Design and Analysis of Algorithms Module 2:

Divide and Conquer Approach

Index -

Lecture 09

Introduction to Divide and Conquer Approach

Divide and Conquer General Method

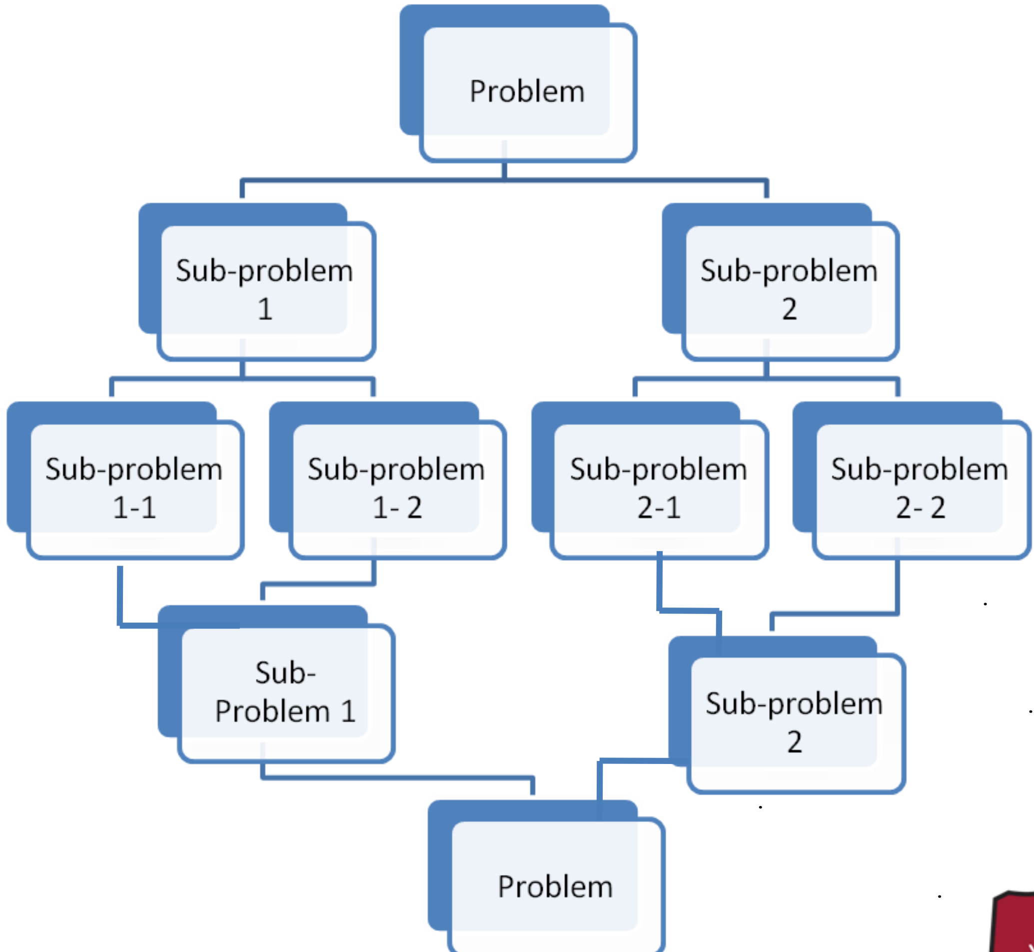
- divide and conquer is an ***algorithm design paradigm.***
- algorithm design paradigm is a generic model or framework which underlies the design of a class of algorithms.
- Strategy:
  - A ***divide-and-conquer*** algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.
  - The solutions to the sub-problems are then ***combined*** to give a solution to the original problem
- It is the basis of efficient algorithms for many problems such as merge sort, Quick sort.
- The correctness of a divide-and-conquer algorithm is usually proved by mathematical induction,
- The computational cost is often determined by solving recurrence relations.

D Y PATIL DEEMED TO BE UNIVERSITY
RAMRAO ADIK INSTITUTE OF TECHNOLOGY
NAVI MUMBAI

Divide and Conquer Approach

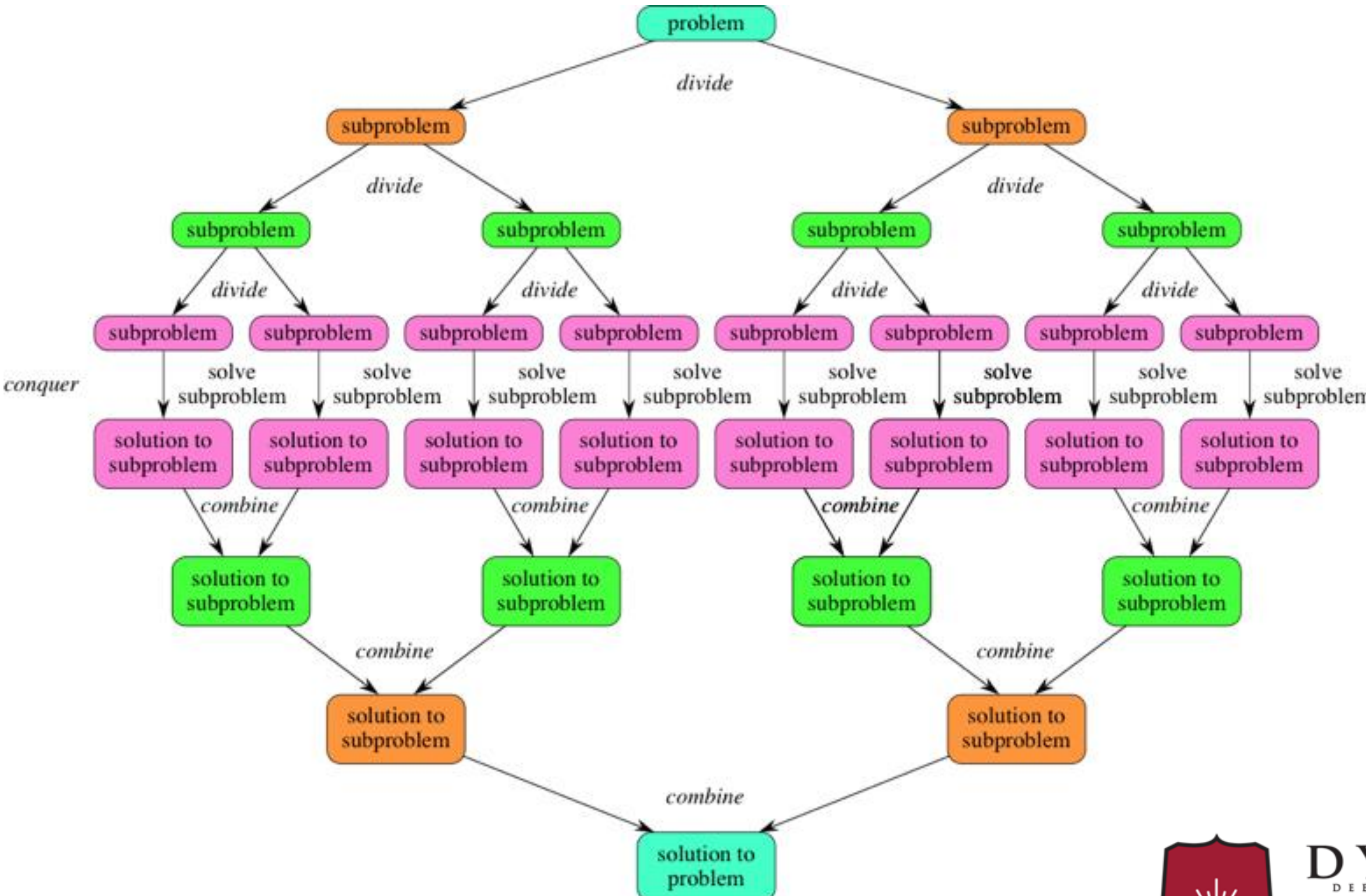
- think of divide-and-conquer algorithm as having three parts:
  1. Divide the problem into a number of subproblems that are smaller o instances of the same problem.
  2. Conquer the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
  3. Combine the solutions to the subproblems into the solution for the original problem.

Divide and Conquer Approach



D Y PATIL DEEMED TO BE UNIVERSITY RAMRAO ADIK INSTITUTE OF TECHNOLOGY NAVI MUMBAI

Divide and Conquer Approach



Divide and Conquer - Advantages

- **Solving difficult problems**
- **Algorithm efficiency**
- **Parallelism**
- **Memory access**
- **Round off control**

Lecture 10

Analysis of Binary Search

Binary Search - Example

- Binary Search, locates/find a element from the list of sorted items.
- Binary search, a divide-and-conquer algorithm where the sub problems are of roughly half the original size.
- Binary search compares the target value to the middle element of the array.
    - If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half.
    - again taking the middle element to compare to the target value, and repeating this until the target value is found.
    - If the search ends with the remaining half being empty, the target is not in the array

Binary Search

- Find x in array [low…..high]
  - **Verify base case**
  - Compare x with middle element in the array
    - There are 3 possible outcome
    - Case 1 : x == array[mid]
      - return mid (index of middle element)
    - Case 2: x < array[mid]
      - Find x in array[low……mid-1]
      - Note: this is same smaller sub-problem.
    - Case 3: x > array[mid]
      - Find x in array[mid+1……high]
      - Note: this is same smaller sub-problem.

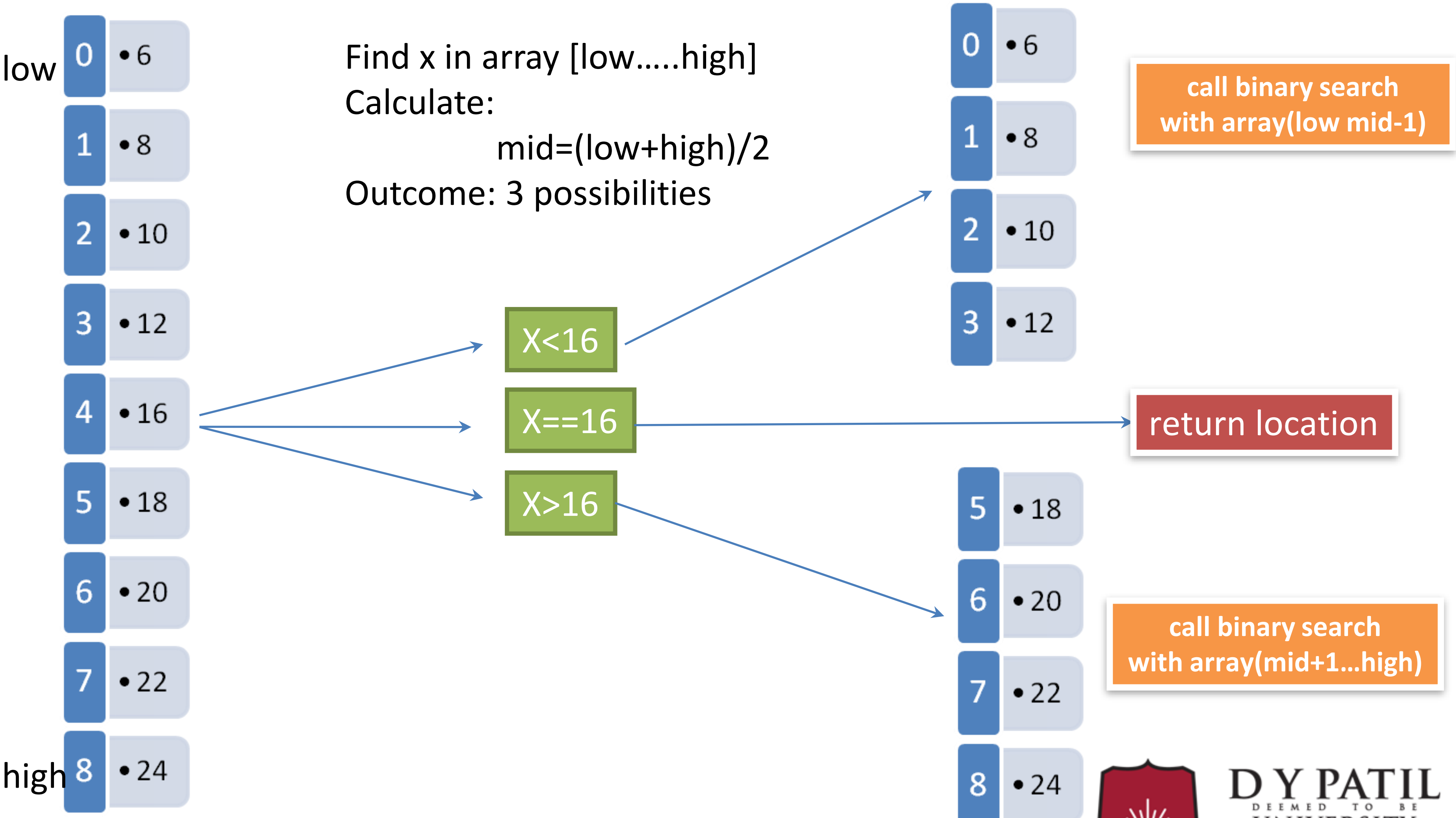
Binary Search Algorithm

1. Read array elements in increasing order.
2. Read the element to be searched, say ‘KEY ‘.
3. int BinarySearch(int array[], int start_index, int end_index, int KEY)

```
{
   if (end_index >= start_index)
   {
      int middle = (start_index+end_index)/2;
      if (array[middle] == KEY)
         return middle;
      if (array[middle] > KEY)
         return BinarySearch(array, start_index, middle-1, KEY);
      elseif (KEY<array[middle])
        return BinarySearch(array, middle+1, end_index, KEY);
   }
   return -1;
}
```

Binary Search - work

Find x in array [low…..high]
Calculate:
mid=(low+high)/2
Outcome: 3 possibilities



Binary Search – Base case

- The recursive call base case - to stop recursion

```
If(low>high){
            //stop recursion
}
```

- We recursive divide and decrease search space, in case key is not equal to middle element by
  - Making search space between low and mid-1 if x<array[mid]
  - Making search space between mid+1 and high if x>array[mid]
- The above logic of divide and decreasing search space, if you have atleast one element in search space of array i.e low<=high

Binary Search – Recurrence Equation

- $T(n)= T(n/2)+O(1)$

- Analysis using master's method
- Case 2 of master's method is application
  - $T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$
  - If $f(n) = \Theta(n^c)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$
  - $a=1$, $b=2$, $c=0$, also $\log_b a = \mathrm{Log}_2 1 = 0$
  - Thus, $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$
  - Thus complexity of Binary search is $\Theta(\log n)$ in worst case.

Binary Search – Recurrence Equation

- $T(n)= T(n/2)+O(1)$

- Best case: in best the element we are looking is found and middle
- Thus, there is no recursive call further
- $T(n)=O(1)$

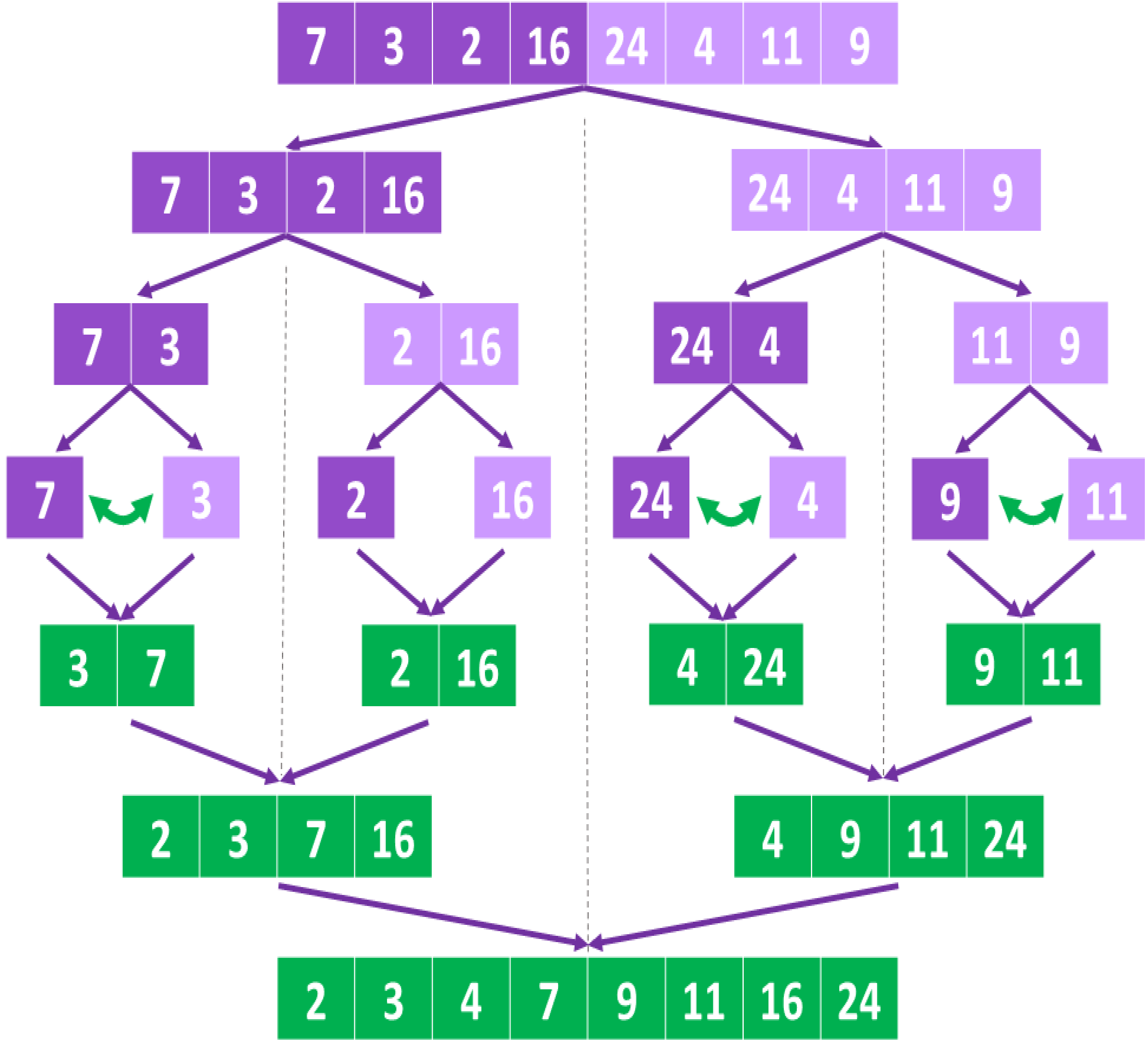
Lecture 11

Analysis of Merge Sort

Merge Sort

- The merge sort algorithm closely follows the divide-and-conquer methodology.
  - Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
  - Conquer: Sort the two subsequences recursively using merge sort.
  - Combine: Merge the two sorted subsequences to produce the sorted answer
- The recursion "bottoms out" when the sequence to be sorted has length 1.

Merge Sort



Step 1:
Split sub-lists in two until you reach pair of values.

Step 3:
Sort/swap pair of values if needed.

Step 4:
Merge and sort sub-lists and repeat process till you merge to the full list.

Merge Sort Algorithm

0	1	2	3	4	5	6	7
2	4	1	6	8	5	3	7

```
MERGESORT(A)
{
        n<- length(A)
        if(n<2)
        {
                return
        }
        mid<- n/2;
        left = array of size (mid)
        right=array of size(n-mid)
        for(i=0 to mid-1)
        {
                left[i] <- A[i]
        }
        for(i=mid to n-1)
        {
                right[i-mid] <- A[i]
        }
        MERGESORT (L)
        MERGESORT (R)
        MERGESORT (L, R, A)
}
```

0	1	2	3	4	5	6	7
2	4	1	6	8	5	3	7

Merge Sort Algorithm

```
MERGESORT (L, R, A)
{
        nL <- length(L)
        nR <- length(R)
        i = 0, j=0, k=0
        while(i<nL && j<nR)
        {
                if(L[i] <= R[j])
                {
                        A[k] <- L[i]; i<- i+1
                }
                else
                {
                        A[k]<- R[j]; j<-j+1
                }
                k=k+1
        }
        while(i<nL)
        {
                A[k]=L[i];  i<-i+1; k<- k+1;
        }
        while(j<nR)
        {
                A[k]=R[j]; j<-j+1; k<-k+1;
        }
}
```

0	1	2	3	4	5	6	7
38	27	43	3	9	82	10	7

What is Recurrence Equation?

Best Case, Average Case and Worst case are same in Merge sort:

$T(1)=1$ for $n=1$

$T(n)= T(n/2) + T(n/2) + Cn$ for $n>1$

Time taken by left sublist to get sorted

Time taken for combine sublist

Time taken by right sublist to get sorted

$$T(n)= \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2)+cn & n>1 \end{cases}$$

Analysis

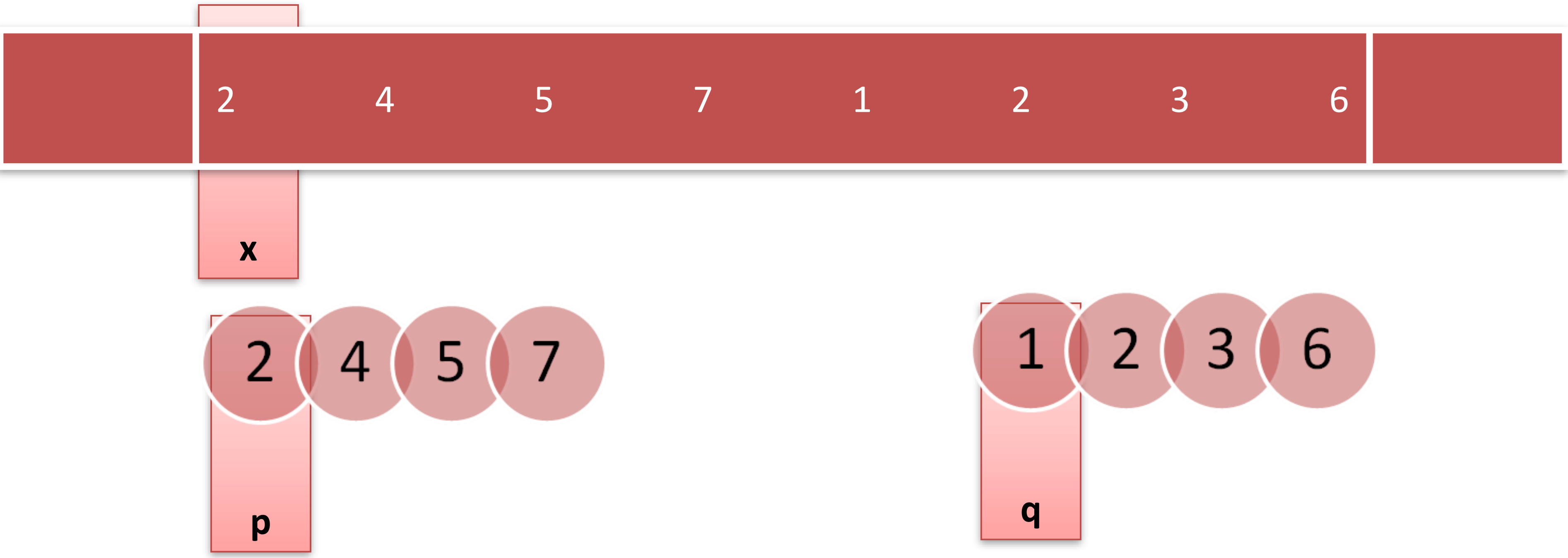
- Recurrence Equation of Merge sort
- $T(n)=2T(n/2)+O(n)$
- How we got above equation:
  - Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.
  - Conquer: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
  - Combine: We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, so $C(n) = \Theta(n)$.

Analysis

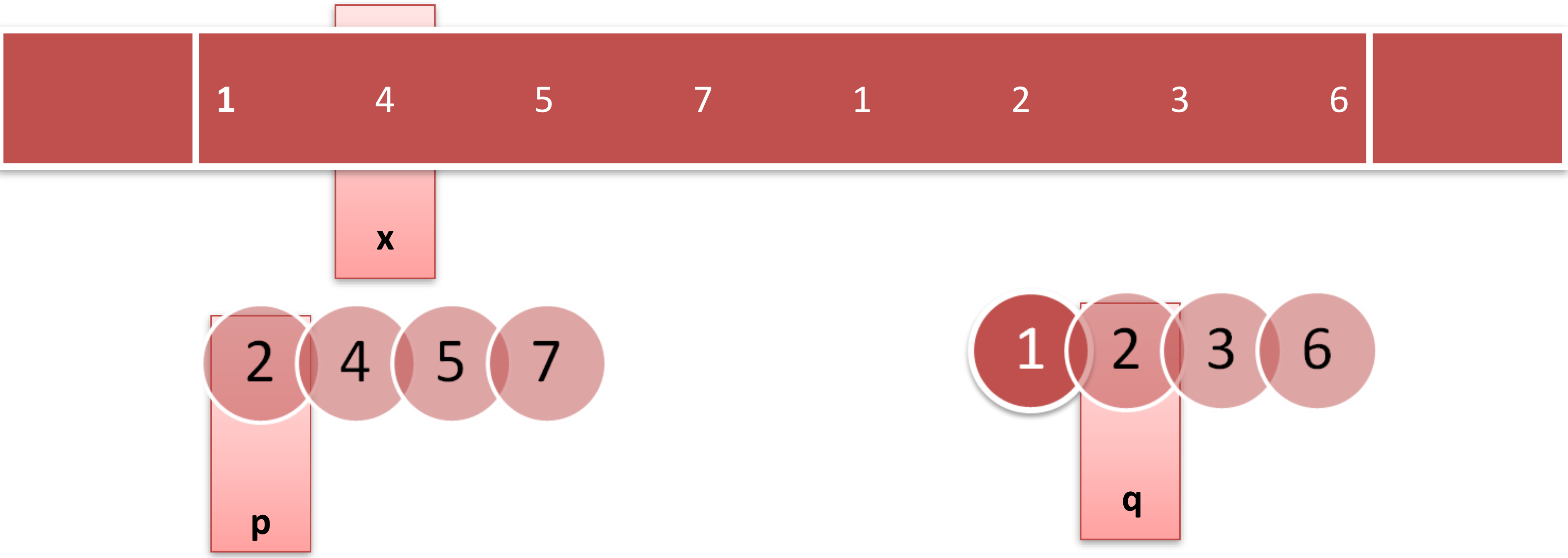
- Recurrence Equation of Merge sort
- $T(n)=2T(n/2)+O(n)$

- Analysis using master’s method
- Case 2 of master’s method is application
  - $T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$
  - If $f(n) = \Theta(n^c)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$
  - $a=2$, $b=2$, $c=0$, also $\mathrm{Log}_b a = \mathrm{Log}_2 2 = 1$
  - Thus, $T(n) = \Theta(n^{1l}og\ n) = \Theta(n\log n)$

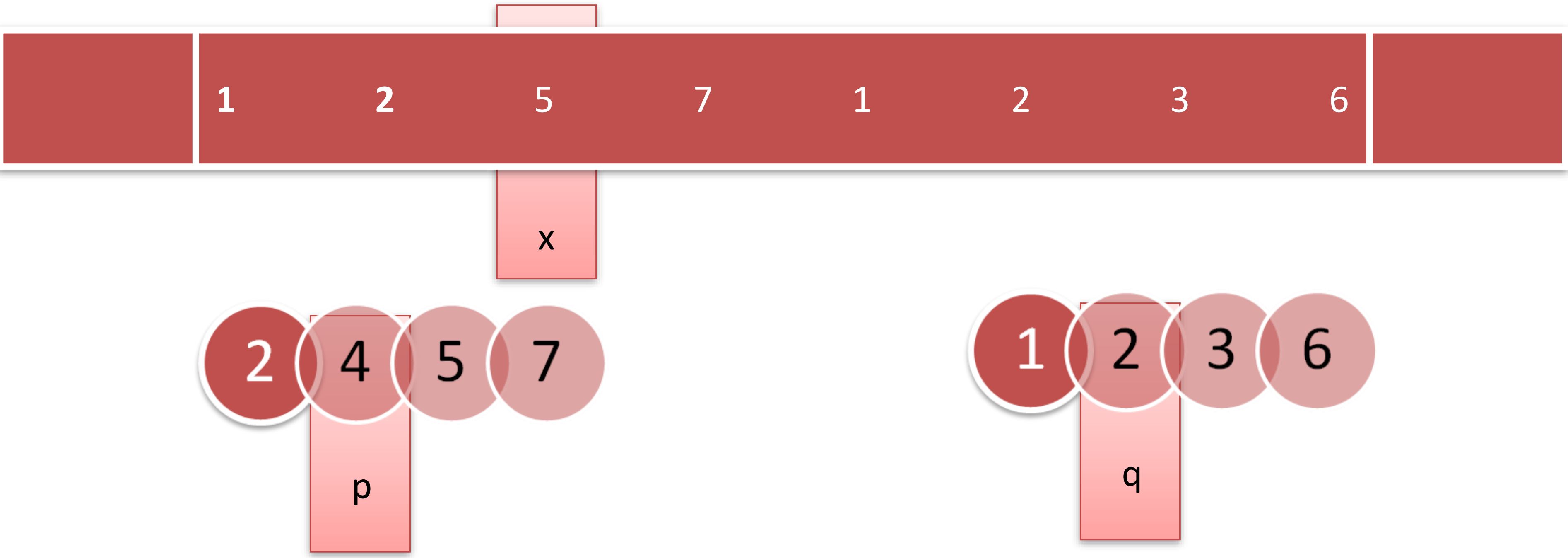
Merge Algorithm - working



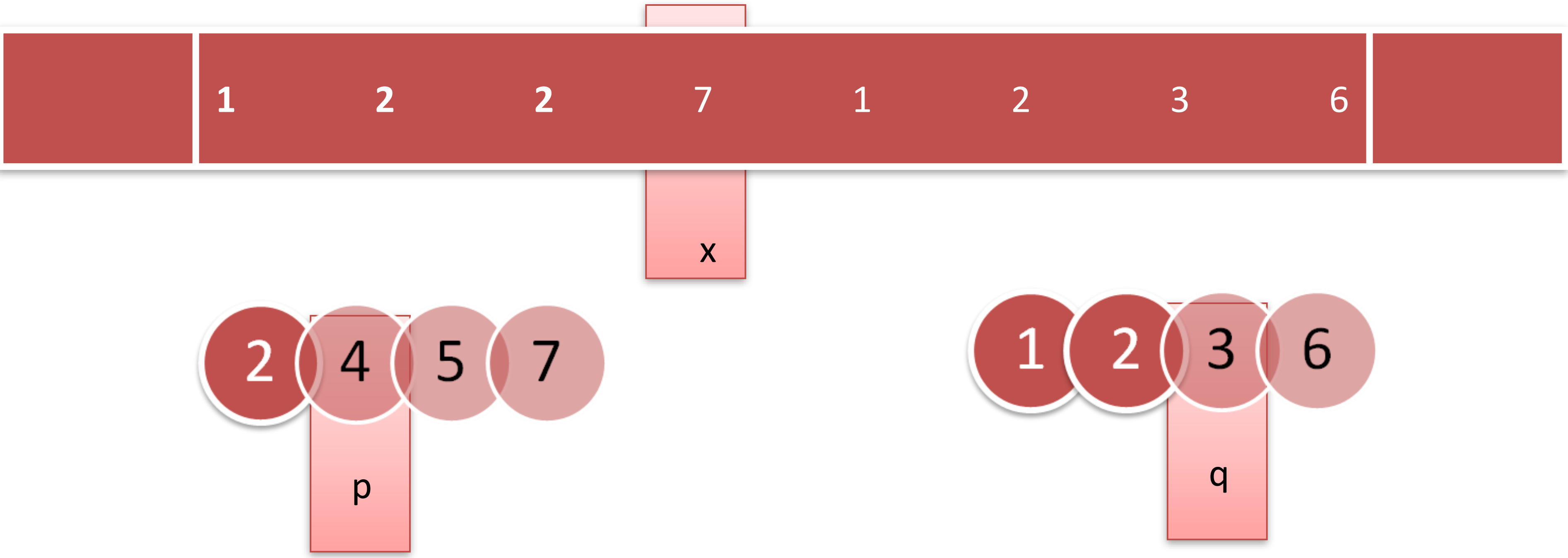
Merge Algorithm - working



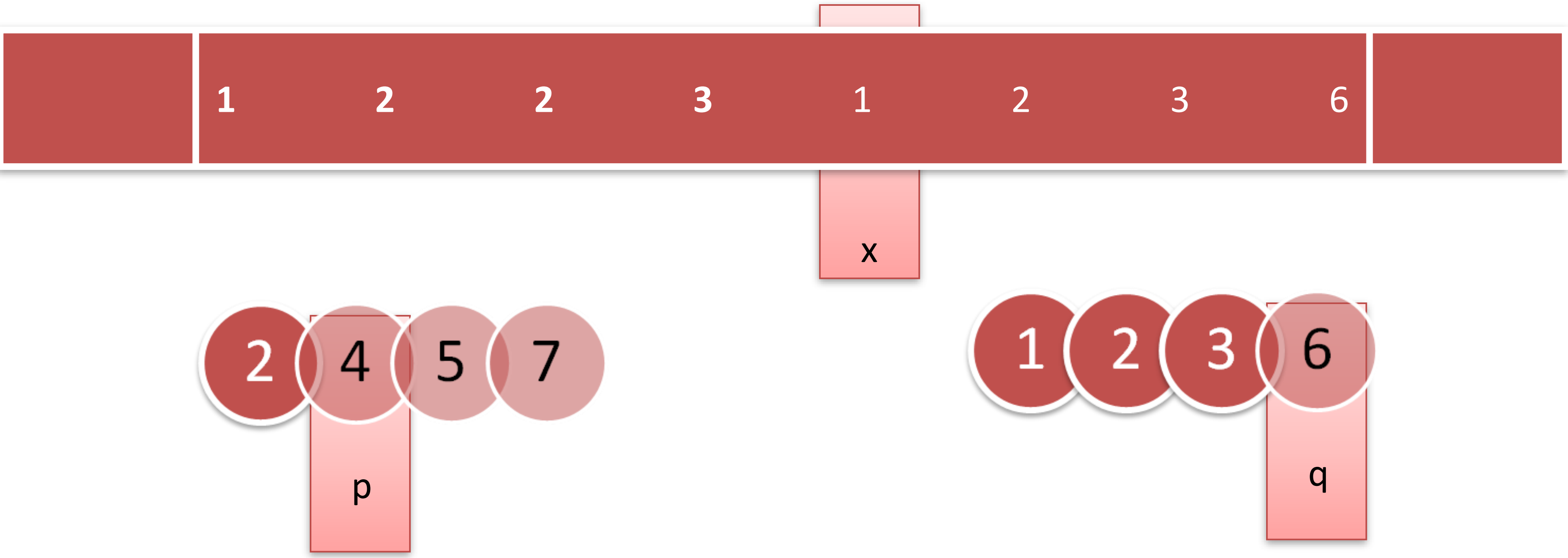
Merge Algorithm - working



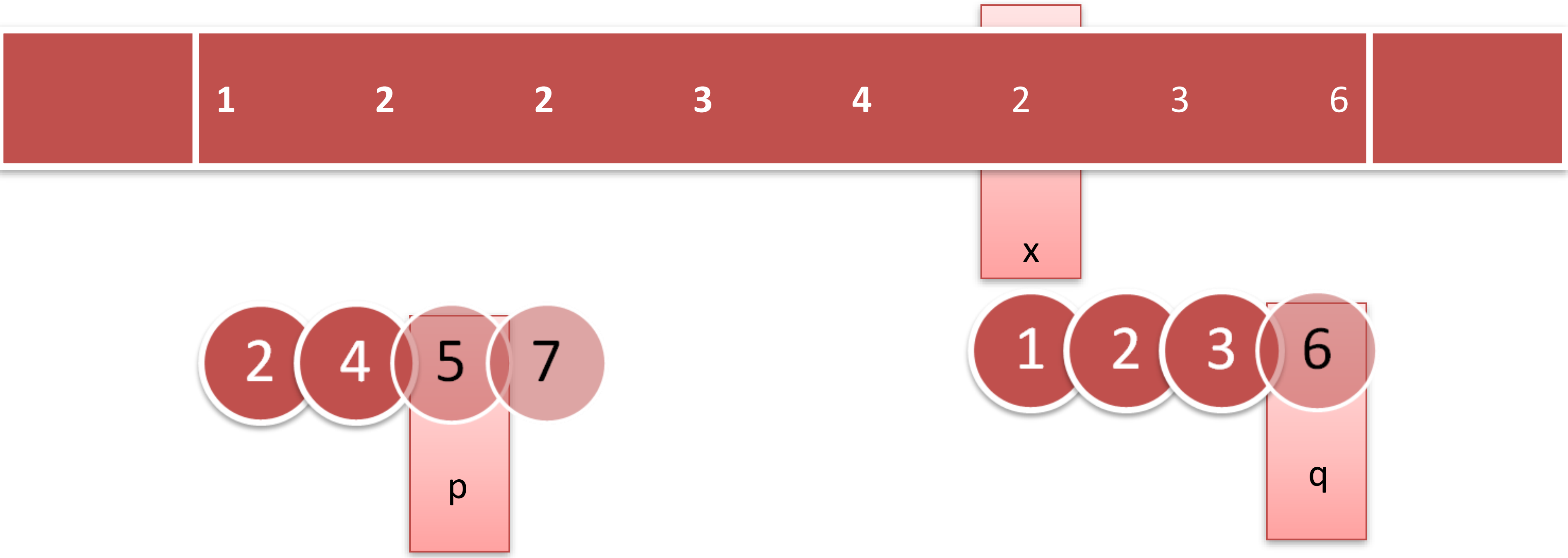
Merge Algorithm - working



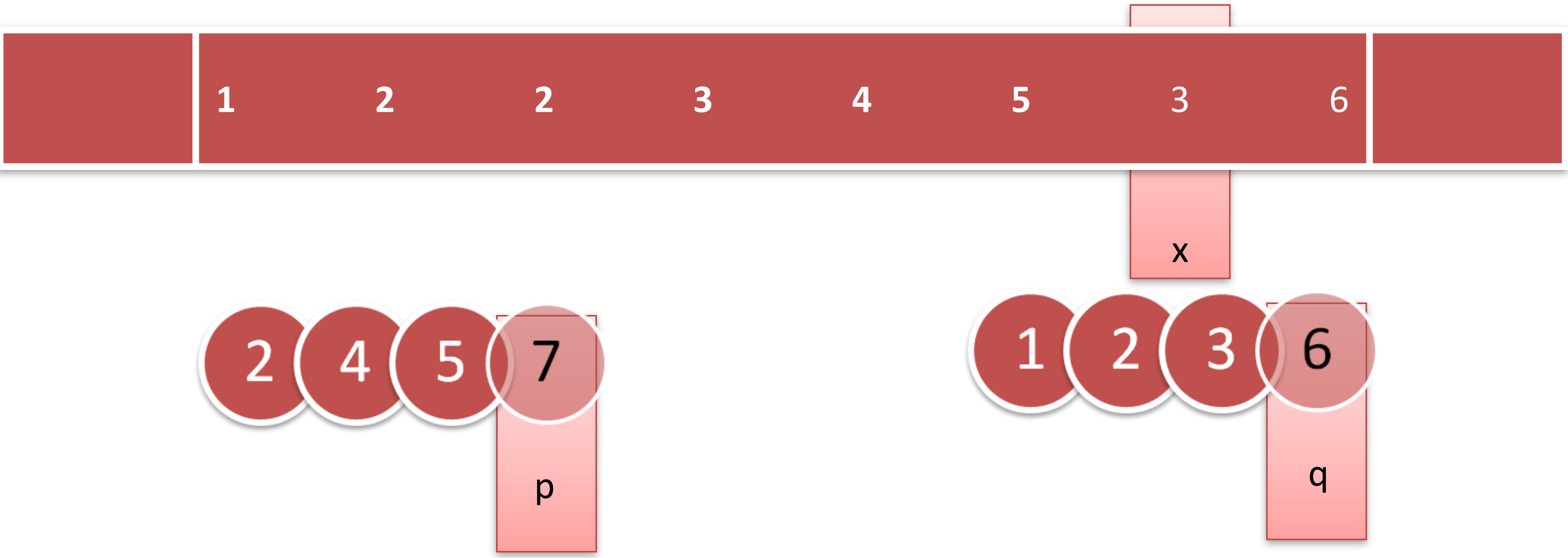
Merge Algorithm - working



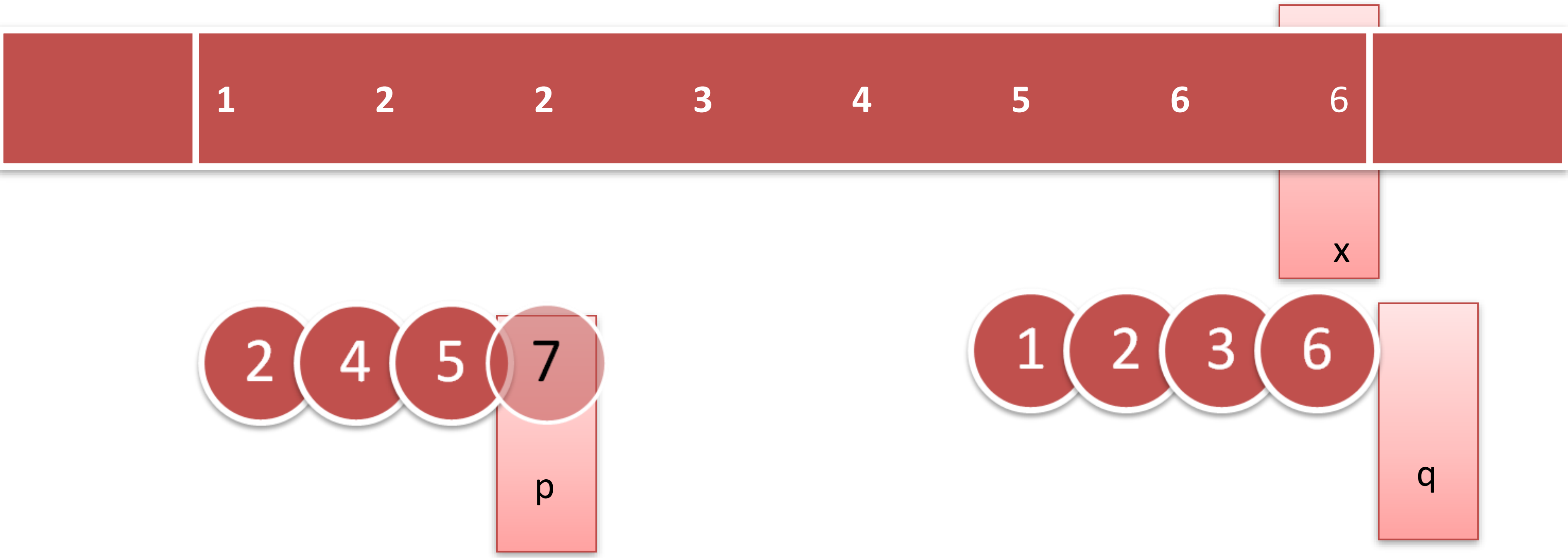
Merge Algorithm - working



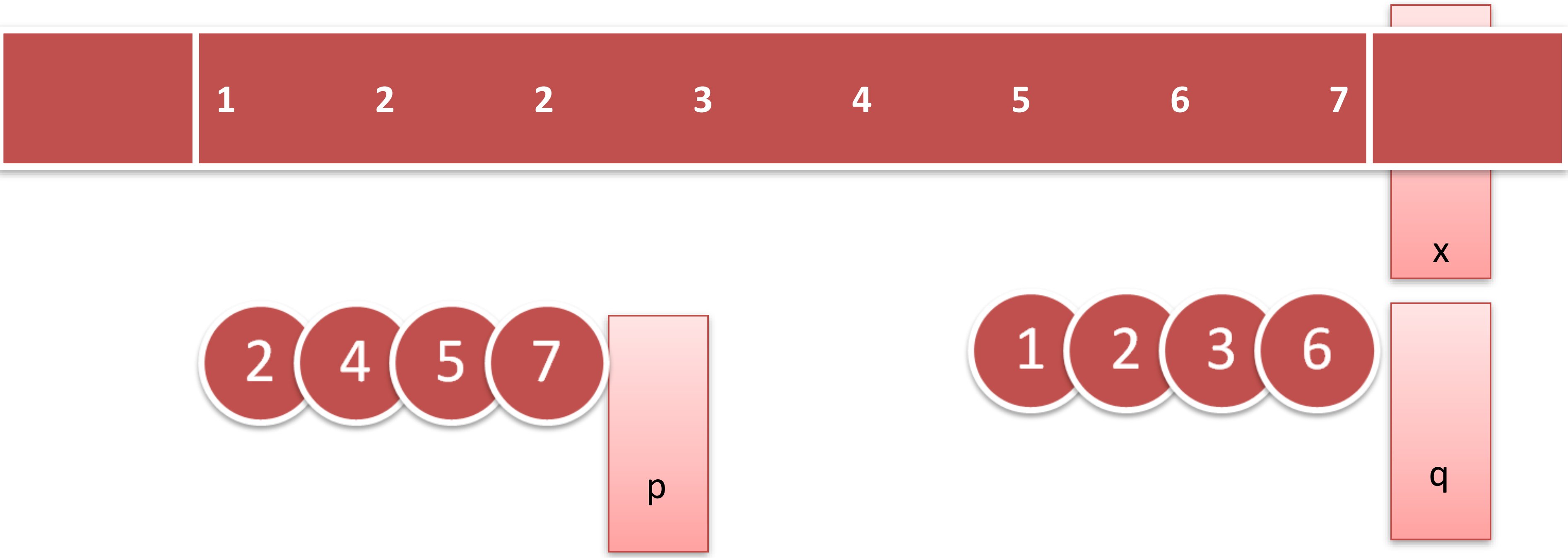
Merge Algorithm - working



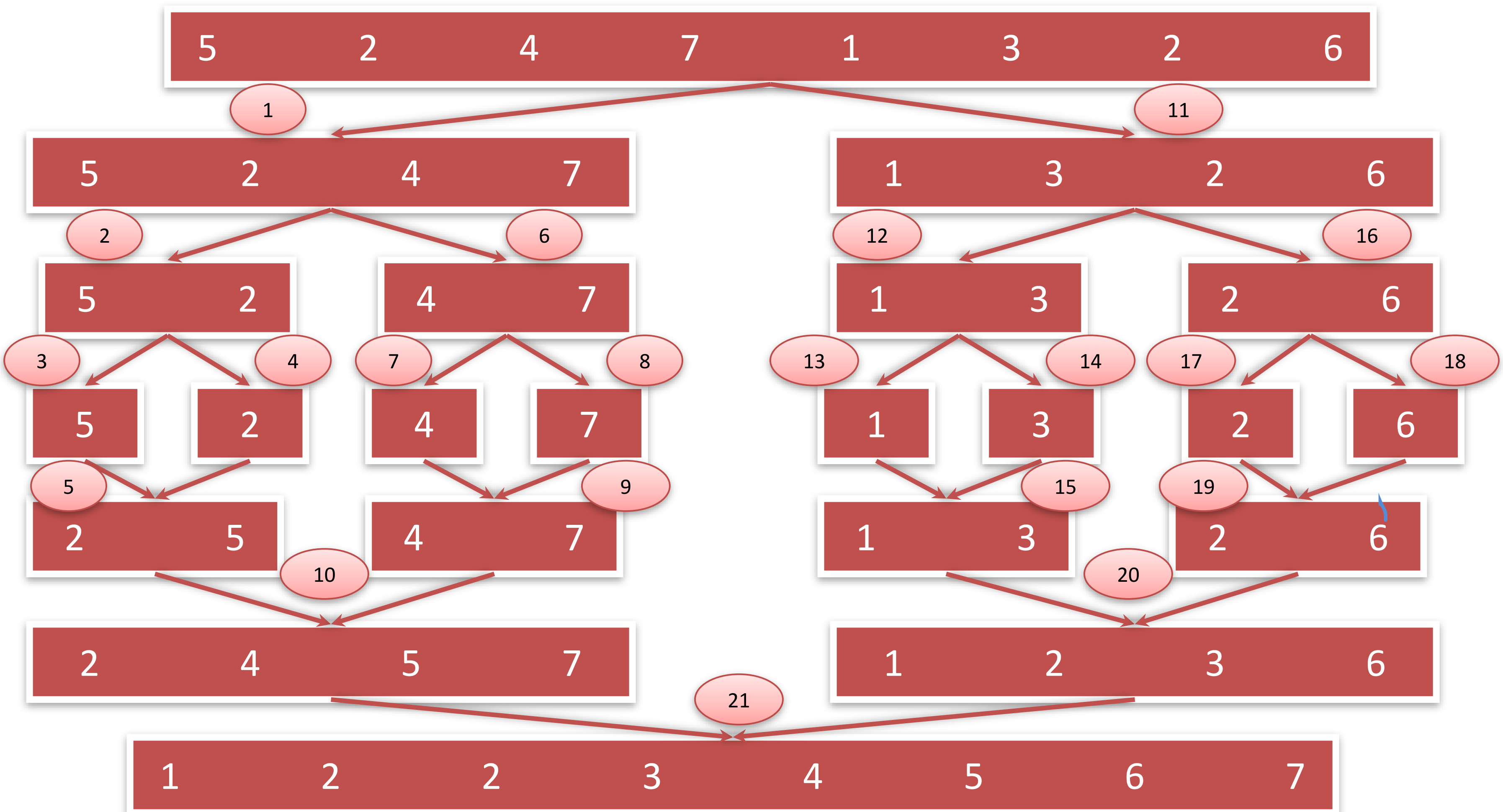
Merge Algorithm - working



Merge Algorithm - working



Example....



5 2 4 7 1 3 2 6
1
11
5 2 4 7
1 3 2 6
2
6
12
16
5 2
4 7
1 3
2 6
3
4
7
8
13
14
17
18
5
2
4
7
1
3
2
6
5
9
15
19
2 5
4 7
1 3
2 6
10
20
2 4 5 7
1 2 3 6
21
1 2 2 3 4 5 6 7

Lecture 12

Analysis of Quick Sort

Quick Sort

- Quicksort, like merge sort, is based on the divide-and-conquer paradigm
- It is the three-step divide-and-conquer process for sorting a typical subarray A[p … r].
  - **Divide:** Partition (rearrange) the array $A[p \ldots r]$ into two (possibly empty) subarrays $A[p \ldots q - 1]$ and $A[q + 1 \ldots r]$
    - such that each element of $A[p \ldots q - 1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q + 1 \ldots r]$.
    - Compute the index q as part of this partitioning procedure.
  - **Conquer:** Sort the two subarrays $A[p \ldots q -1]$ and $A[q +1 \ldots r]$ by recursive calls to quicksort.
  - **Combine:** Since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[p \quad r]$ is now sorted.

Quick Sort Algorithm

0	1	2	3	4	5	6	7	
35	50	15	25	80	20	90	45	+∞

Quick Sort Algorithm- Method 1

```
Quick_sort(l, h)
{
        if(l<h)
        {
                j=partition(l, h);
                Quick_sort(l, j)
                Quick_sort(j+1, h)
        }
}
```

0	1	2	3	4	5	6	7	
35	50	15	25	80	20	90	45	+∞

Quick Sort Algorithm- Method 1

0	1	2	3	4	5	6	7	
35	50	15	25	80	20	90	45	+∞

```
partition(l, h)
{
        pivot=A[l];
        i=l; j=h;
        while(i<j)
        {
                do
                { i++; }
                while(A[i] <= pivot);
                do
                { j--;}
                while(A[j] > pivot);
                if(i<j)
                { swap(A[i], A[j]);}
        }
        swap(A[l], A[j]);
        return j;
}
```

Quick Sort – Algorithm Method 2

```
QUICKSORT(A, p, r)
    if p < r
        then q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q - 1)
        QUICKSORT(A, q + 1, r)
```

- Quick sort divides array into subarrays:

2	8	7	1	3	5	6	4

2	1	3

4

7	5	6	8

p q r

Recursively call Quick sort A[q+1…r]

Recursively call Quick sort A[p…q-1]

D Y PATIL DEEMED TO BE UNIVERSITY
RAMRAO ADIK INSTITUTE OF TECHNOLOGY
NAVI MUMBAI

Quick Sort – Algorithm Method 2

- QUICKSORT(A, p, r)

1. if p < r
2. then q ← PARTITION(A, p, r)
3. QUICKSORT(A, p, q - 1)
4. QUICKSORT(A, q + 1, r)

- Recurrence Equation:

$$T(n) = T(q) + T(n - q) + f(n)$$

Call to quicksort A[p…q-1]

Call to quicksort A[q+1…r]

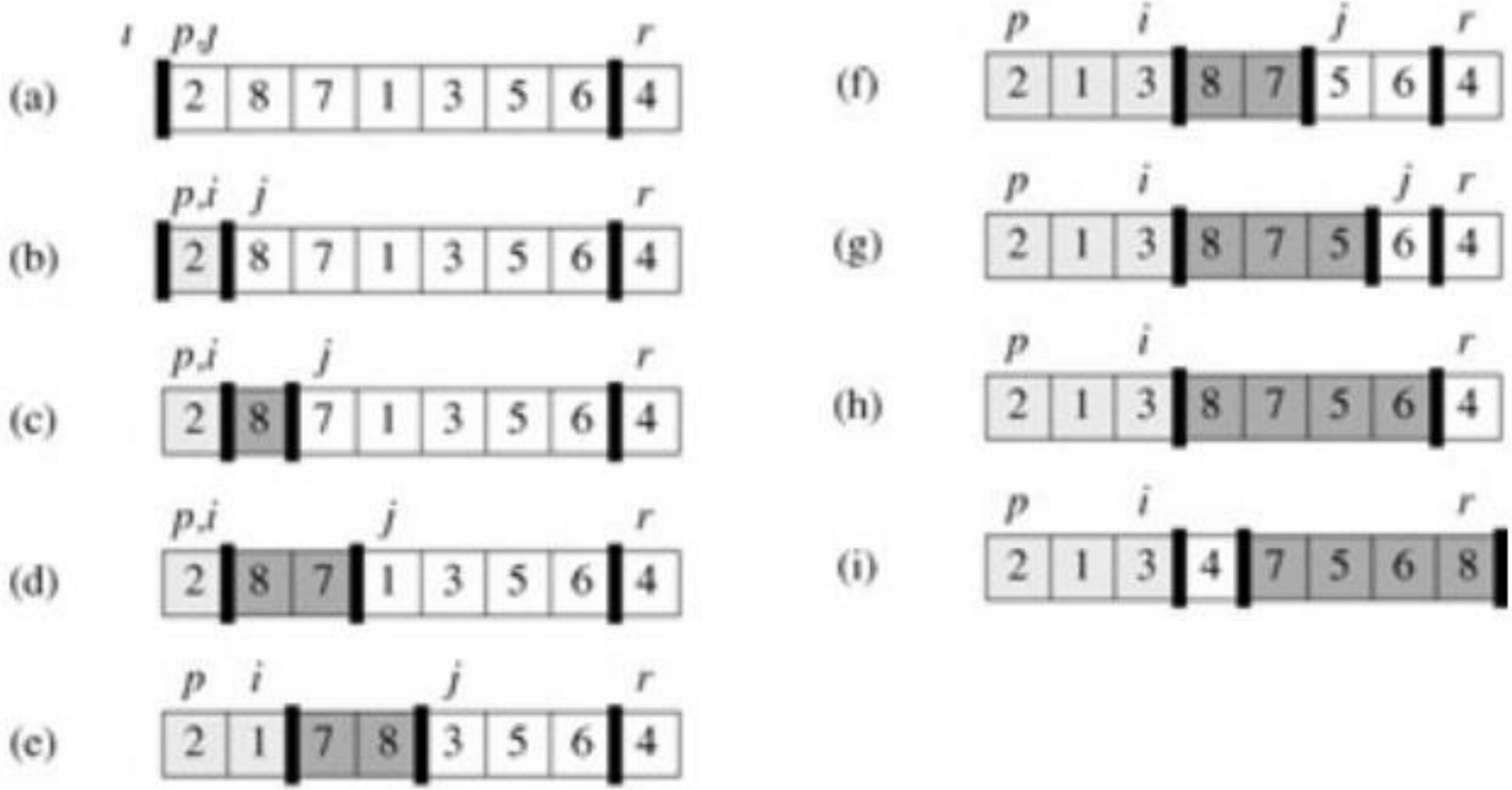
Call to partition

Quick Sort – Algorithm Method 2

```
PARTITION(A, p, r)
    x ← A[r]
    i ← p - 1
    for j ← p to r - 1
        do if A[j] ≤ x
            then i ← i + 1
            exchange A[i] ↔ A[j]
    exchange A[i + 1] ↔ A[r]
    return i + 1
```

Quick Sort - Algorithm

- operation of PARTITION function on an 8-element array



D Y PATIL DEEMED TO BE UNIVERSITY
RAMRAO ADIK INSTITUTE OF TECHNOLOGY
NAVI MUMBAI

Quick Sort - Algorithm

- PARTITION(A, p, r)

```
x ← A[r]
i ← p - 1
for j ← p to r - 1
    do if A[j] ≤ x
        then i ← i + 1
        exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

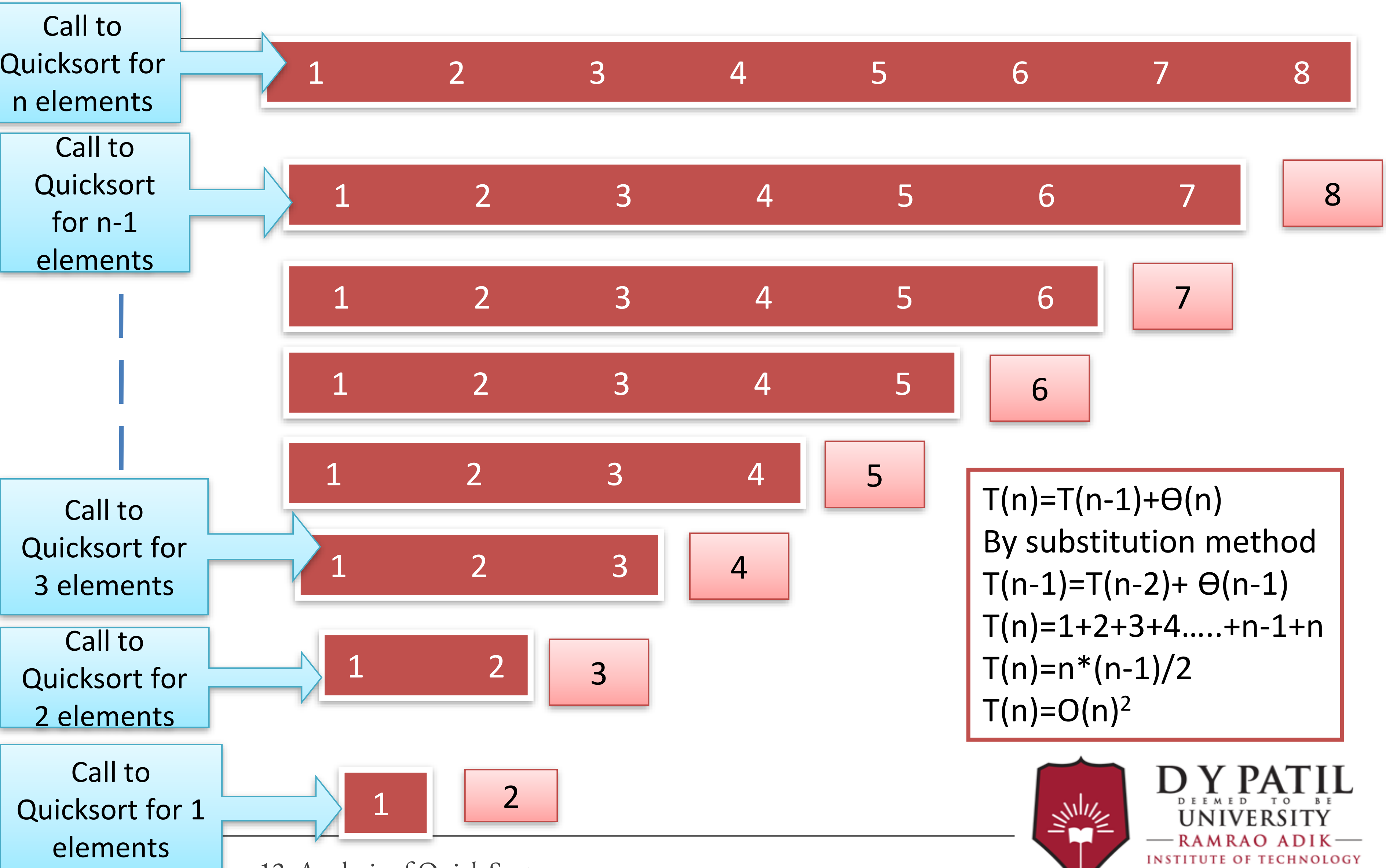
- The running time of PARTITION on the subarray $A[p\ldots..r]$ is $\Theta(n)$
- Thus recurrence equation becomes:

$$T(n) = T(q) + T(n - q) + \Theta\,(n)$$

Quick Sort - Analysis

- The complexity of Quick sort depends upon partitioning.
- If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort (Best Case).
- If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort (Worst Case).

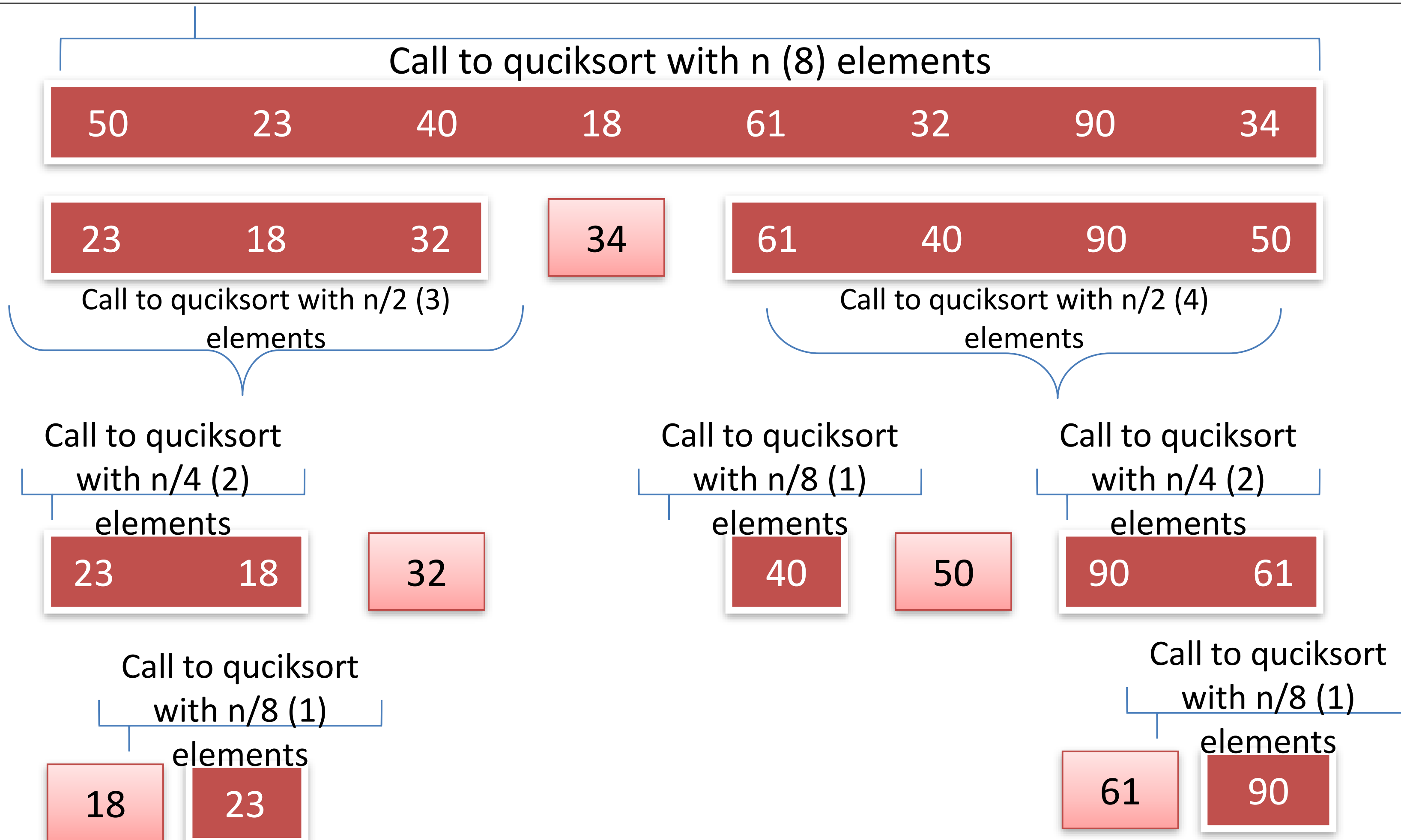
Worst Case



T(n)=T(n-1)+Ɵ(n)
By substitution method
T(n-1)=T(n-2)+ Ɵ(n-1)
T(n)=1+2+3+4…..+n-1+n
T(n)=n*(n-1)/2
$T(n)=O(n)^2$

D Y PATIL DEEMED TO BE UNIVERSITY
RAMRAO ADIK INSTITUTE OF TECHNOLOGY
NAVI MUMBAI

Best Case

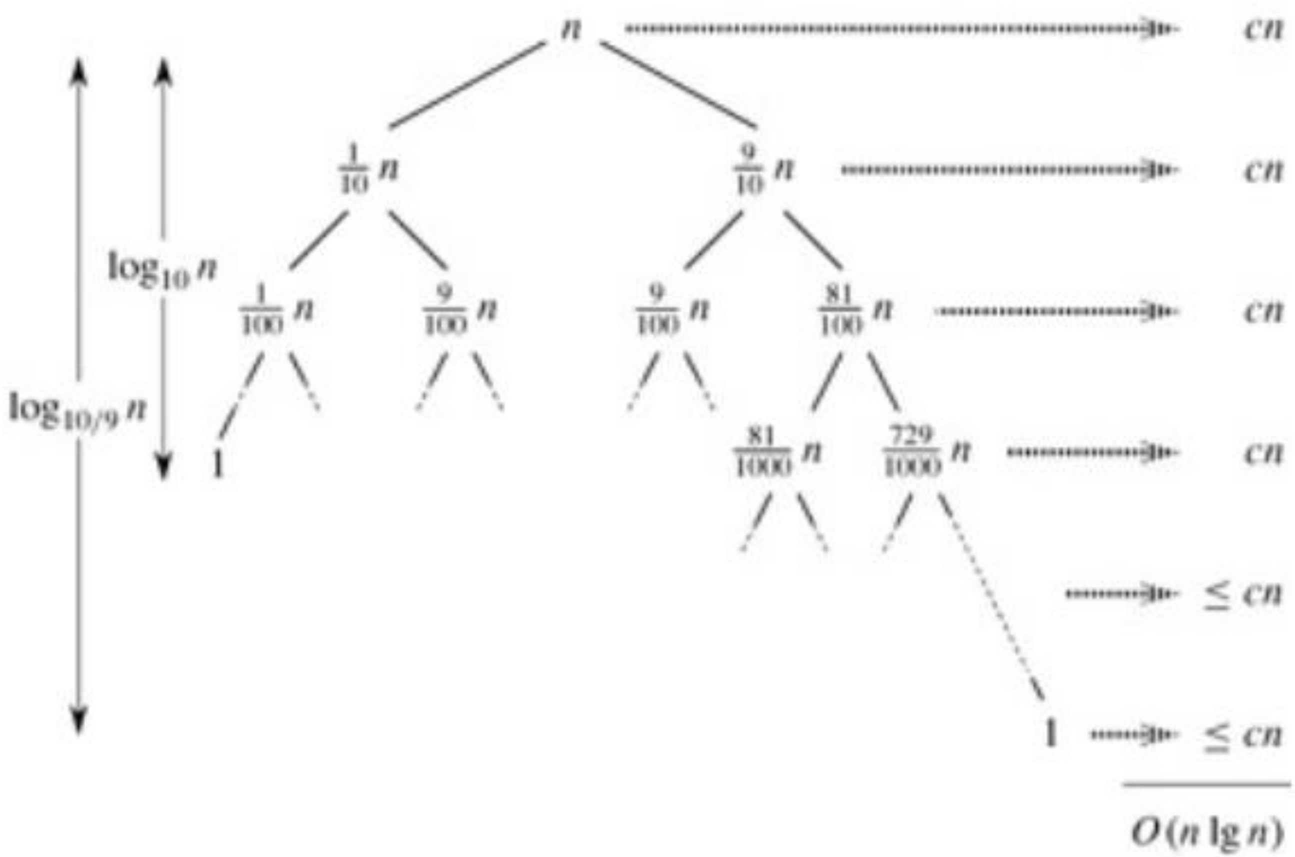

Call to quciksort with n (8) elements
50 23 40 18 61 32 90 34
23 18 32
34
61 40 90 50
Call to quciksort with n/2 (3) elements
Call to quciksort with n/2 (4) elements
Call to quciksort with n/4 (2) elements
23 18
32
Call to quciksort with n/8 (1) elements
40
50
Call to quciksort with n/4 (2) elements
90 61
Call to quciksort with n/8 (1) elements
18
23
Call to quciksort with n/8 (1) elements
61
90

Best Case

- Looking example, every time the array is divide in to half
- Thus, $T(n)=2T(n/2)+ \Theta (n)$
- The above equation is similar to merge sort
- Analysis using master's method
- Case 2 of master's method is application
  - $T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$
  - If $f(n) = \Theta(n^c)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$
  - $a=2$, $b=2$, $c=0$, also $\mathrm{Log}_b a = \mathrm{Log}_2 2 = 1$
  - Thus, $T(n) = \Theta(n^1 \log n) = \Theta(n\log n)$

Average Case

- Suppose on calling qucik sort the size of left subarray is 9n/10 and right subarray is n/10.



Thank You