Marwadi University

Department of Computer Engineering

Sorting & Searching

Unit#5

Data Structure
01CE0301 / 3130702

Ravikumar Natarajan

Highlights

- Sorting
  - Bubble Sort
  - Selection Sort
  - Quick Sort
  - Merge Sort
- Searching
  - Sequential Search and
  - Binary Search

Sorting

- Sorting Algorithms are methods of reorganizing a large number of items into some specific order such as highest to lowest, or vice-versa, or even in some alphabetical order.
- These algorithms take an input list, processes it (i.e, performs some operations on it) and produce the sorted list.
- By sorting data, it is easier to search through it quickly and easily.

- The simplest example of sorting is a **dictionary, telephone directory**.

Sorting

- Two categories in sorting,
    - Internal sorting
        - Takes place in the main memory of the computer.
    - External sorting
        - Takes place in secondary storage.

Bubble Sort

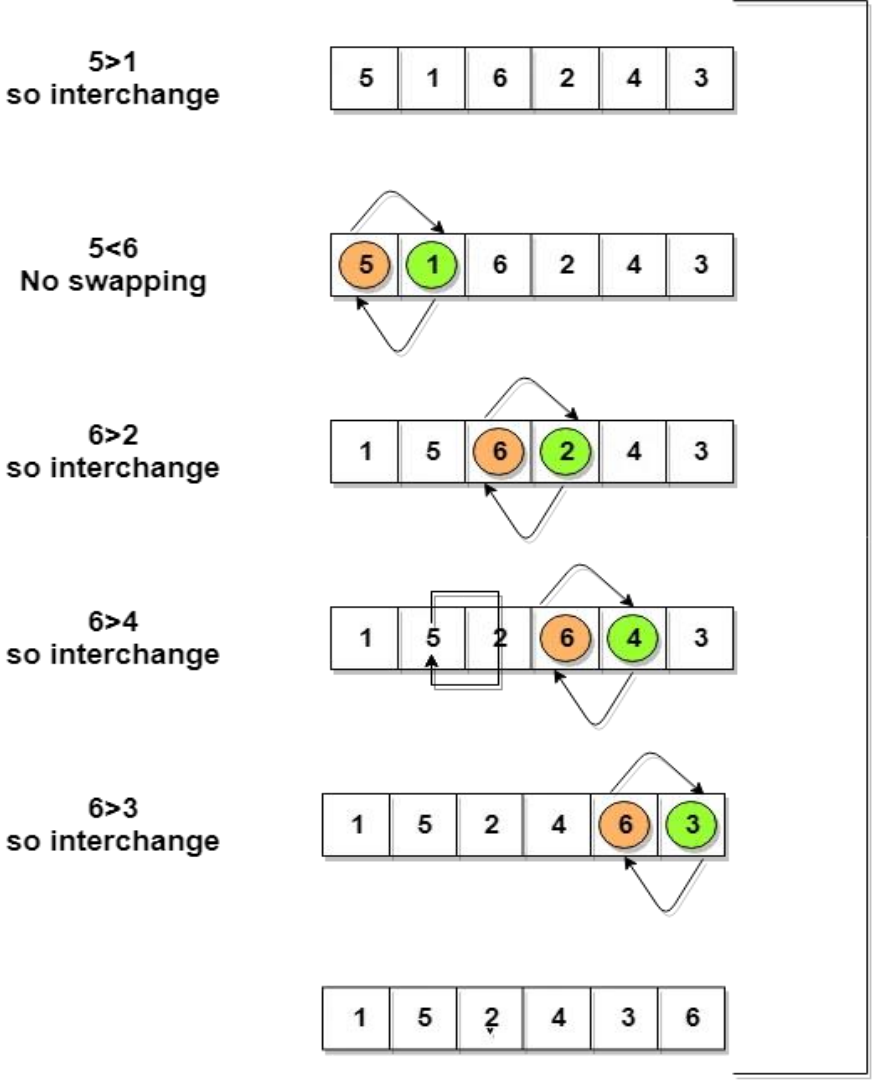
- Bubble sort, also referred to as comparison sort.
- It is the easiest and simplest of all the sorting algorithms. It works on the principle of repeatedly swapping adjacent elements in case they are not in the right order.

- In simpler terms, if the input is to be sorted in ascending order, the bubble sort will first compare the first two elements in the array.
- In case the second one is smaller than the first, it will swap the two, and move on to the next element, and so on.

Bubble Sort

- **If we have total n elements, then we need to repeat this process for n-1 times.**
- With every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Bubble Sort



5>1
so interchange

5 1 6 2 4 3

5<6
No swapping

5 1 6 2 4 3

6>2
so interchange

1 5 6 2 4 3

6>4
so interchange

1 5 2 6 4 3

6>3
so interchange

1 5 2 4 6 3

1 5 2 4 3 6

This is first insertion

similarly, after all the
iterations, the array
gets sorted

- So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.
- Similarly after the second iteration, 5 will be at the second last index, and so on..

Bubble Sort

15, 16, 6, 8, 5

```
for(i=0; i<n-1; i++)
{
   for(j=0; j<n-1; j++)
    {
       if(a[j] > a[j+1])
        {
           temp = a[j];
           a[j] = a[j+1];
           a[j+1] = temp;
          }
       }
   }
```

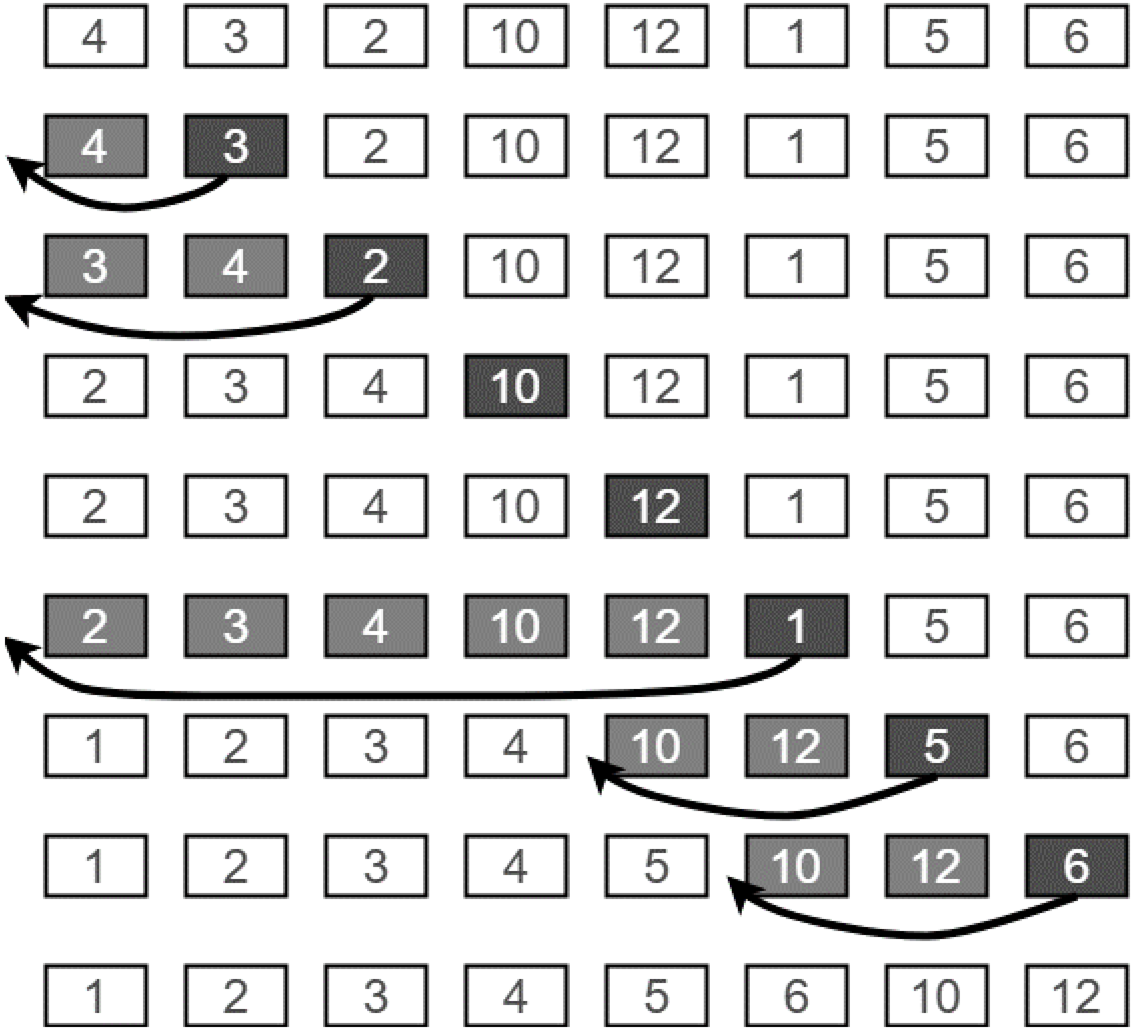
Bubble Sort

- **Time complexity**
- Best case: Omega (n)
- Average case: Theta (n^2)
- Worst case: Big O (n^2)

Insertion Sort

- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- The array is virtually split into a sorted and an unsorted part.
- Values from the unsorted part are picked and placed at the correct position in the sorted part.

Insertion Sort



4 3 2 10 12 1 5 6

4 3 2 10 12 1 5 6

3 4 2 10 12 1 5 6

2 3 4 10 12 1 5 6

2 3 4 10 12 1 5 6

2 3 4 10 12 1 5 6

1 2 3 4 10 12 5 6

1 2 3 4 5 10 12 6

1 2 3 4 5 6 10 12

Insertion Sort

5, 4, 10, 1, 6, 2

```
for (i = 1 ; i <= n - 1; i++)
  {
         j = i;
      while ( j > 0 && arr[j-1] > arr[j])
      {
        temp    = arr[j];
        arr[j]  = arr[j-1];
        arr[j-1] = temp;
        j--;
      }
  }
```

Insertion Sort

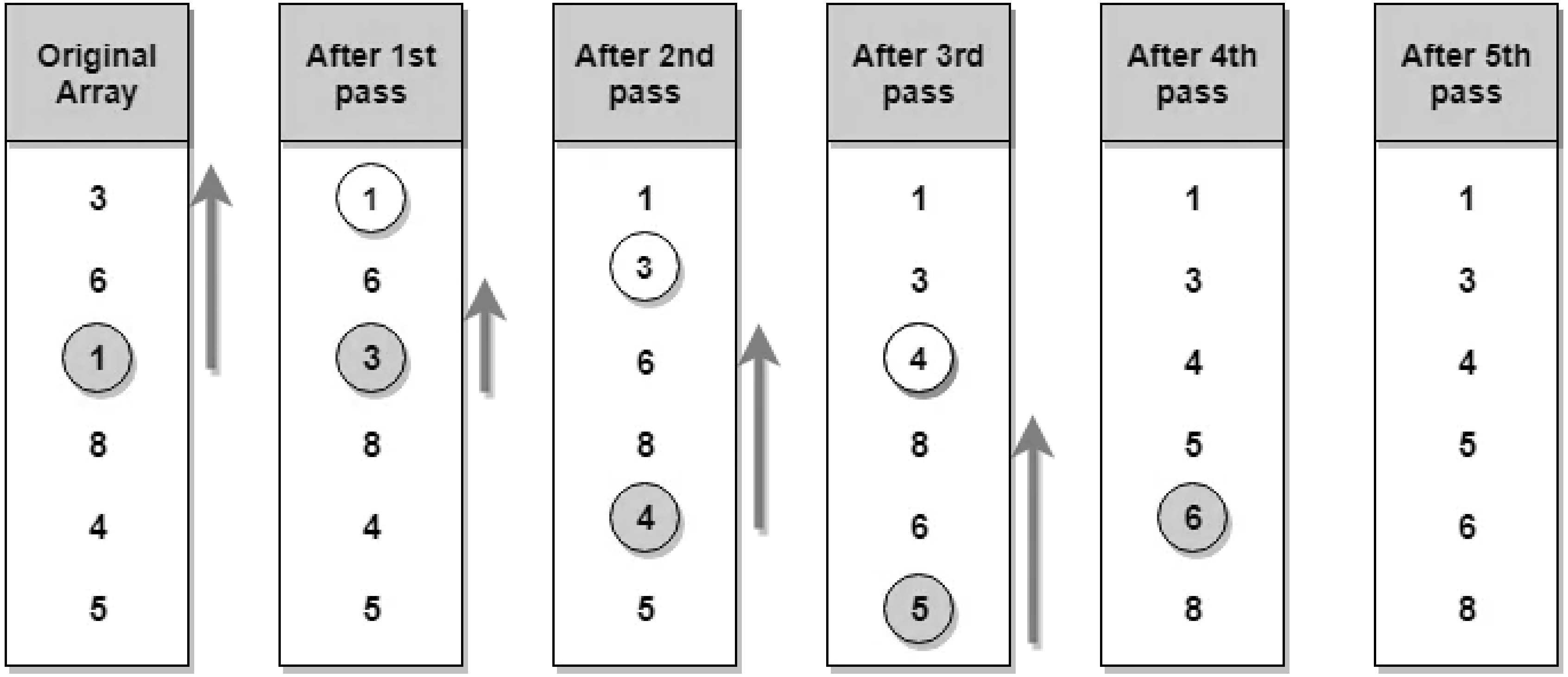
- **Time complexity**
- Best case: Omega (n)
- Average case: Theta (n^2)
- Worst case: Big O (n^2)

Selection Sort

- Selection sort is conceptually the most simplest sorting algorithm.
- This algorithm will first find the smallest element in the array and swap it with the element in the first position, then it will find the second smallest element and swap it with the element in the second position, and it will keep on doing this until the entire array is sorted.

- It is called selection sort because it repeatedly selects the next-smallest element and swaps it into the right place.

Selection Sort



Selection Sort

9, 3, 1, 4, 2, 7, 5

```
for (i = 0 ; i <= n - 1; i++)
  {
       min = i;
      for ( j > i+1; j<n; j++)
      {
        if(a[j] < a[min])
          {
            min = j;
          }
        }
        temp     = a[i]; //FirstIndex
        a[i]   = a[min]; //SecondIndex
        a[min] = temp;
  }
```

Selection Sort

- **Time complexity**
- Best case: Omega (n^2)
- Average case: Theta (n^2)
- Worst case: Big O (n^2)

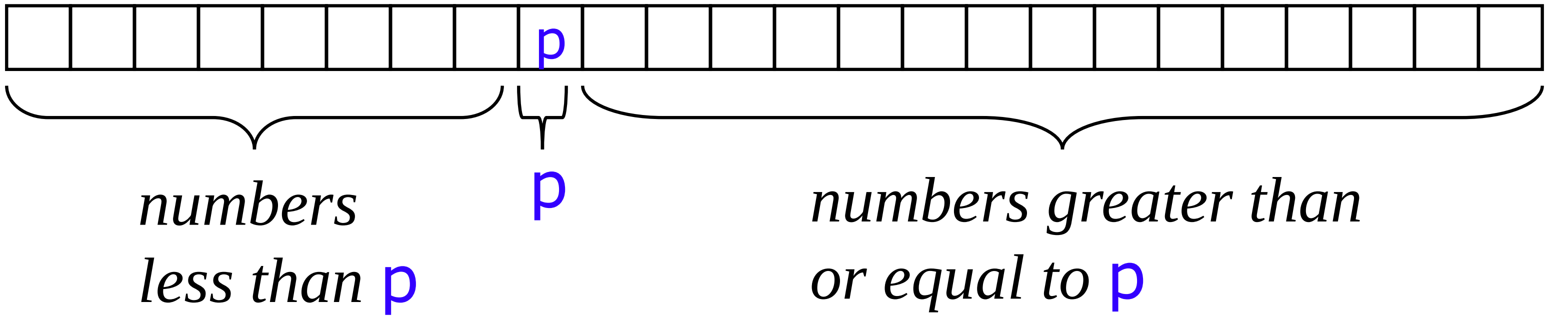
Quick Sort

- QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.
- There are many different versions of quickSort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.
- The key process in quickSort is partition().

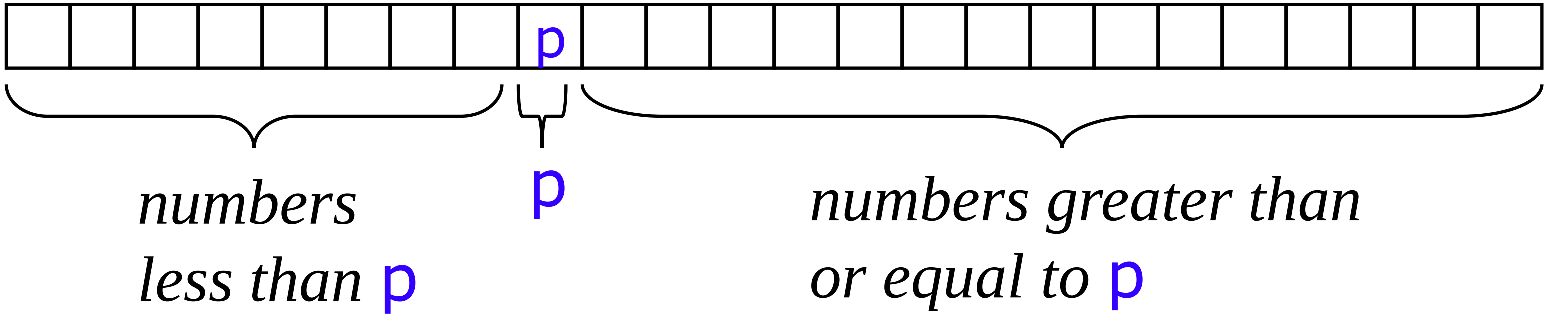
Quick Sort

- **Partitioning**
- A key step in the Quicksort algorithm is partitioning the array
- We choose some (any) number p in the array to use as a pivot
- We partition the array into three parts:



Quick Sort

- **Partitioning**
- Choose an array value (say, the first) to use as the pivot
    - Starting from the left end, find the first element that is greater than or equal to the pivot
    - Searching backward from the right end, find the first element that is less than the pivot
- Interchange (swap) these two elements
- Repeat, searching from where we left off, until done



Quick Sort

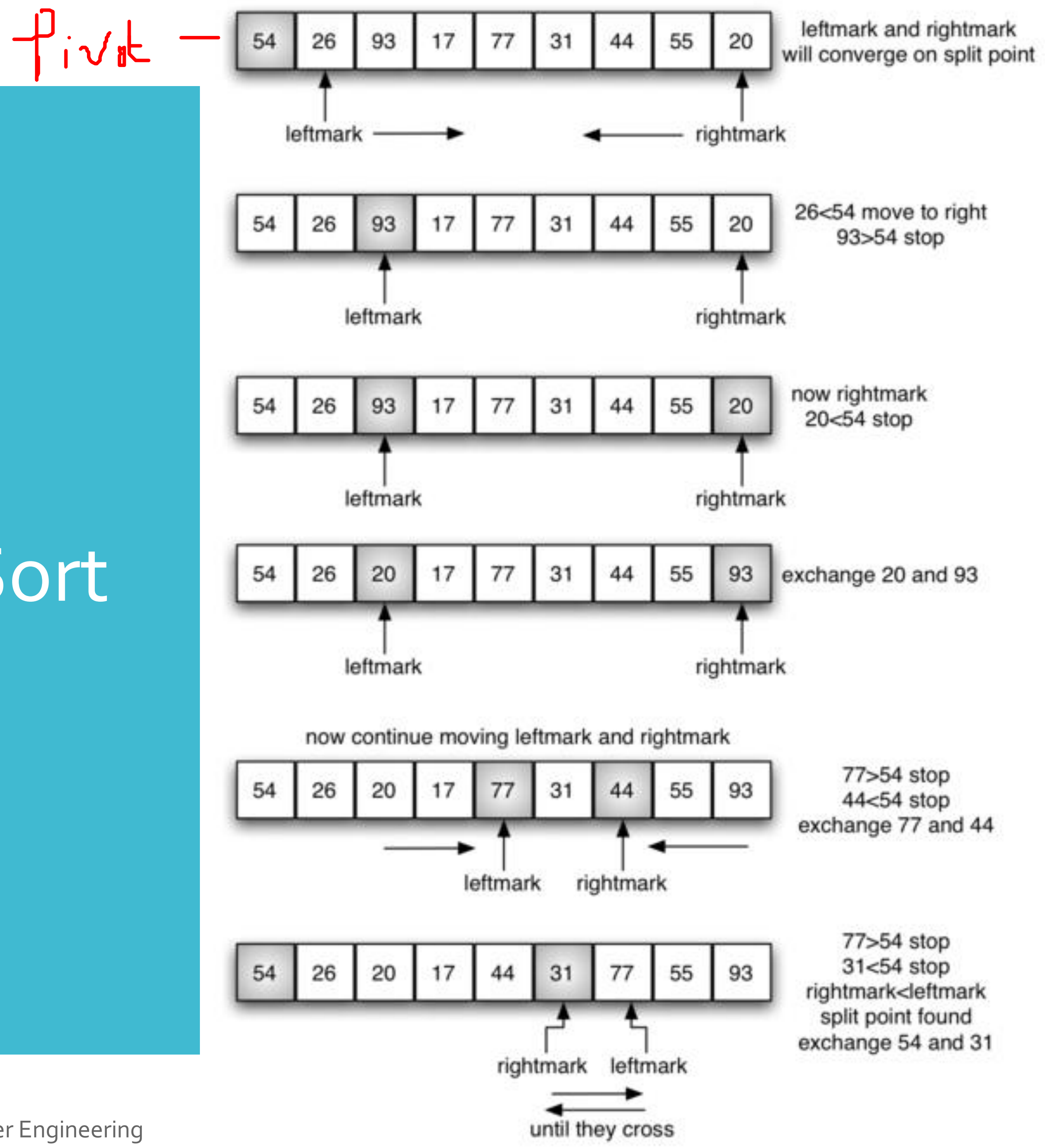
```
void quickSort(int list[10],int first,int last){
   int pivot,i,j,temp;

    if(first < last){
       pivot = first;
       i = first;
       j = last;

       while(i < j){
          while(list[i] <= list[pivot] && i < last)
             i++;
          while(list[j] && list[pivot])
             j--;
          if(i < j){
              temp = list[i];
              list[i] = list[j];
              list[j] = temp;
          }
       }
```

```
  temp = list[pivot];
        list[pivot] = list[j];
        list[j] = temp;
        quickSort(list,,j,first-1);
        quickSort(list,j+1,last);
     }
}
```

Quick Sort



Pivot —

54 26 93 17 77 31 44 55 20

leftmark and rightmark will converge on split point

leftmark → ← rightmark

54 26 93 17 77 31 44 55 20

26<54 move to right
93>54 stop

leftmark rightmark

54 26 93 17 77 31 44 55 20

now rightmark
20<54 stop

leftmark rightmark

54 26 20 17 77 31 44 55 93

exchange 20 and 93

leftmark rightmark

now continue moving leftmark and rightmark

54 26 20 17 77 31 44 55 93

77>54 stop
44<54 stop
exchange 77 and 44

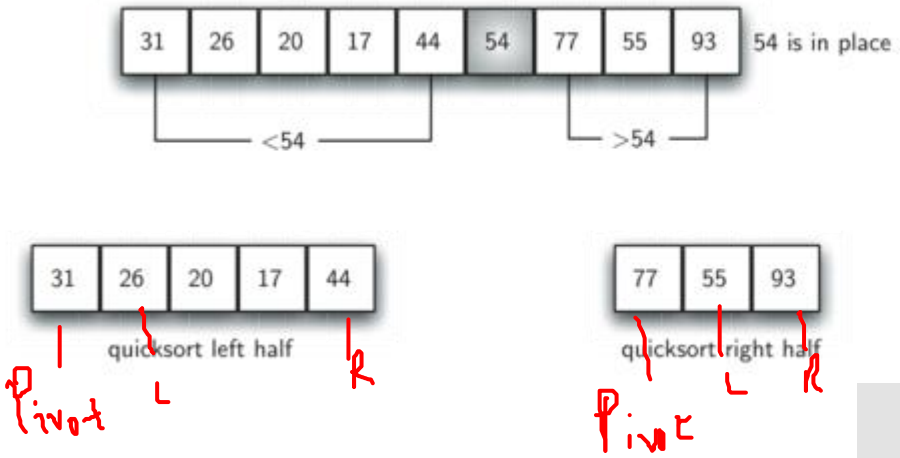
leftmark rightmark

54 26 20 17 44 31 77 55 93

77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31

rightmark leftmark

until they cross



31 26 20 17 44 54 77 55 93

54 is in place

<54 >54

31 26 20 17 44

quicksort left half

Pivot L R

77 55 93

quicksort right half

Pivot L R

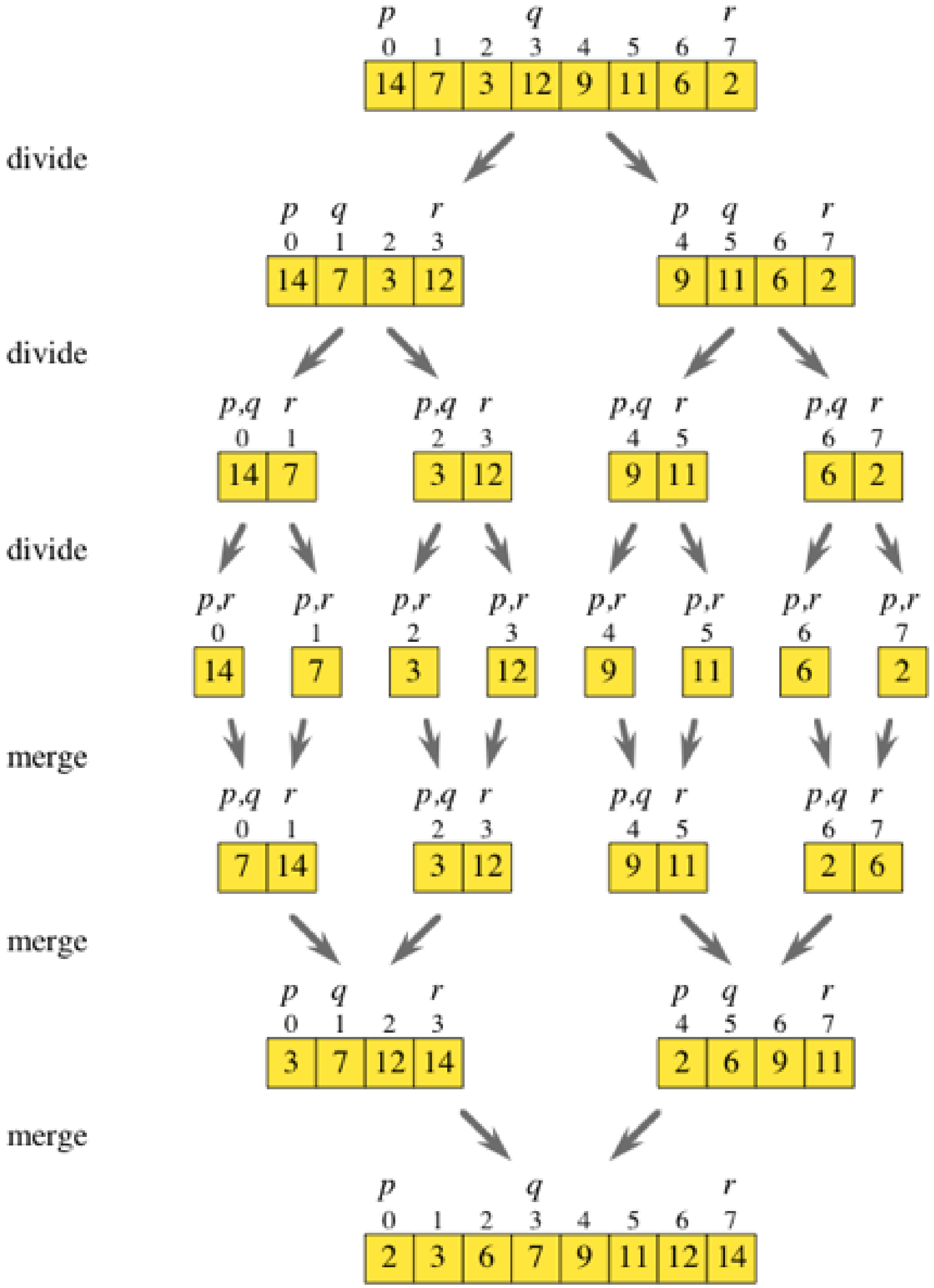
Quick Sort

- **Time complexity**
- Best case: Omega (n log(n))
- Average case: Theta (n log(n))
- Worst case: Big O (n^2)

Merge Sort

- 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 is the algorithm which follows divide and conquer approach. Consider an array A of n number of elements. The algorithm processes the elements in 3 steps.
  - If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
  - Conquer means sort the two sub-arrays recursively using the merge sort.
  - Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

Merge Sort


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

Merge Sort

```
Merge(a, lb,mid, ub)
{
i=lb;
j=mid+1;
K=lb;
}
While(i<=mid && j<=ub)
{
If(a[i] <= a[j])
{
b[k]=a[i]; i++; }
Else {
B[k] = a[j]; j++ }
K++; }
If(i>mid) {
While(j<=ub) {
B[k]=a[j]; j++; k++; }}
Else{
While(i<=mid) {
B[k]=a[i]; i++; k++; }
```

```
For(k=lb; k<=ub; k++) {
A[k]=b[k];
}
```

```
mergeSort(a,lb,ub) {
If(lb<ub) {
Mid = (lb+ub)/2;
mergeSort(a, lb, mid);
mergeSort(a, mid+1, up);
merge(a, lb, mid, up);
}
```

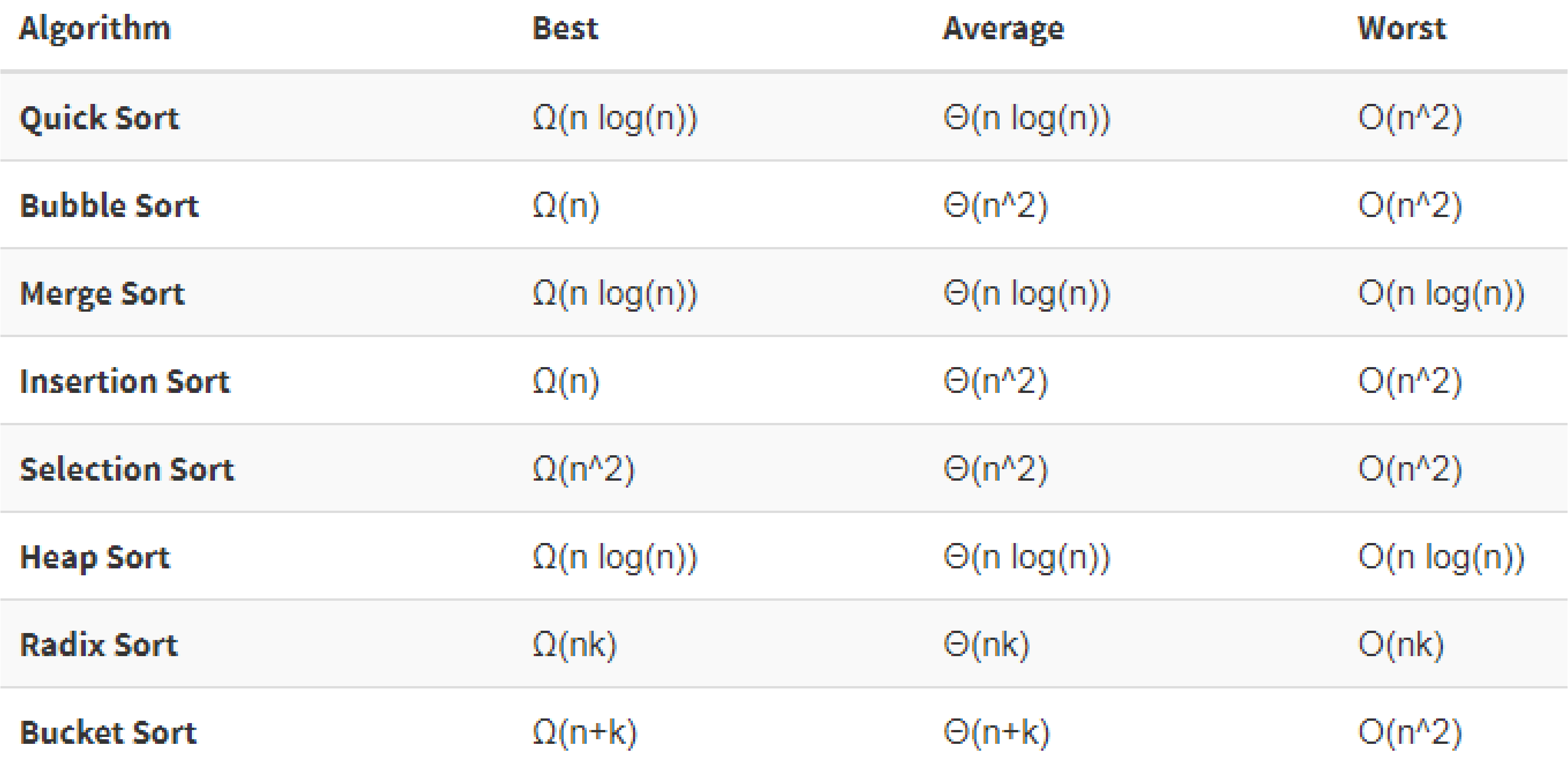
Merge Sort

15, 5 , 24, 8, 1, 3, 16, 10 ,20

Merge Sort

- **Time complexity**
- Best case: Omega (n log(n))
- Average case: Theta (n log(n))
- Worst case: Big O (n log(n))

Time Complexity of Searching Algorithms



Algorithm	Best	Average	Worst
Quick Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)
Merge Sort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Radix Sort	Ω(nk)	Θ(nk)	O(nk)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)

Thank You.