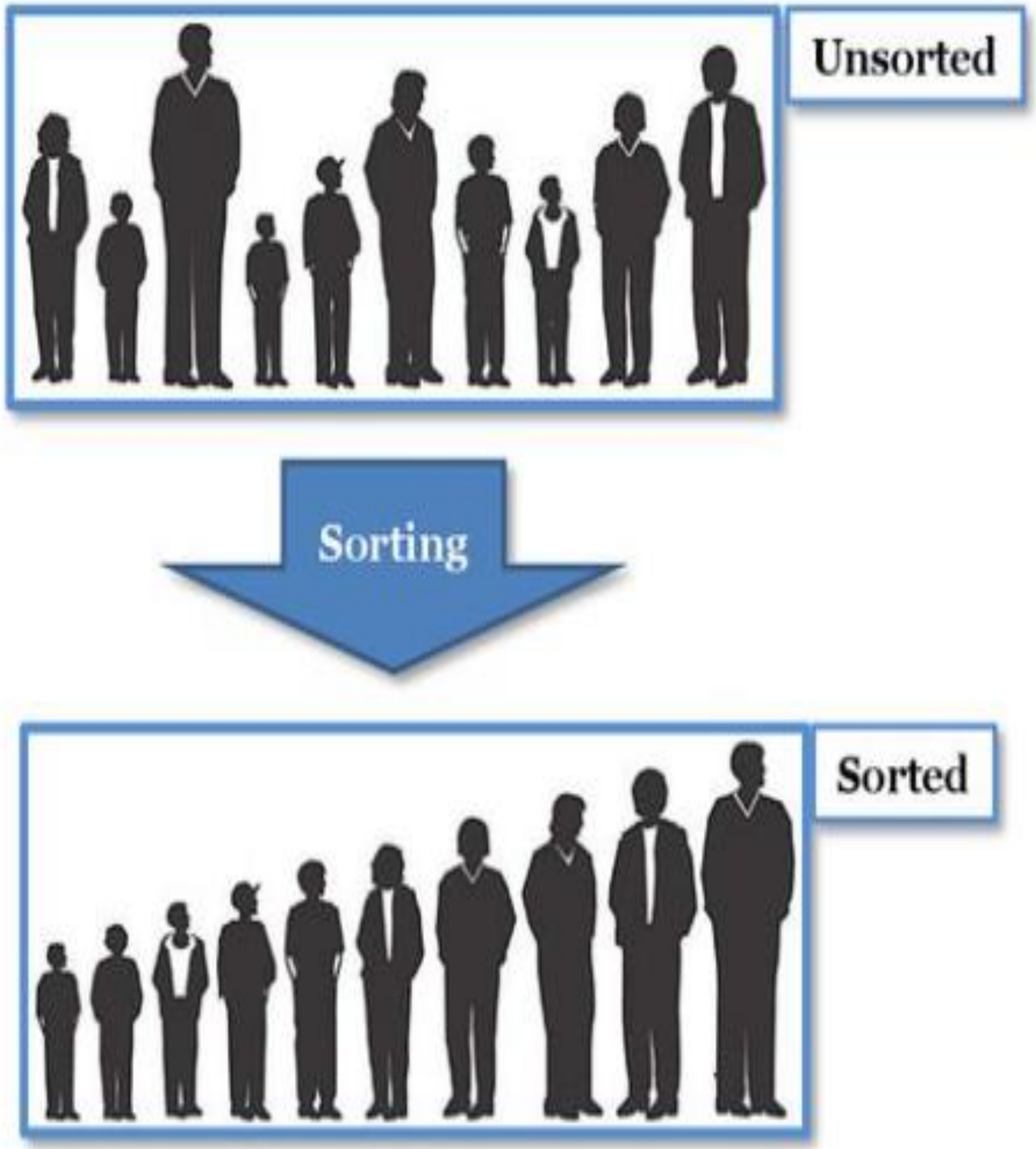
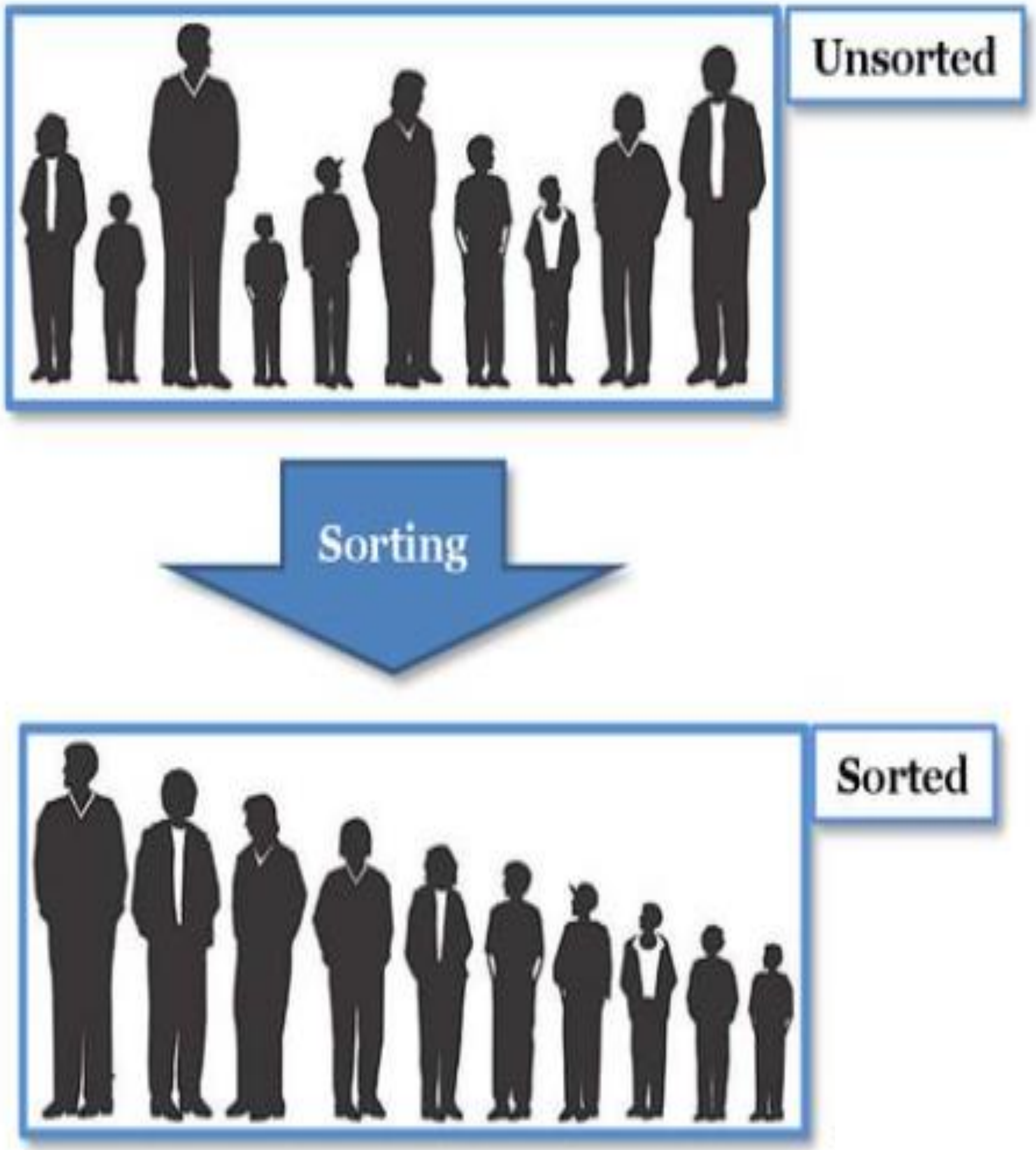
Sep22 : Day 4

Kiran Waghmare

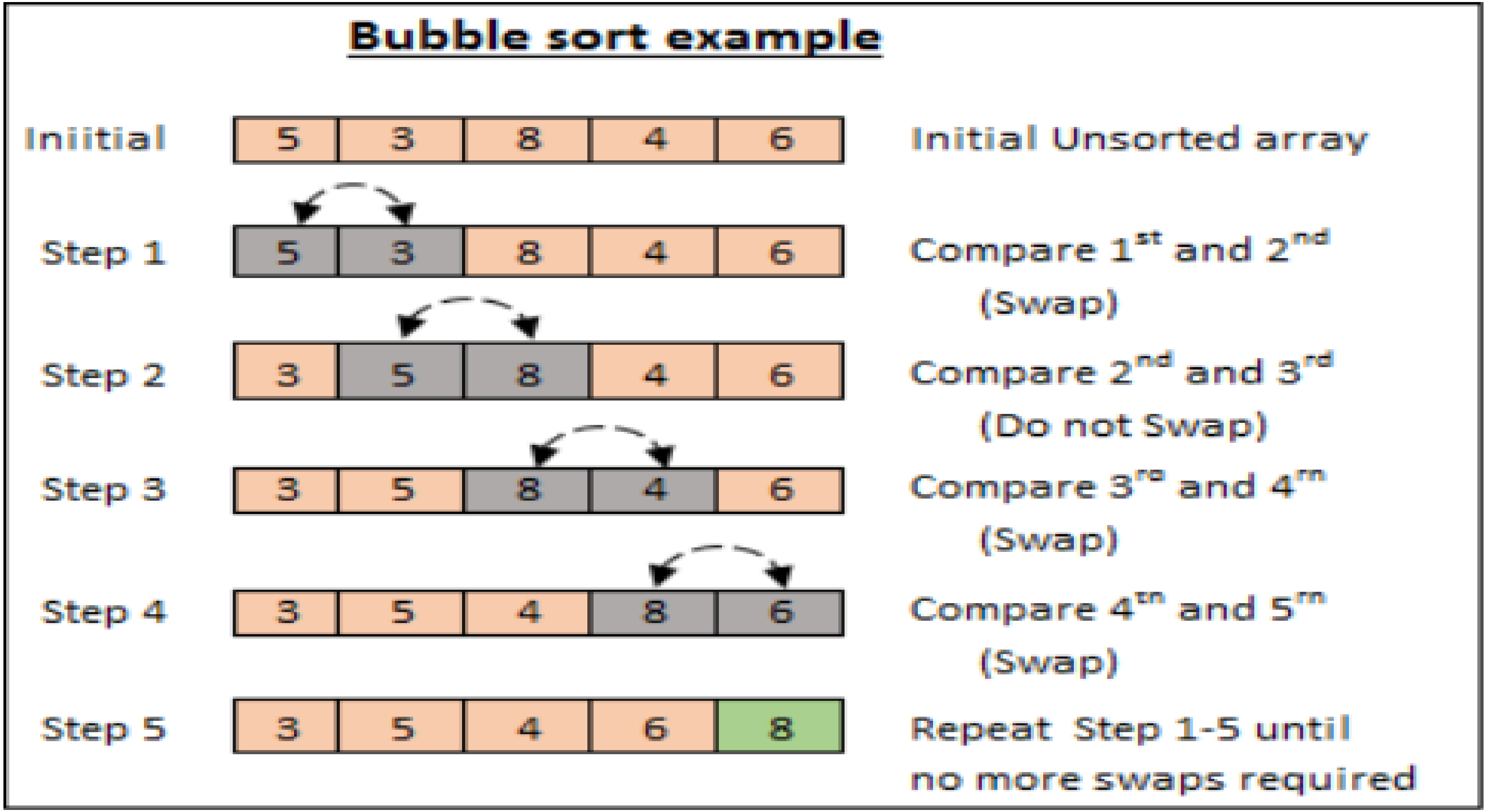
CDAC Mumbai


Unsorted
Sorting
Sorted


Unsorted
Sorting
Sorted

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
Iniitial
5 3 8 4 6
Initial Unsorted array
Step 1
5 3 8 4 6
Compare 1st and 2nd
(Swap)
Step 2
3 5 8 4 6
Compare 2nd and 3rd
(Do not Swap)
Step 3
3 5 8 4 6
Compare 3rd and 4th
(Swap)
Step 4
3 5 4 8 6
Compare 4th and 5th
(Swap)
Step 5
3 5 4 6 8
Repeat Step 1-5 until
no more swaps required

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
```

TIME COMPLEXITY

- The time complexity for bubble sort is calculated in terms of the number of comparisons f(n) (or of number of loops)

- Here two loops(outer loop and inner loop) iterates(or repeated) the comparison.

- The inner loop is iterated one less than the number of elements in the list (i.e., n-1 times) and is reiterated upon every iteration of the outer loop

$$f(n) = (n - 1) + (n - 2) + \ldots\ldots + 2 + 1$$

$$= n(n - 1) = O(n2).$$

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.

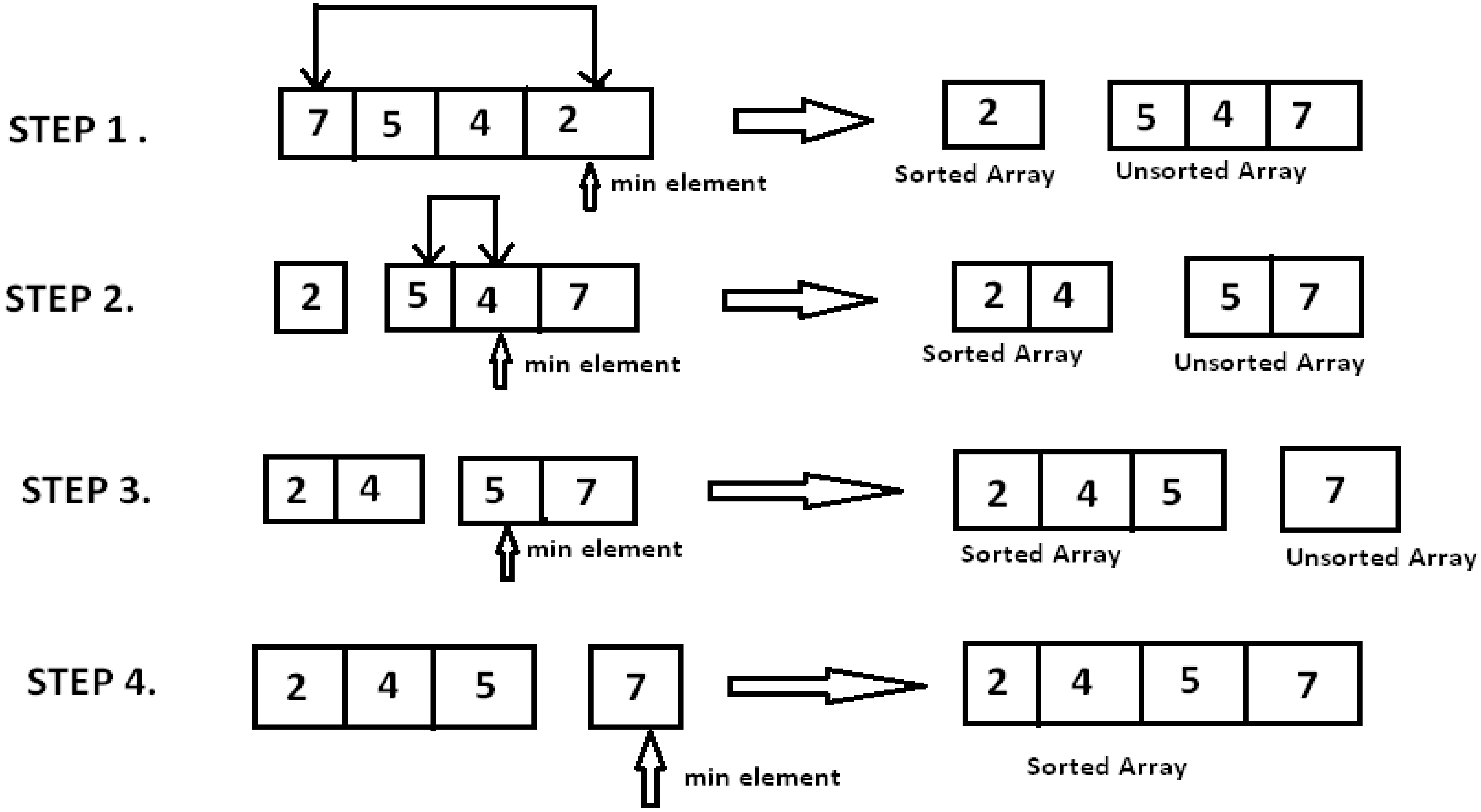
- Let a[n] be a linear array of n elements. The selection sort works asfollows:

- 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]


STEP 1 .
7
5
4
2
min element
2
Sorted Array
5
4
7
Unsorted Array
STEP 2.
2
5
4
7
min element
2
4
Sorted Array
5
7
Unsorted Array
STEP 3.
2
4
5
7
min element
2
4
5
Sorted Array
7
Unsorted Array
STEP 4.
2
4
5
7
min element
2
4
5
7
Sorted Array

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]);
}
```

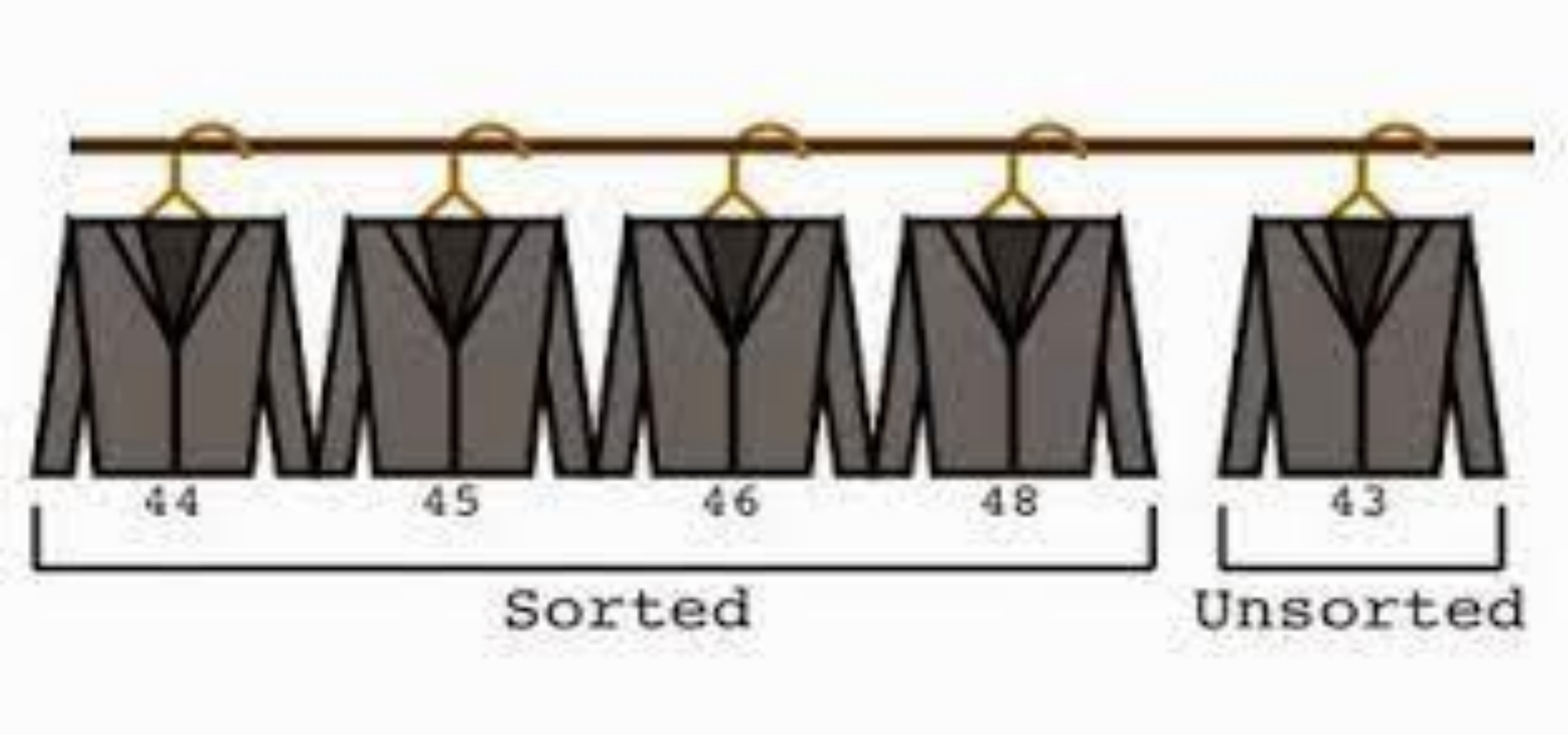
Time Complexity

- 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.

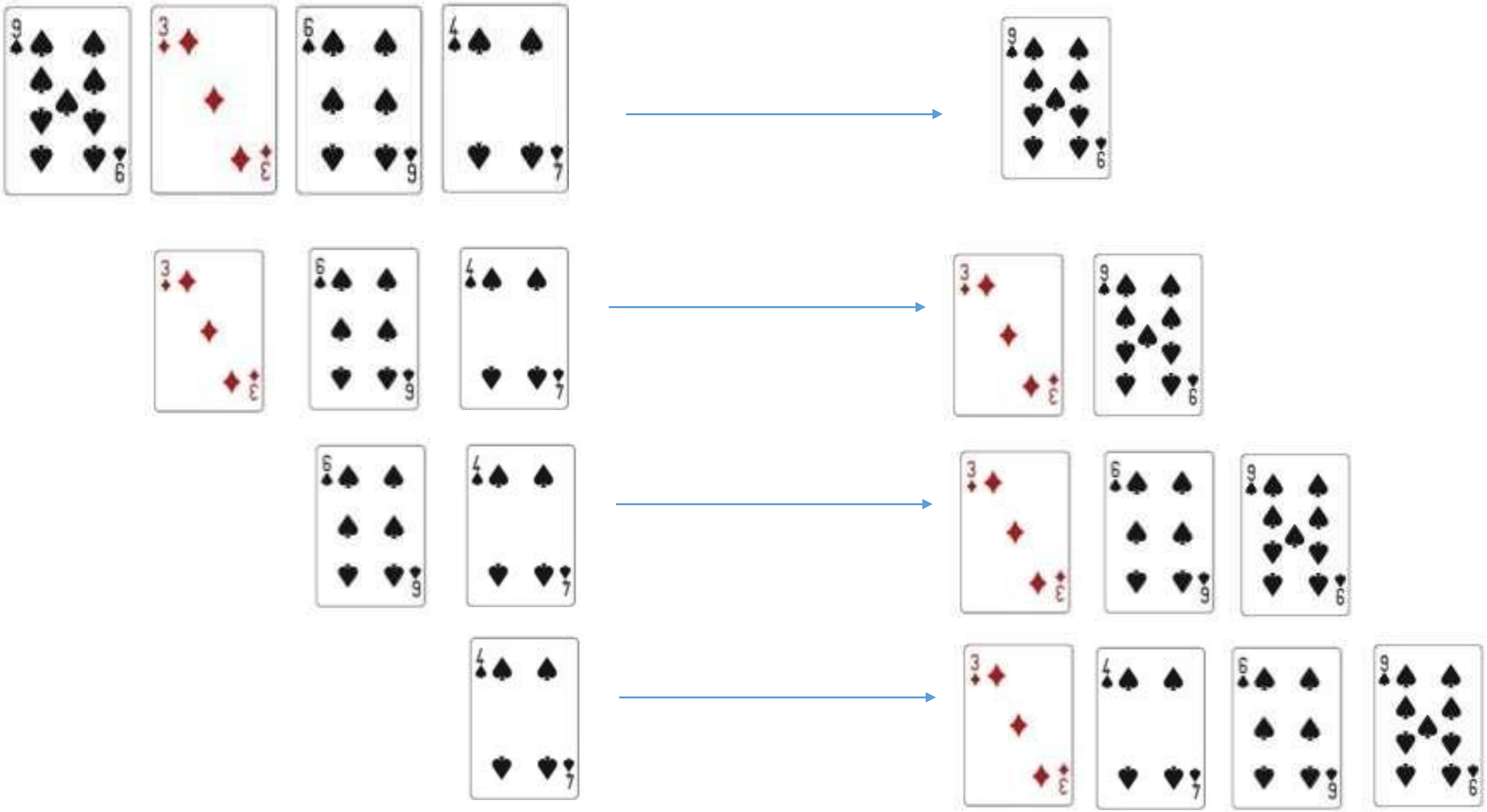
- Pick one card at a time from the table, and insert it into the correct position in the left hand.

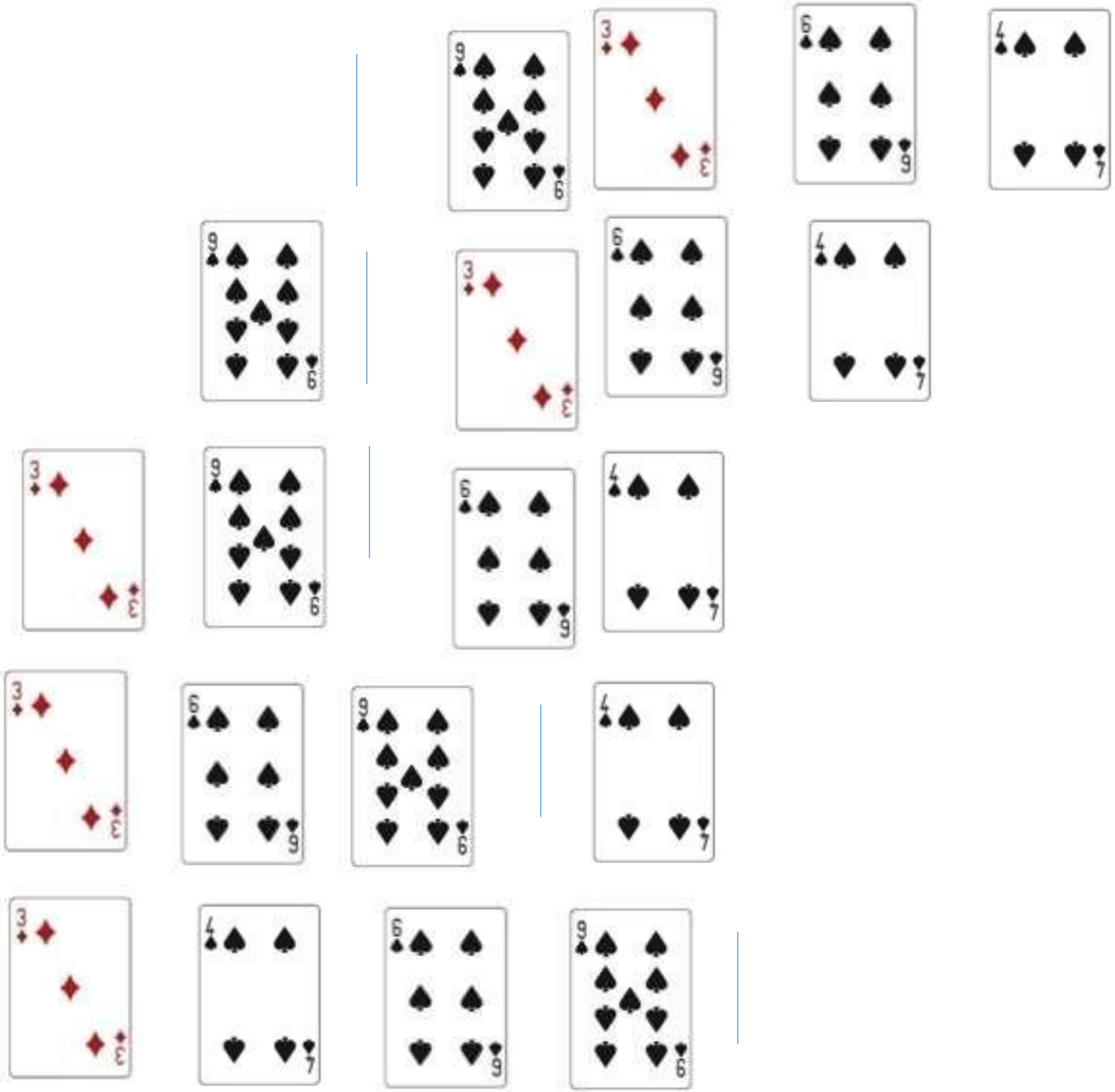
- Compare it with each of the cards already in the hand, from right to left

- 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(*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

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

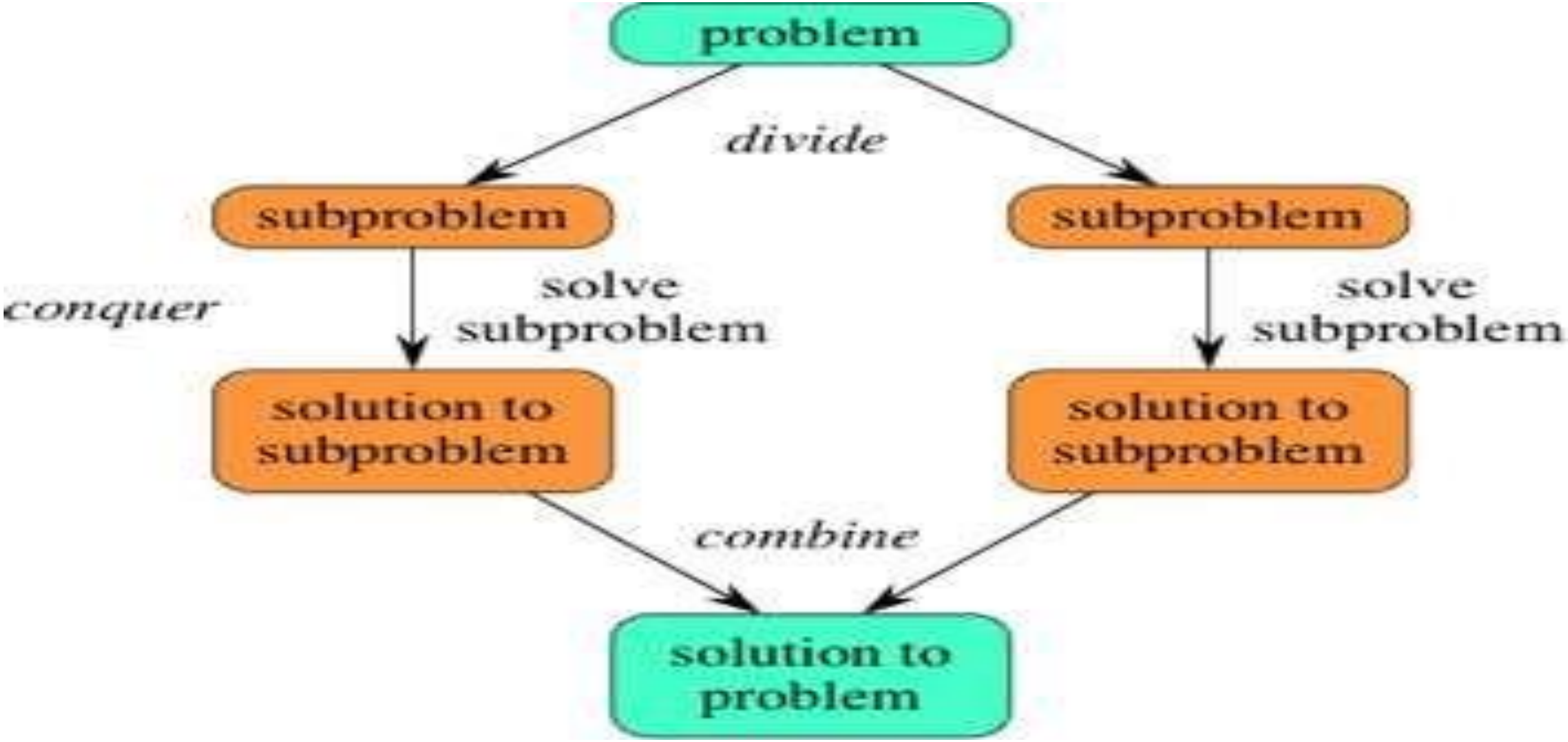
Divide and conquer algorithms

- 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.

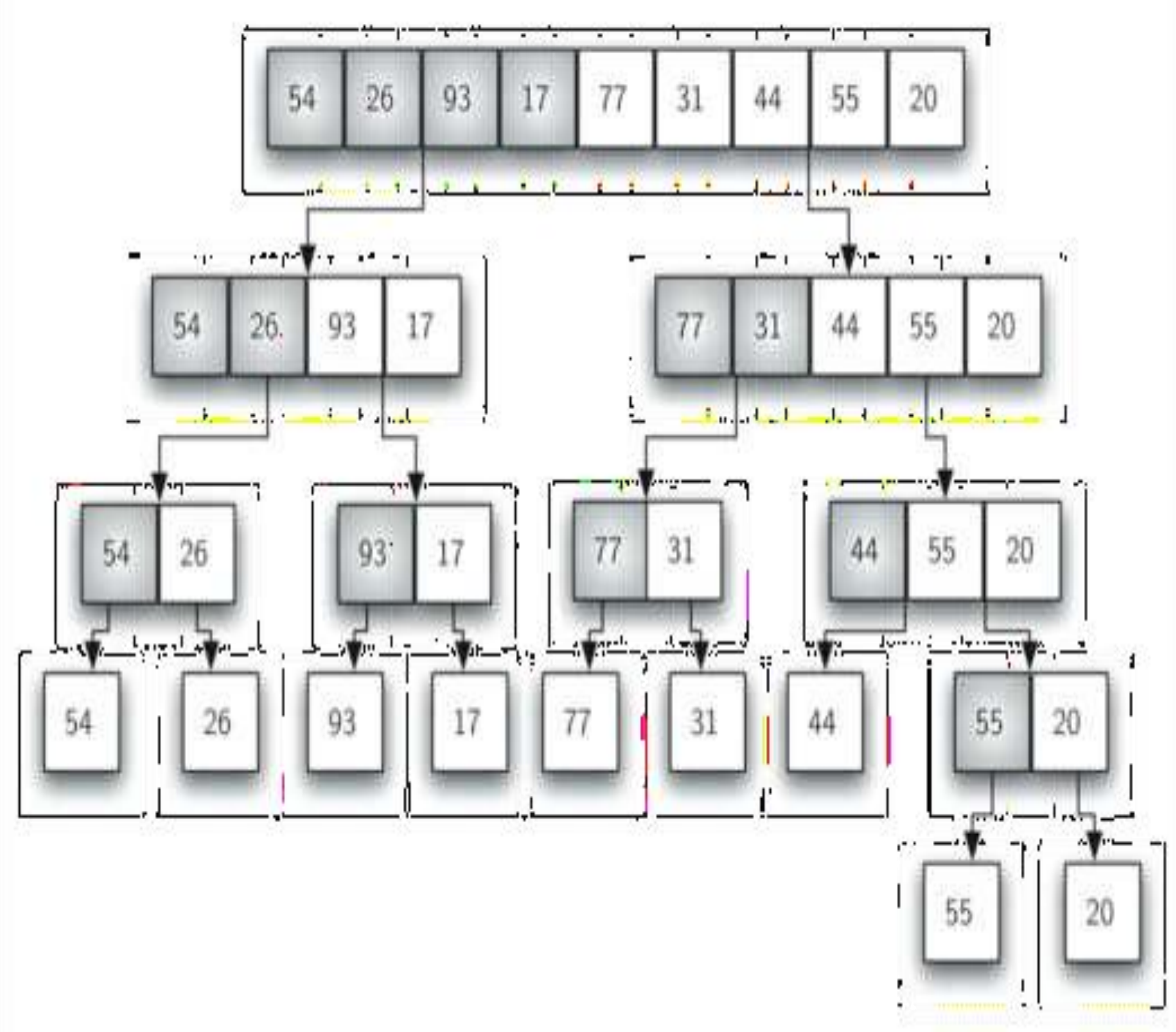
- With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

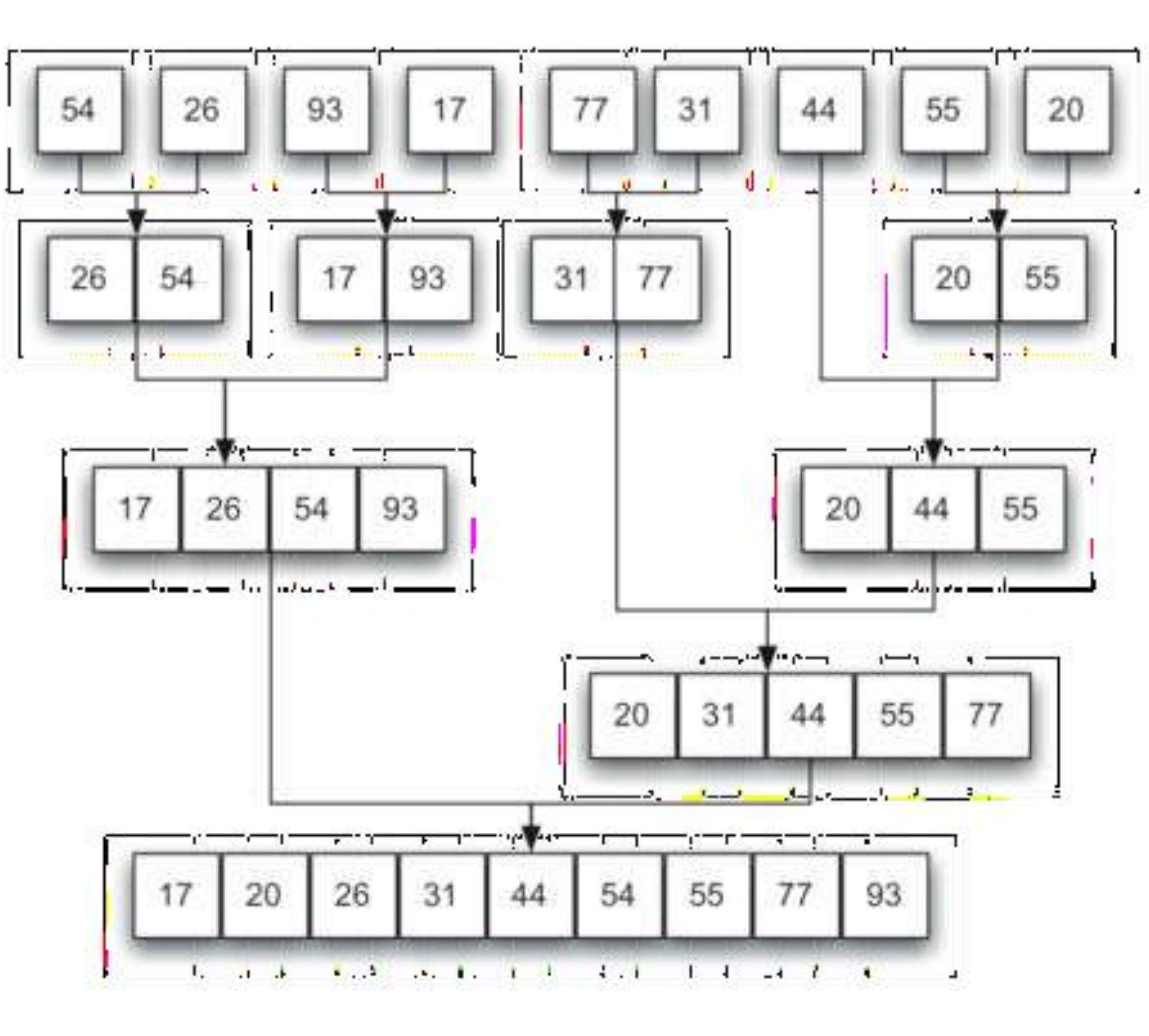
Merge Sort

- 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]


54 26 93 17 77 31 44 55 20
54 26 93 17
77 31 44 55 20
54 26
93 17
77 31
44 55 20
54
26
93
17
77
31
44
55 20
55
20


54 26 93 17 77 31 44 55 20
26 54
17 93
31 77
20 55
17 26 54 93
20 44 55
20 31 44 55 77
17 20 26 31 44 54 55 77 93

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:

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

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

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

```
Merge (A, p, q, r)
    n1 ← (q – p) + 1
    n2 ← (r – q)
    create arrays L[1..n1+1] and R[1..n2+1]
    for i ← 1 to n1 do
        L[i] ← A[(p + i) -1]
    for j ← 1 to n2 do
        R[j] ← A[q + j]
    L[n1 + 1] ← ∞
    R[n2 + 1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r do
        count ← count + 1
        if L[I] <= R[j]
            then  A[k] ← L[i]
                  i ←i + 1
            else A[k] ← R[j]
                  j ← j + 1
```

Analysis of merge Sort

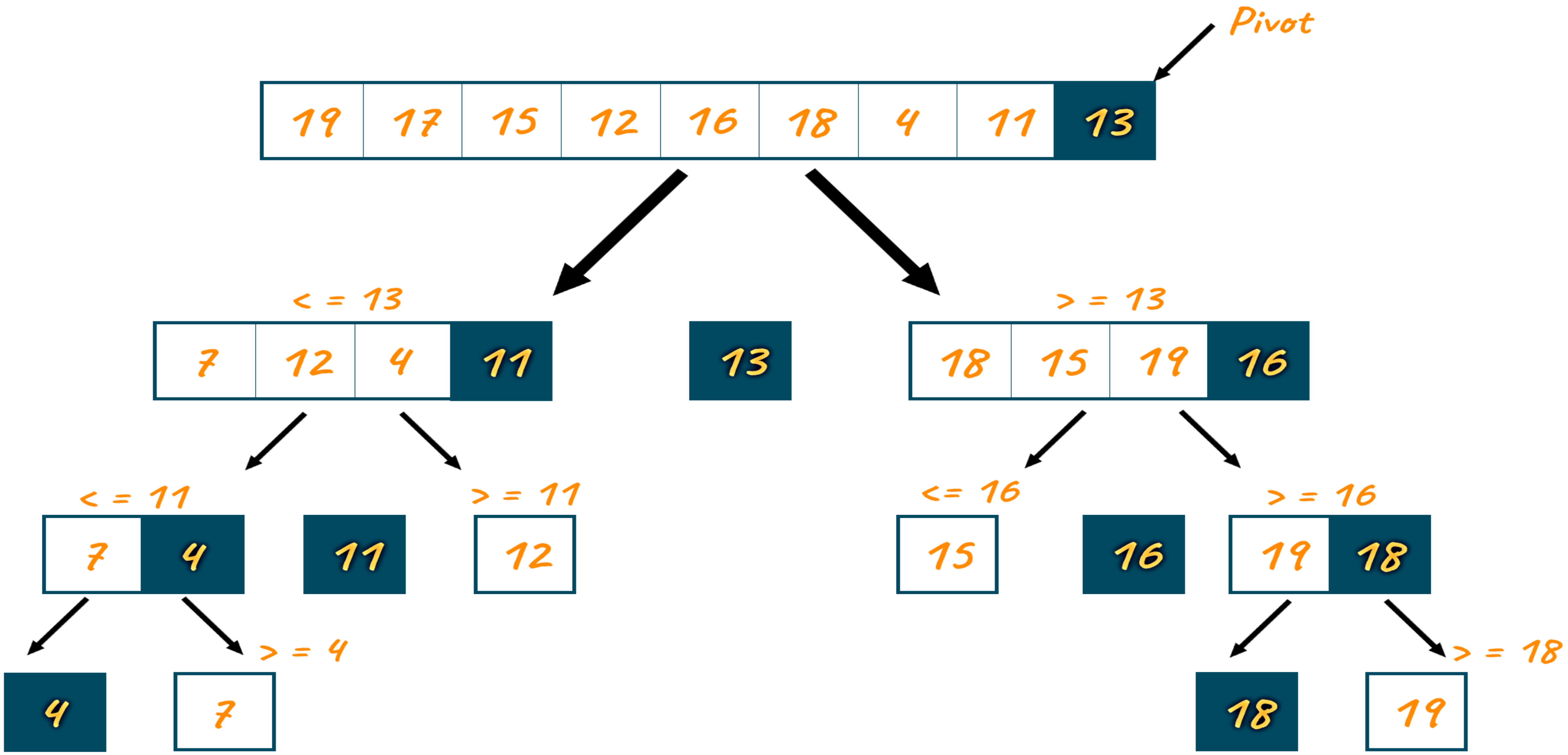
- We can view merge sort as creating a tree of calls, where each level of recursion is a level in the tree.

- 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

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.


Pivot
19 17 15 12 16 18 4 11 13
< = 13
7 12 4 11
13
> = 13
18 15 19 16
< = 11
7 4
11
> = 11
12
<= 16
15
16
> = 16
19 18
> = 4
4
7
18
> = 18
19

Quick Sort

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

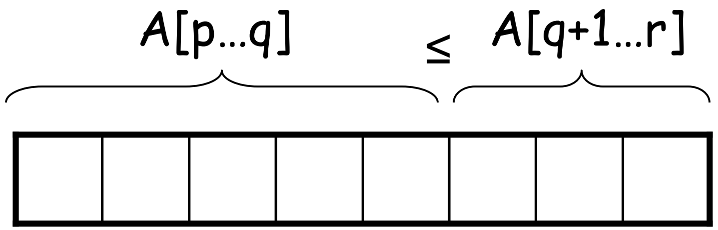
- Choose an element pivot=a[n-1]. Suppose that elements of an array a are partitioned so that pivot is placed into position I and the following condition hold:
  - Each elements in position 0 through i-1 is less than or equal to pivot
  - Each of the elements in position i+1 through n-1 is greater than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

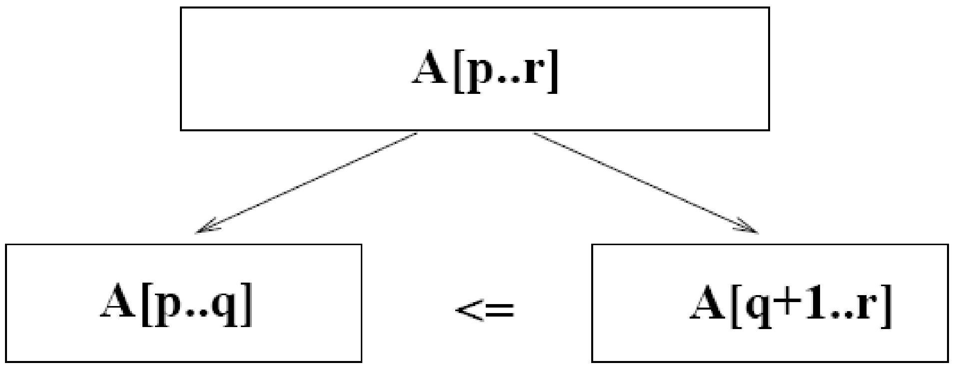
Algorithm

- Choosing a pivot
  - To partition 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

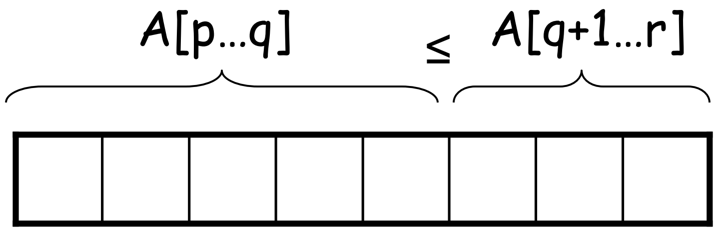
Quicksort



- Sort an array $A[p \ldots r]$
- **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

The following procedure implements quicksort:

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    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)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```