Data Structure and Algorithms

Session-21

Dr. Subhra Rani Patra
SCOPE, VIT Chennai

Quick Sort Algorithm:

✓Quick Sort is a Divide and Conquer algorithm.

✓At each step it finds 'Pivot' and then makes sure that all the smaller elements are left of 'Pivot' and all bigger elements are 'Right' of 'Pivot'.

✓It does this recursively until the entire array is sorted.

9	4	6	3	7	1	2	11	5

4	3	1	2	5	6	9	11	7

9	4	6	3	7	1	2	11	5
4	3	1	2	5	6	9	11	7
1	2	4	3	5	6	9	11	7
1	2	3	4	5	6	9	11	7

1	2	3	4	5	6	7	11	9
1	2	3	4	5	6	7	11	9
1	2	3	4	5	6	7	11	9
1	2	3	4	5	6	7	9	11

9	4	6	3	7	1	2	11	5

Quick Sort Algorithm:

```
QuickSort (A, p, q)
 if (p<q)
   r = partition(A, p, q)
   QuickSort(A,p,r-1)
   QuickSort(A,r+1, q)

Partition(A, p, q){
  pivot = q
  i=p-1
  for (j = p to q)
      if (A[j] <= A[pivot]
          increment i and then swap(A[i], [j])
```

Time & Space complexity of Quick Sort Algorithm:

```
QuickSort (A, p, q) ------------------------------------------------ T(n)
 if (p<q) ----------------------------------------------------------- O(1)
   r = partition(A, p, q) ------------------------------------------ O(n)
   QuickSort(A,p,r-1) ---------------------------------------------- T(n/2)
   QuickSort(A,r+1,p) ---------------------------------------------- T(n/2)


Partition(A, p, q){
  pivot = q ----------------------------------------------------------- O(1)
  i=p-1 --------------------------------------------------------------- O(1)
  for (j = p to q) ---------------------------- O(n)  ┐
      if (A[i] <= A[pivot] ------------------- O(1)  ├ ---------------- O(n)
         increment i and swap(A[i], [j]) ---- O(1)  ┘
```

Time Complexity - $O(n \log n)$

Space Complexity – $O(n)$

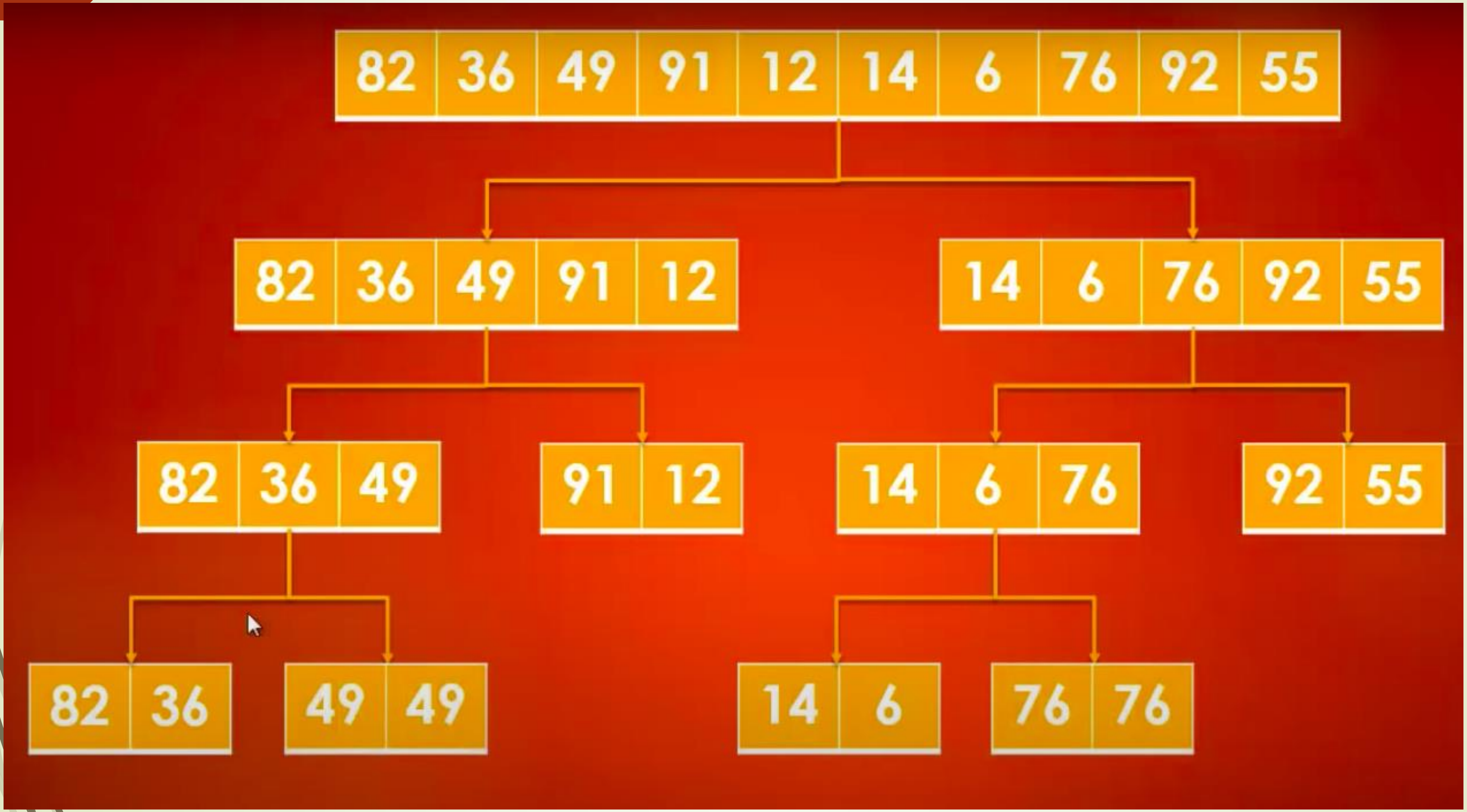
When to Use/Avoid Quick Sort:

- ✓ **When to use:**
  - ✓ When average case is desired to be $O(n \log n)$
- ✓ **When not to use:**
  - ✓ Space is a concern
  - ✓ When stable sort is required

Practical uses of Quick Sort:

- ✓ C# , Java 7 & Android:

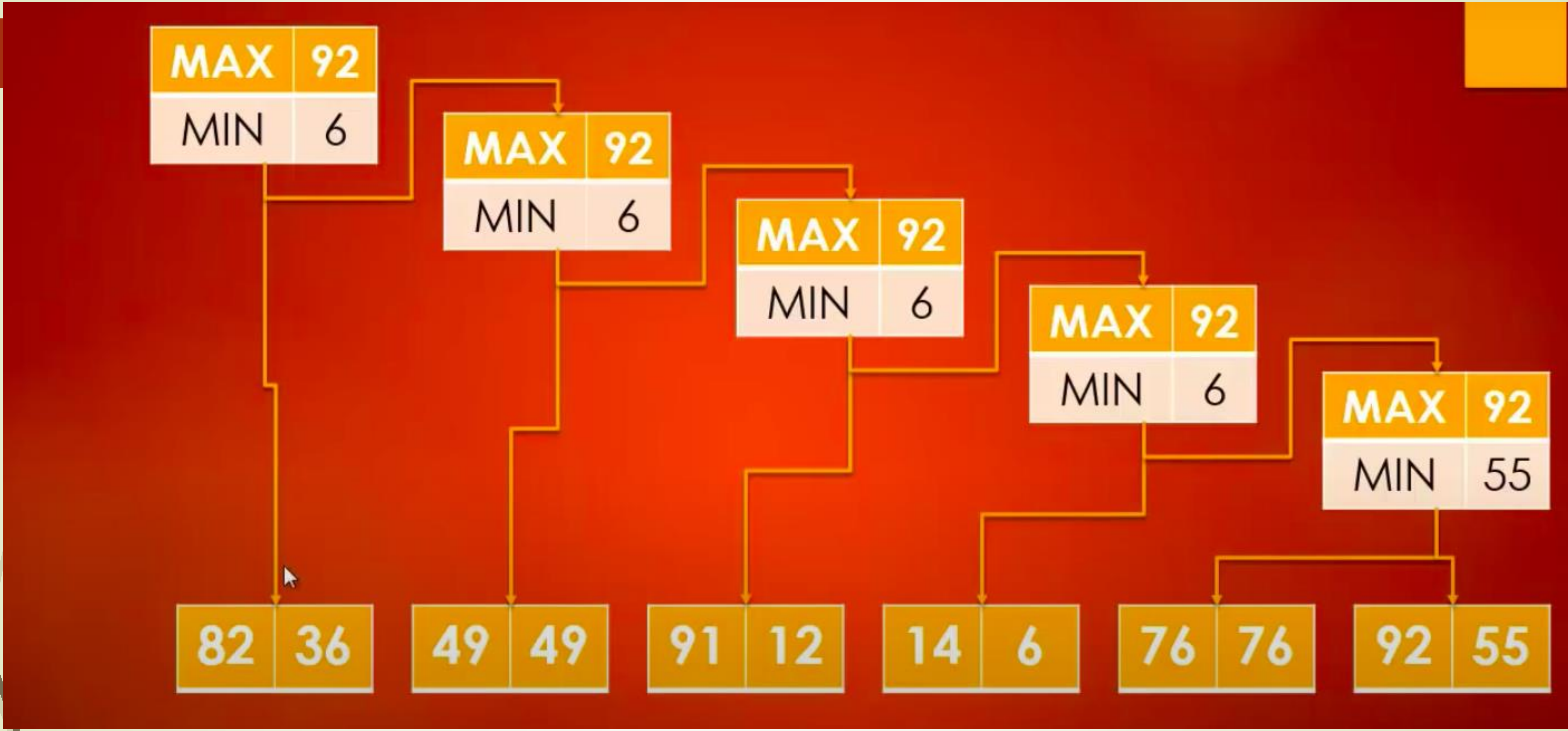
Min-Max Algorithm using Divide and Conquer




MAX 92
MIN 55
82 36
49 49
91 12
14 6
76 76
92 55

MAX
92
MIN
6
MAX
92
MIN
55
82
36
49
49
91
12
14
6
76
76
92
55


MAX 92
MIN 6
MAX 92
MIN 6
MAX 92
MIN 55
82 36
49 49
91 12
14 6
76 76
92 55



MAX 92
MIN 6
MAX 92
MIN 6
MAX 92
MIN 6
MAX 92
MIN 6
MAX 92
MIN 55
82 36
49 49
91 12
14 6
76 76
92 55

Min-Max algorithm with Naïve approach

```
Algorithm: Max-Min-Element (numbers[])
max := numbers[1]
min := numbers[1]

for i = 2 to n do
   if numbers[i] > max then
      max := numbers[i]
   if numbers[i] < min then
      min := numbers[i]
return (max, min)
```

Divide and Conquer approach

```
int[] findMinMax(int A[], int start, int end)
{
    int max;
    int min;
    if ( start == end )
    {
        max = A[start]
        min = A[start]
    }
    else if ( start + 1 == end )
    {
        if ( A[start] < A[end] )
        {
            max = A[end]
            min = A[start]
        }
        else
        {
            max = A[start]
            min = A[end]
        }
    }
```

Divide and Conquer approach contd..

```
    else
    {
        int mid = start + (end - start)/2
        int left[] = findMinMax(A, start, mid)
        int right[] = findMinMax(A, mid+1, end)
        if ( left[0] > right[0] )
            max = left[0]
        else
            max = right[0]
        if ( left[1] < right[1] )
            min = left[1]
        else
            min = right[1]
    }
    // By convention, we assume ans[0] as max and ans[1] as min
    int ans[2] = {max, min}
   return ans
}
```

Time Complexity of Min-Max Algorithm

- The number of comparison in Naive method is $T(n)=2n - 2$
- The number of comparison in Divide and conquer is $T(n) = (3n/2) - 2$
- Compared to Naïve method, in divide and conquer approach, the number of comparisons is less
- However, using the asymptotic notation both of the approaches are represented by **O(n)**


Thank you