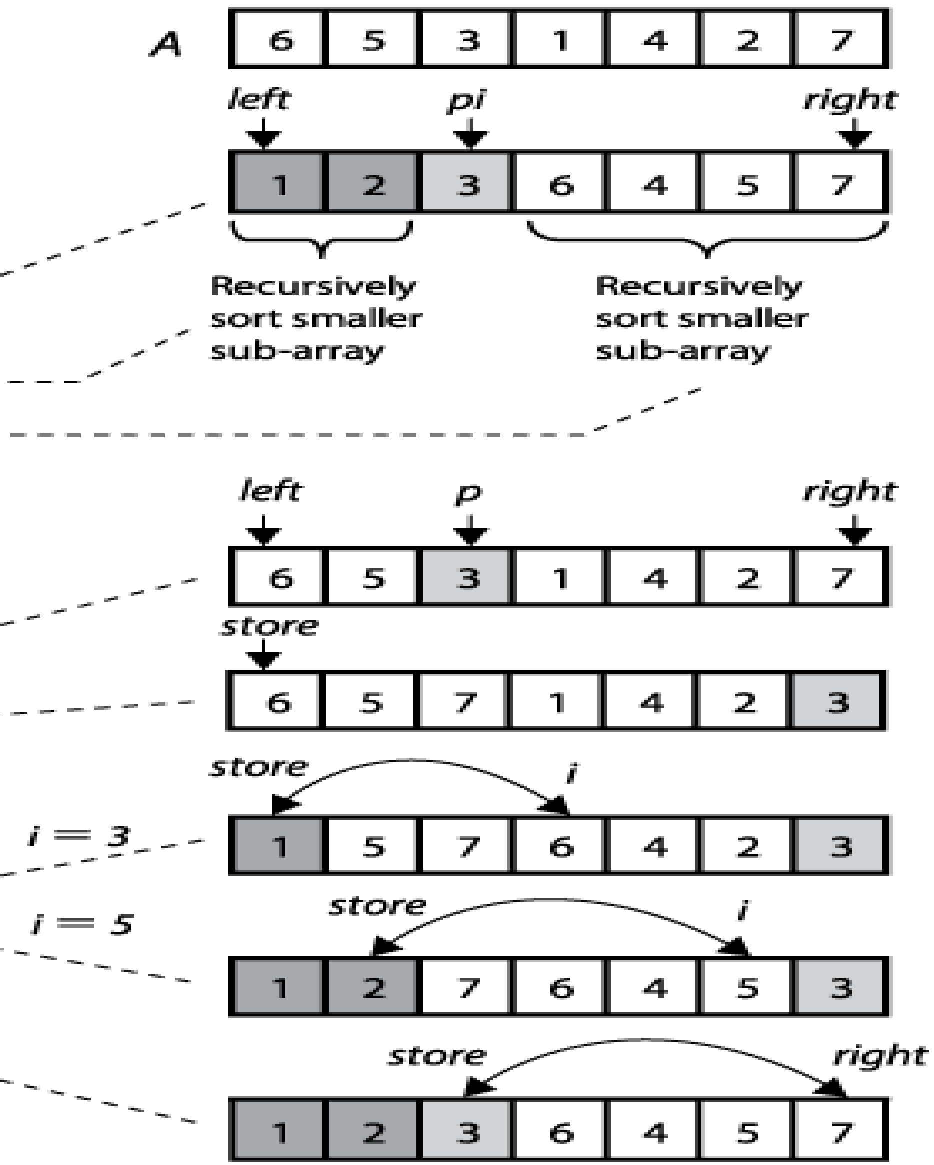
end

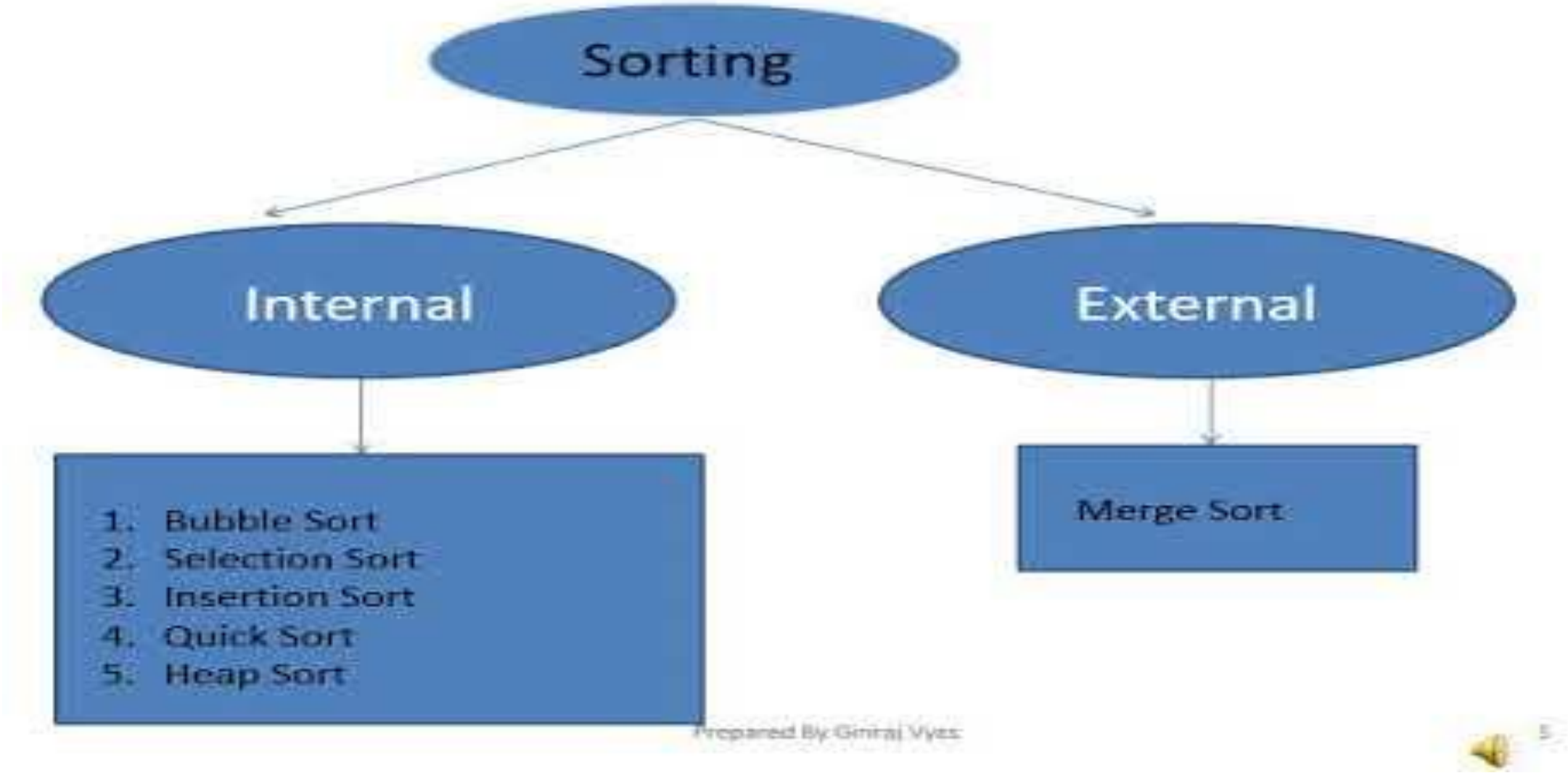
quickSort (A, left, right)
1. **if** (left < right) **then**
2. pi = partition (A, left, right)
3. quickSort (A, left, pi − 1)
4. quickSort (A, pi + 1, right)

end

partition (A, left, right)
1. p = select pivot in A[left, right]
2. swap A[p] and A[right]
3. store = left
4. **for** i = left **to** right − 1 **do**
5. **if** (A[i] ≤ A[right]) **then**
6. swap A[i] and A[store]
7. store++
8. swap A[store] and A[right]
9. **return** store

end




Sorting
Internal
External
1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Quick Sort
5. Heap Sort
Merge Sort
Prepared By Giriraj Vyas