Design and Analysis of Algorithms (CSC-314)

B.Sc. CSIT

Unit-3:Divide and Conquer Algorithms

Introduction:

- In computer science, divide and conquer is an algorithm design paradigm.
- 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.
- The divide-and-conquer technique is the basis of efficient algorithms for many problems, such as sorting (e.g., quicksort, merge sort).

Unit-3:Divide and Conquer Algorithms

Introduction:

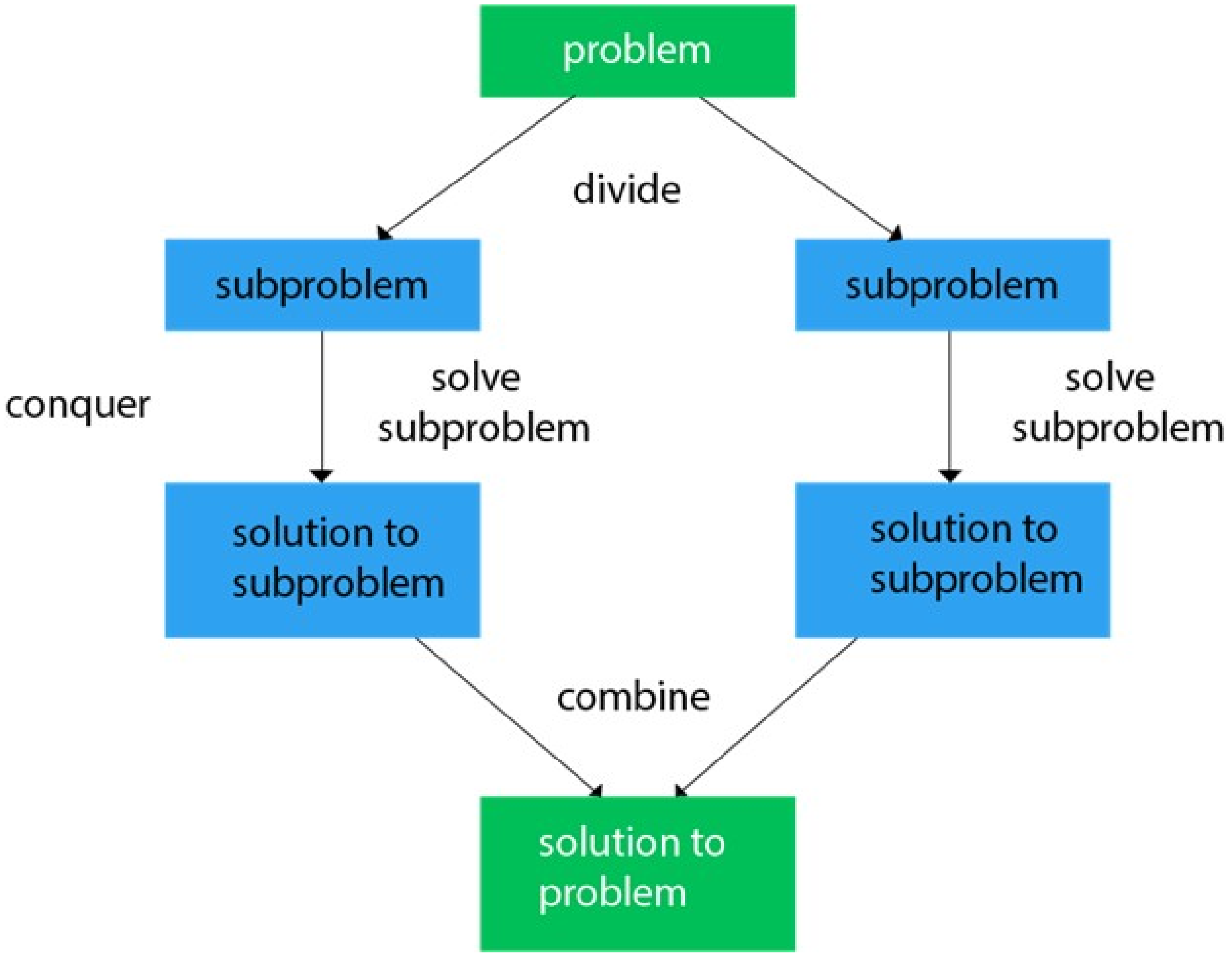
- This technique can be divided into the following three parts:
  - **Divide**: This involves dividing the problem into smaller sub-problems.
  - **Conquer**: Solve sub-problems by calling recursively until solved.
  - **Combine**: Combine the sub-problems to get the final solution of the whole problem.

Unit-3:Divide and Conquer Algorithms

Introduction:

- This technique can be divided into the following three parts:
  - **Divide**: This involves dividing the problem into smaller sub-problems.
  - **Conquer**: Solve sub-problems by calling recursively until solved.
  - **Combine**: Combine the sub-problems to get the final solution of the whole problem.

Unit-3:Divide and Conquer Algorithms



Unit-3:Divide and Conquer Algorithms

Algorithm:

```
Algorithm D and C (P)  {
    if small(P)
        then return S(P)
    else {
        – divide P into smaller instances P1 ,P2 .....Pk
        – Apply D and C to each sub problems
        – Return combine (D and C(P1)+ D and C(P2)+.......+D and
          C(Pk))
          }
}
```

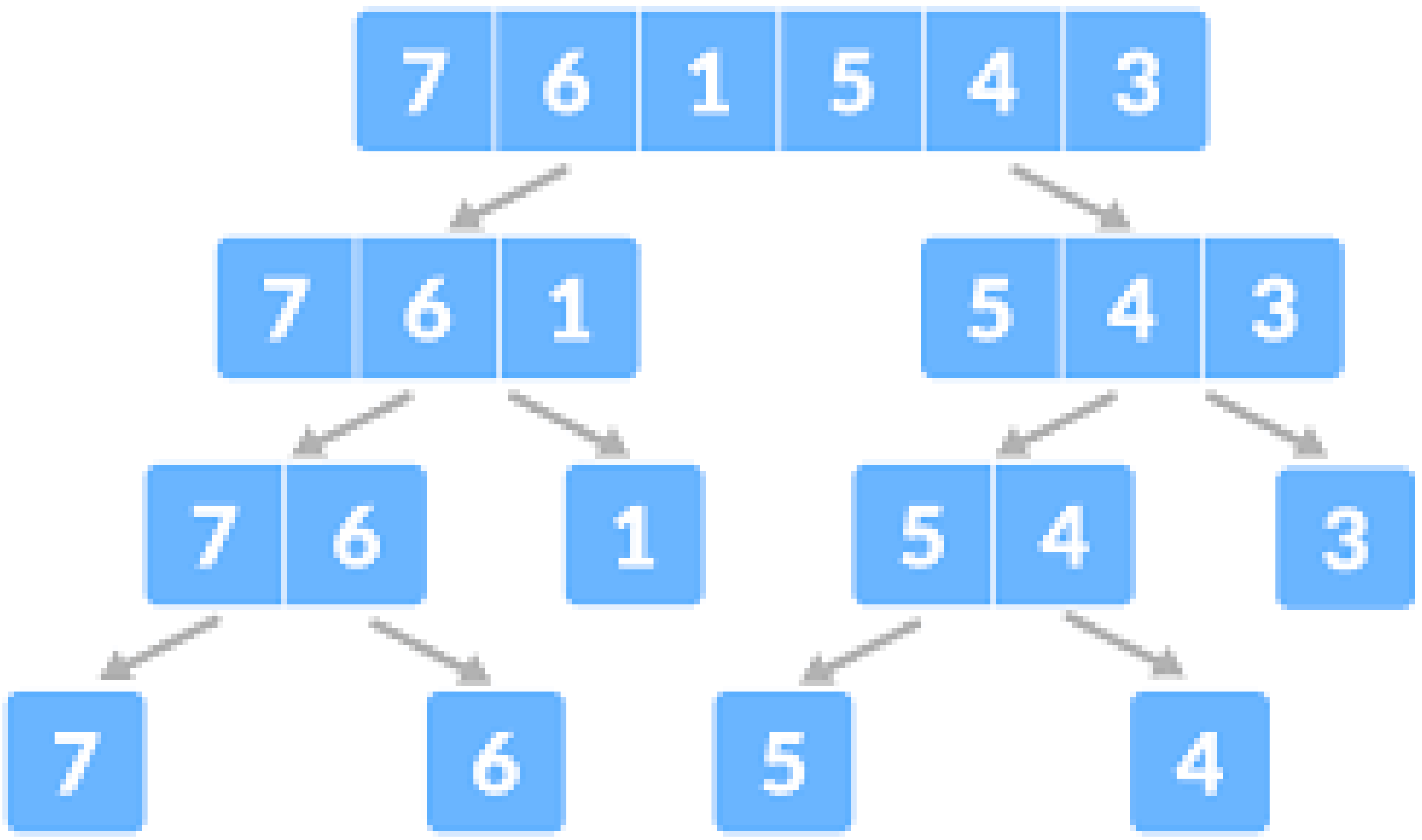
Unit-3:Divide and Conquer Algorithms

Let a recurrence relation is expressed as

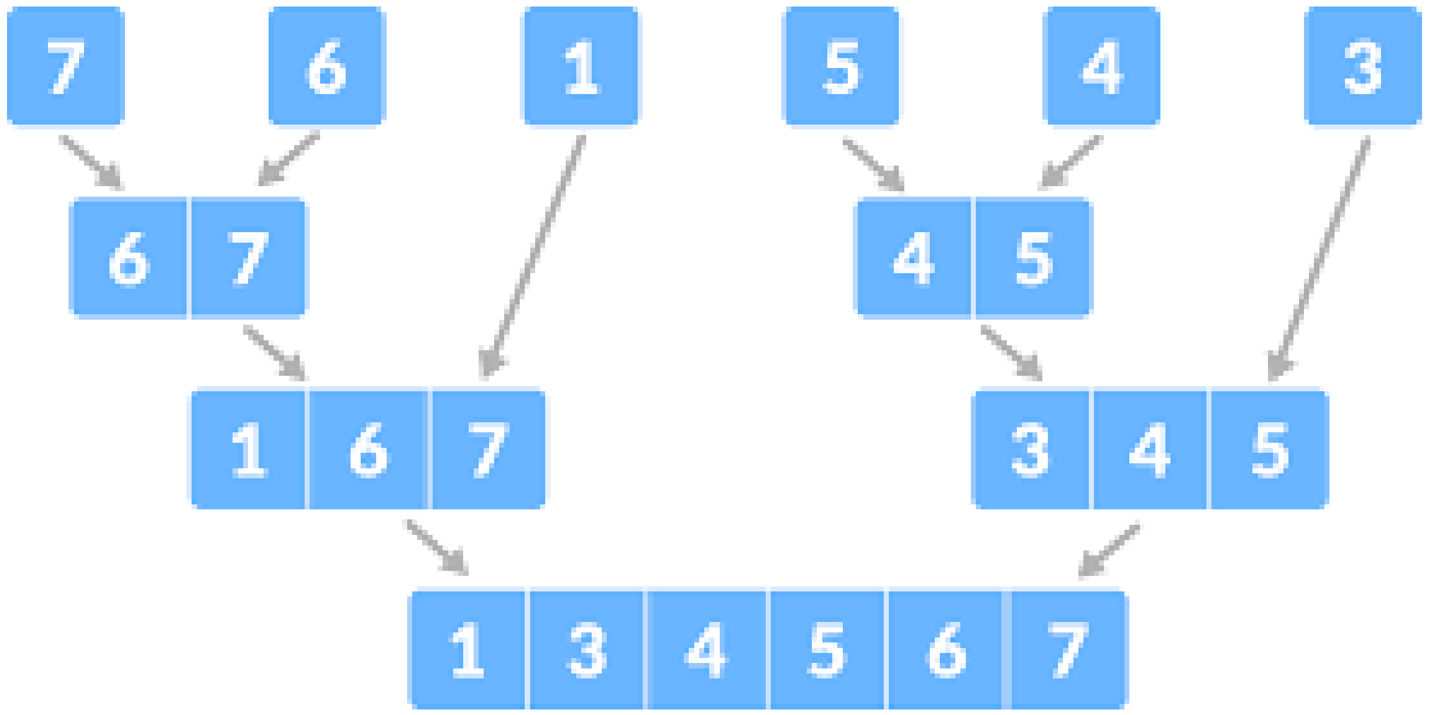
$$T(n) = \begin{cases} \Theta(1), & \text{if } n <= C \\ aT(n/b) + D(n) + C(n), & \text{otherwise} \end{cases}$$

- Here,
  - n=input size
  - a = no. of sub-problems
  - n/b = input size of the sub-problems

Unit-3:Divide and Conquer Algorithms



Unit-3:Divide and Conquer Algorithms



Unit-3:Divide and Conquer Algorithms

Some of the specific computer algorithms are based on the Divide & Conquer approach:

- Maximum and Minimum Problem
- Binary Search
- Sorting (merge sort, quick sort)
- Tower of Hanoi...etc.

Unit-3:Divide and Conquer Algorithms

Some Applications of Divide and Conquer Approach:

- **Following algorithms are based on the concept of the Divide and Conquer Technique:**
- **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
- **Quicksort**: It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.

Unit-3:Divide and Conquer Algorithms

Some Applications of Divide and Conquer Approach:

- **Following algorithms are based on the concept of the Divide and Conquer Technique:**
- **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.
- **Closest Pair of Points:** It is a problem of computational geometry. This algorithm emphasizes finding out the closest pair of points in a metric space, given n points, such that the distance between the pair of points should be minimal.

Unit-3:Divide and Conquer Algorithms

Some Applications of Divide and Conquer Approach:

- **Following algorithms are based on the concept of the Divide and Conquer Technique:**
- **Strassen's Algorithm:** It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be much faster than the traditional algorithm when works on large matrices.
- **Cooley-Tukey Fast Fourier Transform (FFT) algorithm:** The Fast Fourier Transform algorithm is named after J. W. Cooley and John Turkey. It follows the Divide and Conquer Approach and imposes a complexity of O(nlogn).
- **Karatsuba algorithm for fast multiplication:** It is one of the fastest multiplication algorithms of the traditional time, invented by Anatoly Karatsuba in late 1960 and got published in 1962. It multiplies two n-digit numbers in such a way by reducing it to at most single-digit.

Unit-3:Divide and Conquer Algorithms

Some Applications of Divide and Conquer Approach:

- **Following algorithms are based on the concept of the Divide and Conquer Technique:**
- **Strassen's Algorithm:** It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be much faster than the traditional algorithm when works on large matrices.
- **Cooley-Tukey Fast Fourier Transform (FFT) algorithm:** The Fast Fourier Transform algorithm is named after J. W. Cooley and John Turkey. It follows the Divide and Conquer Approach and imposes a complexity of O(nlogn).
- **Karatsuba algorithm for fast multiplication:** It is one of the fastest multiplication algorithms of the traditional time, invented by Anatoly Karatsuba in late 1960 and got published in 1962. It multiplies two n-digit numbers in such a way by reducing it to at most single-digit.

Unit-3:Divide and Conquer Algorithms

Advantages of Divide and Conquer

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses cache memory without occupying much space because it solves simple sub-problems within the cache memory instead of accessing the slower main memory.
- It is more proficient than that of its counterpart Brute Force technique.
- Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing

Unit-3:Divide and Conquer Algorithms

Disadvantages of Divide and Conquer

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
- An explicit stack may overuse the space.
- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

Unit-3:Divide and Conquer Algorithms

What is Search?

- Search is a utility that enables its user to find documents, files, media, or any other type of data held inside a database.
- Search works on the simple principle of matching the criteria with the records and displaying it to the user. In this way, the most basic search function works.

Unit-3:Divide and Conquer Algorithms

Searching Algorithms:

- In computer science, a search algorithm is an algorithm which solves a search problem.
- Search algorithms work to retrieve information stored within some data structure, or calculated in the search space of a problem domain, either with discrete or continuous values.
- Types of algorithms:
  - Linear (we have discussed in unit 2)
  - binary, and
  - hashing

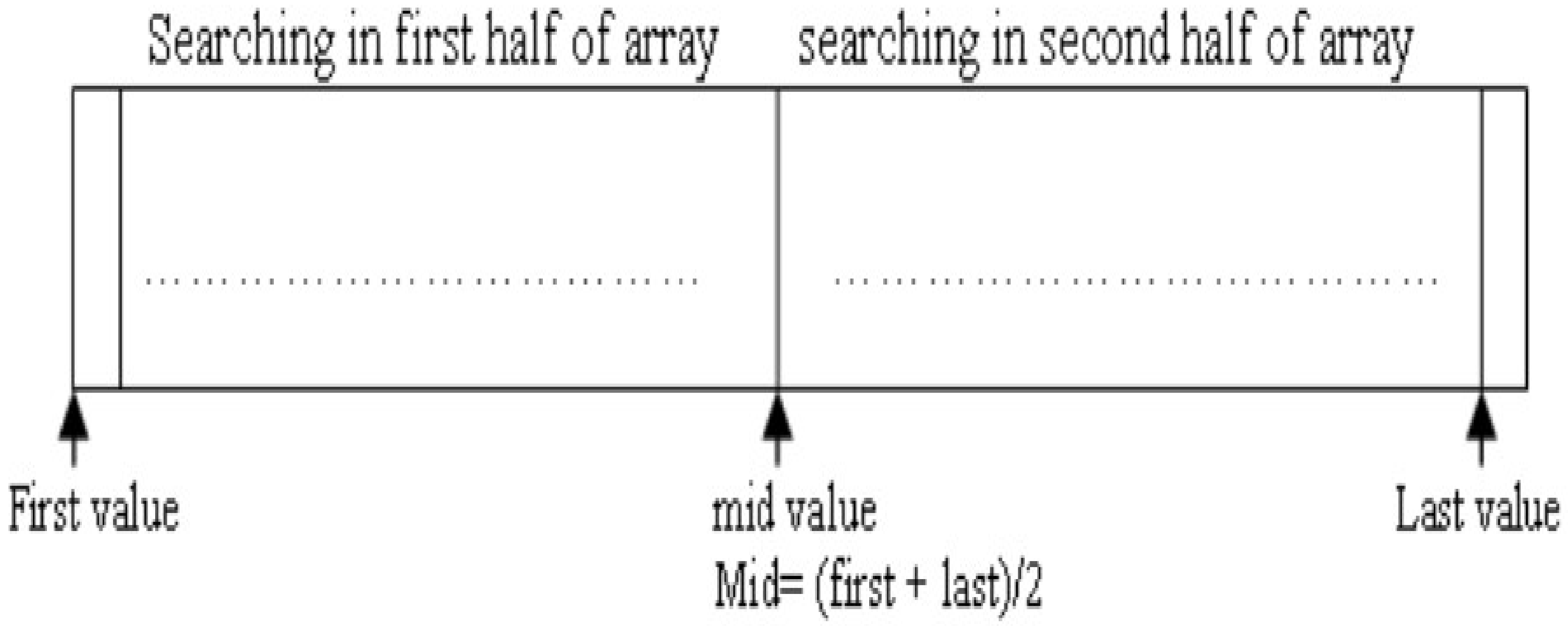
Unit-3:Divide and Conquer Algorithms

Binary search:

- Binary search, also known as half-interval search or logarithmic search or binary chop is a search algorithm that finds the position of a target value within a sorted array.
- A binary search is an advanced type of search algorithm that finds and fetches data from a sorted list of items.
- Its core working principle involves dividing the data in the list to half until the required value is located and displayed to the user in the search result.

Unit-3:Divide and Conquer Algorithms

Binary search:


Searching in first half of array
searching in second half of array
First value
mid value
Mid= (first + last)/2
Last value

Unit-3:Divide and Conquer Algorithms

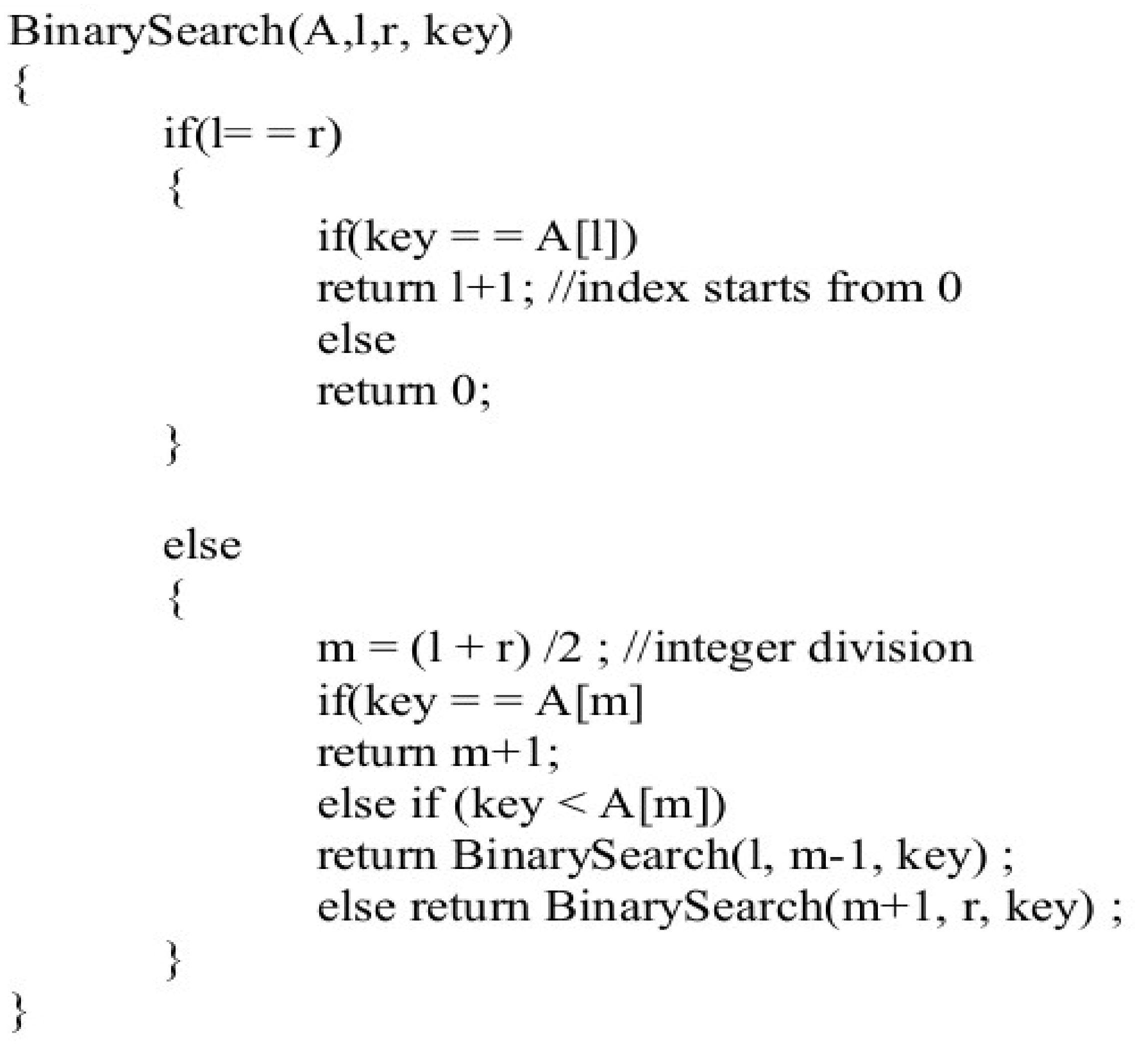
How Binary Search Works?

The binary search works in the following manner:

- The search process initiates by locating the middle element of the sorted array of data.
- After that, the key value is compared with the element:
  - If it is the desired value then the search is successful
  - If the key value is smaller than the search only in first half of the array.
  - In case the key value is greater than searching is carried in second half of the array.

Unit-3:Divide and Conquer Algorithms

- Pseudo code:

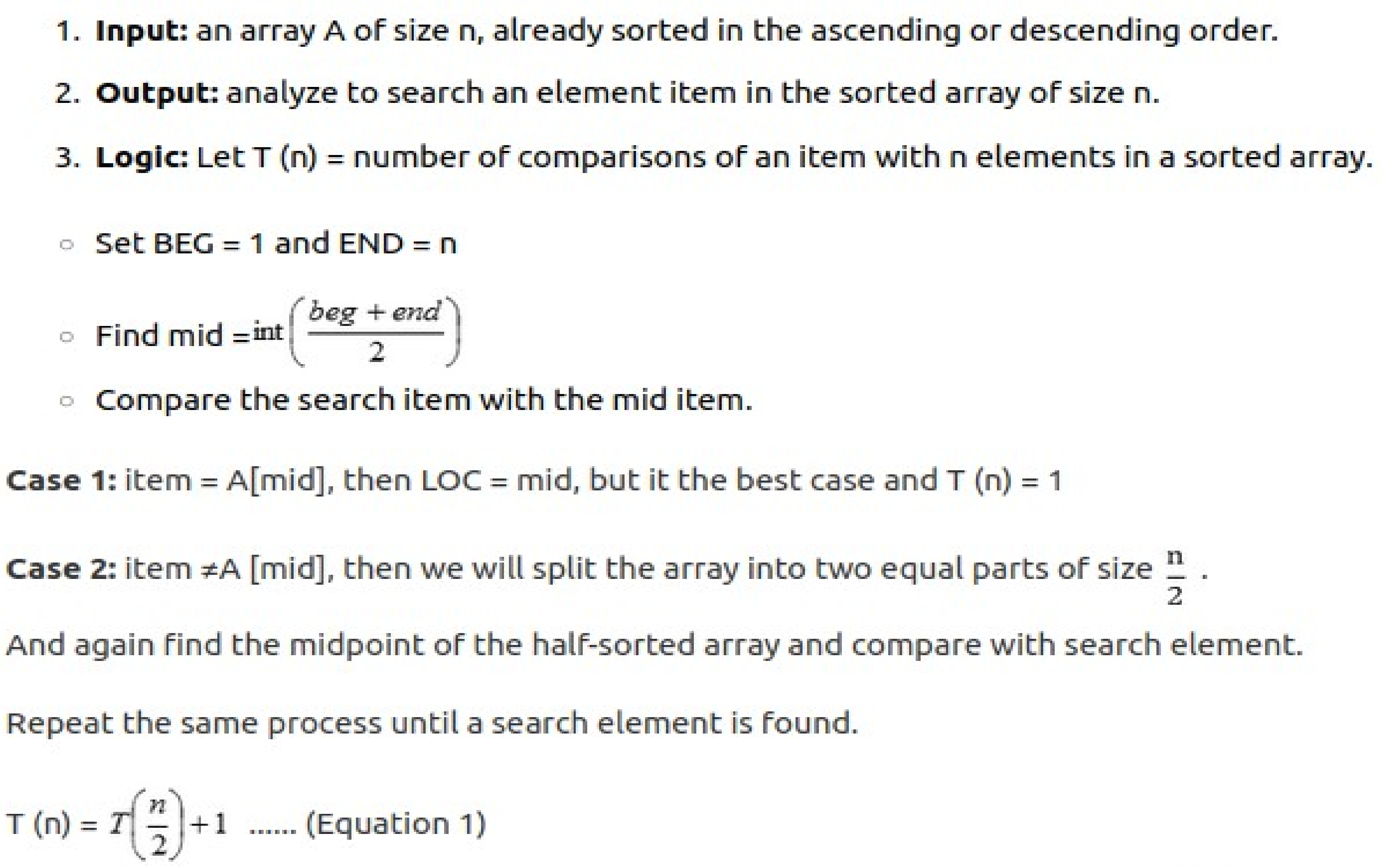


```
BinarySearch(A,l,r, key)
{
        if(l= = r)
        {
                if(key = = A[l])
                return l+1; //index starts from 0
                else
                return 0;
        }

        else
        {
                m = (l + r) /2 ; //integer division
                if(key = = A[m]
                return m+1;
                else if (key < A[m])
                return BinarySearch(l, m-1, key) ;
                else return BinarySearch(m+1, r, key) ;
        }
}
```

Unit-3:Divide and Conquer Algorithms

Analysis:



1. **Input:** an array A of size n, already sorted in the ascending or descending order.
2. **Output:** analyze to search an element item in the sorted array of size n.
3. **Logic:** Let T (n) = number of comparisons of an item with n elements in a sorted array.

   - Set BEG = 1 and END = n
   - Find mid = $\text{int}\left(\frac{beg + end}{2}\right)$
   - Compare the search item with the mid item.

Case 1: item = A[mid], then LOC = mid, but it the best case and T (n) = 1

Case 2: item ≠A [mid], then we will split the array into two equal parts of size $\frac{n}{2}$.

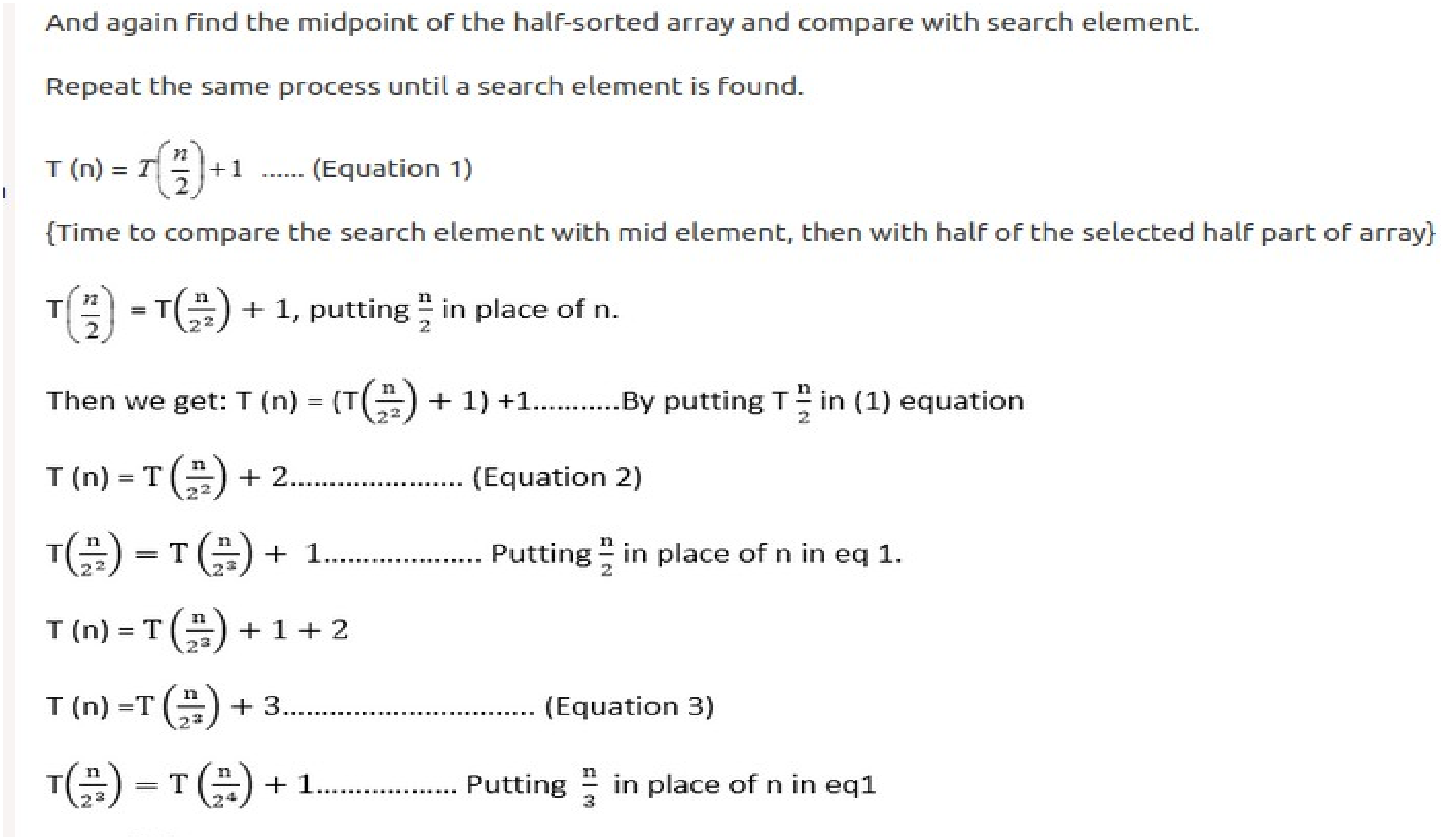
And again find the midpoint of the half-sorted array and compare with search element.

Repeat the same process until a search element is found.

T (n) = $T\left(\frac{n}{2}\right)+1$ ...... (Equation 1)

Unit-3:Divide and Conquer Algorithms

Analysis:



And again find the midpoint of the half-sorted array and compare with search element.

Repeat the same process until a search element is found.

$T(n) = T\left(\frac{n}{2}\right) + 1$ ...... (Equation 1)

{Time to compare the search element with mid element, then with half of the selected half part of array}

$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + 1$, putting $\frac{n}{2}$ in place of n.

Then we get: $T(n) = (T\left(\frac{n}{2^2}\right) + 1) + 1$..........By putting $T\frac{n}{2}$ in (1) equation

$T(n) = T\left(\frac{n}{2^2}\right) + 2$...................... (Equation 2)

$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + 1$.................... Putting $\frac{n}{2}$ in place of n in eq 1.

$T(n) = T\left(\frac{n}{2^3}\right) + 1 + 2$

$T(n) = T\left(\frac{n}{2^3}\right) + 3$............................... (Equation 3)

$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1$.................. Putting $\frac{n}{3}$ in place of n in eq1

Unit-3:Divide and Conquer Algorithms

Analysis:

$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1$.................. Putting $\frac{n}{3}$ in place of n in eq1

Put $T\left(\frac{n}{2^3}\right)$ in eq (3)

$T(n) = T\left(\frac{n}{2^4}\right) + 4$

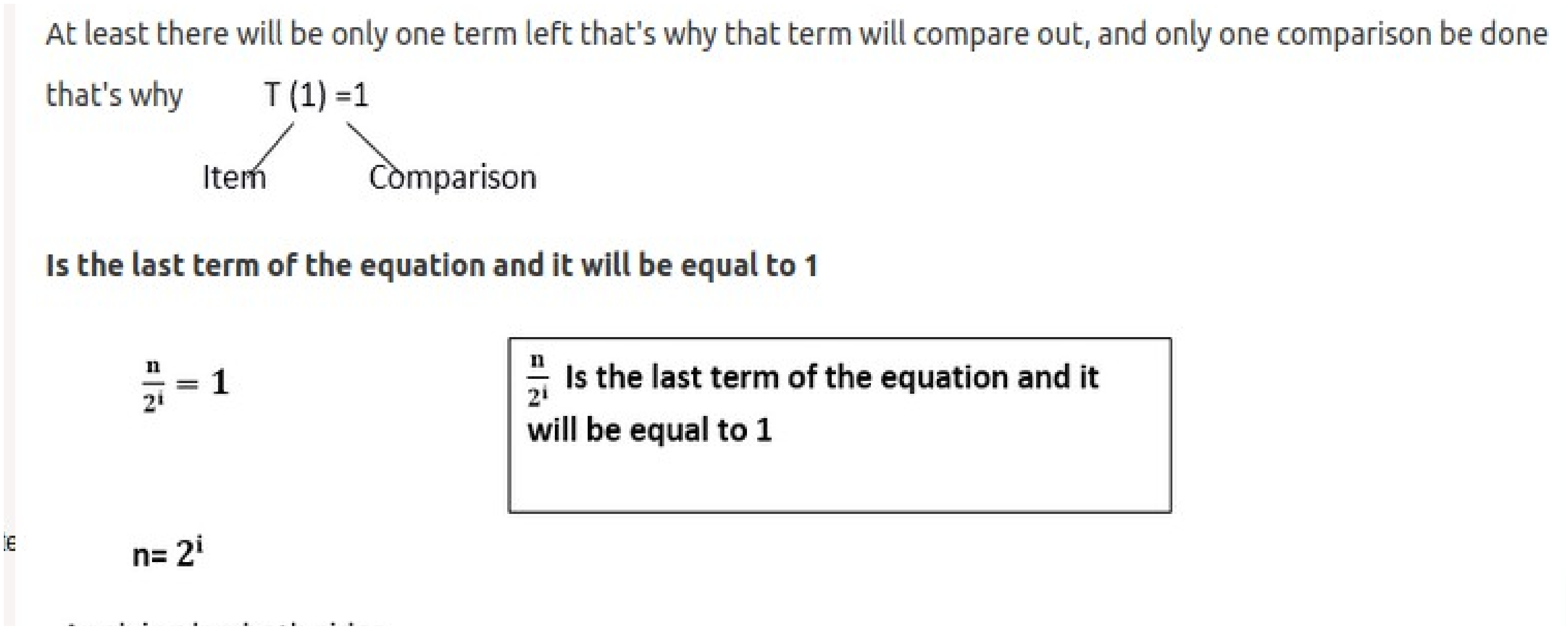
Repeat the same process ith times

$T(n) = T\left(\frac{n}{2^i}\right) + i$ .... ..

Stopping Condition: T (1) =1

Unit-3:Divide and Conquer Algorithms

Analysis:



At least there will be only one term left that's why that term will compare out, and only one comparison be done

that's why T (1) =1

Item Comparison

Is the last term of the equation and it will be equal to 1

$$\frac{n}{2^i} = 1$$

$\frac{n}{2^i}$ **Is the last term of the equation and it will be equal to 1**

n= 2^i

Unit-3:Divide and Conquer Algorithms

Analysis:

Applying log both sides

$\log n = \log_2 i$

$\text{Log } n = i \log 2$

$\frac{\log n}{\log 2} = 1$

$\log_2 n = i$

$T(n) = T\left(\frac{n}{2^i}\right) + i$

$\boxed{\frac{n}{2^i} = 1 \text{ as in eq 5}}$

$= T(1) + i$

$= 1 + i$.................................... T (1) =1 by stopping condition

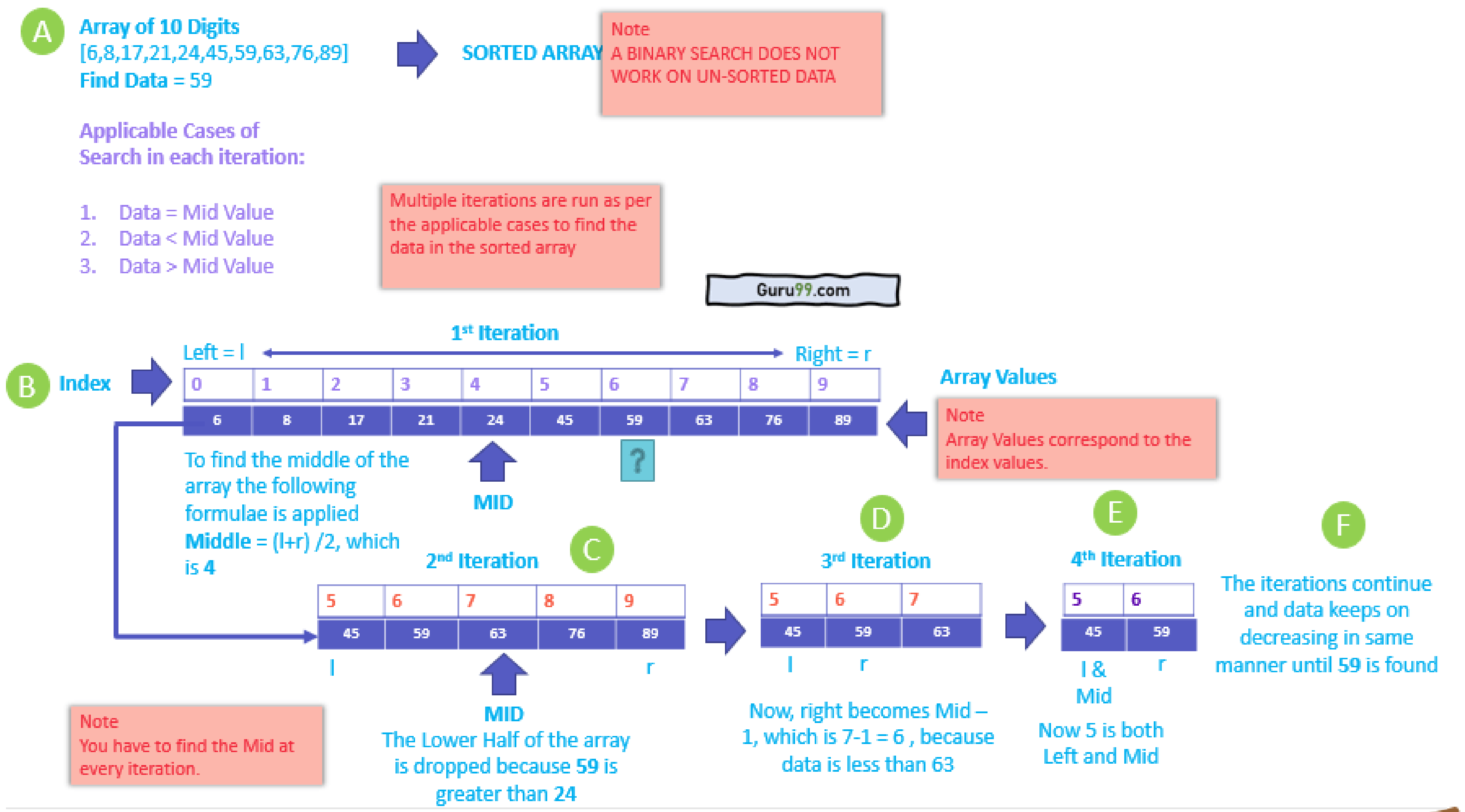
$= 1 + \log_2 n$

$= \log_2 n$ .... (1 is a constant that's why ignore it)

Therefore, binary search is of order o ($\log_2 n$)

Unit-3:Divide and Conquer Algorithms

CONCEPT DIAGRAM



A Array of 10 Digits
[6,8,17,21,24,45,59,63,76,89]
Find Data = 59

SORTED ARRAY

Note
A BINARY SEARCH DOES NOT WORK ON UN-SORTED DATA

Applicable Cases of Search in each iteration:

1. Data = Mid Value
2. Data < Mid Value
3. Data > Mid Value

Multiple iterations are run as per the applicable cases to find the data in the sorted array

Guru99.com

1st Iteration

Left = l — Right = r

B Index

0	1	2	3	4	5	6	7	8	9
6	8	17	21	24	45	59	63	76	89

Array Values

Note
Array Values correspond to the index values.

To find the middle of the array the following formulae is applied **Middle** = (l+r) /2, which is **4**

?

MID

C 2nd Iteration

5	6	7	8	9
45	59	63	76	89

l — r

MID
The Lower Half of the array is dropped because 59 is greater than 24

D 3rd Iteration

5	6	7
45	59	63

l — r

Now, right becomes Mid – 1, which is 7-1 = 6 , because data is less than 63

E 4th Iteration

5	6
45	59

l & Mid — r

Now 5 is both Left and Mid

F The iterations continue and data keeps on decreasing in same manner until 59 is found

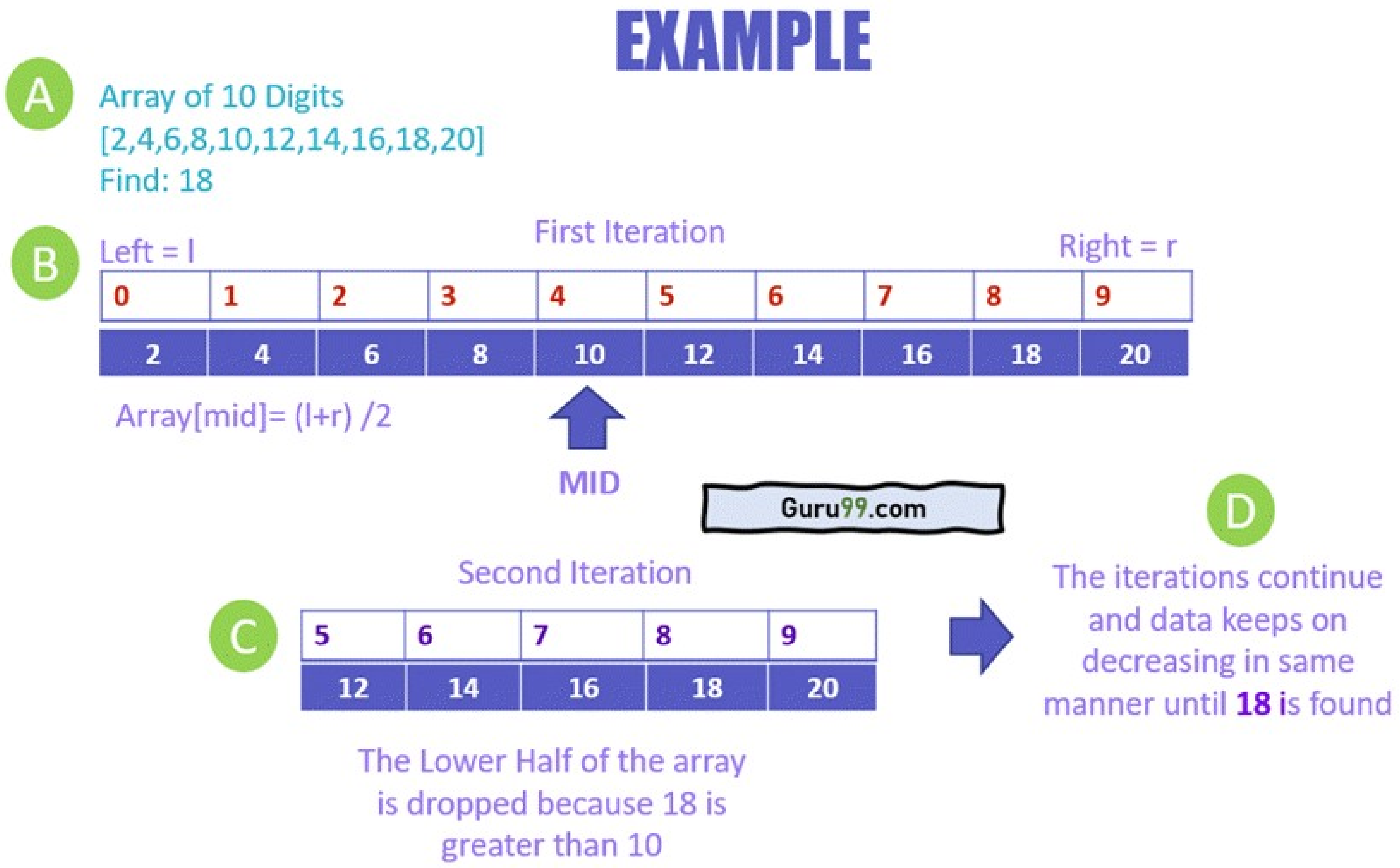
Note
You have to find the Mid at every iteration.

Unit-3:Divide and Conquer Algorithms

The above image illustrates the following:

- You have an array of 10 digits, and the element 59 needs to be found.
- All the elements are marked with the index from 0 – 9. Now, the middle of the array is calculated. To do so, you take the left and rightmost values of the index and divide them by 2.The result is 4.5, but we take the floor value. Hence the middle is 4.
- The algorithm drops all the elements from the middle (4) to the lowest bound because 59 is greater than 24, and now the array is left with 5 elements only.
- Now, 59 is greater than 45 and less than 63. The middle is 7. Hence the right index value becomes middle – 1, which equals 6, and the left index value remains the same as before, which is 5.
- At this point, you know that 59 comes after 45. Hence, the left index, which is 5, becomes mid as well.
- These iterations continue until the array is reduced to only one element, or the item to be found becomes the middle of the array.

Unit-3:Divide and Conquer Algorithms



Unit-3:Divide and Conquer Algorithms

The above image illustrates the following:

- You have an array of sorted values ranging from 2 to 20 and need to locate 18.
- The average of the lower and upper limits is (l + r) / 2 = 4. The value being searched is greater than the mid which is 4.
- The array values less than the mid are dropped from search and values greater than the mid-value 4 are searched.
- This is a recurrent dividing process until the actual item to be searched is found.

Unit-3:Divide and Conquer Algorithms

Max-Min Algorithm:

- The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.
- To find the maximum and minimum numbers in a given array numbers [ ] of size n, the following algorithm can be used.
  - First we are representing the naive method and
  - then we will present divide and conquer approach.

Unit-3:Divide and Conquer Algorithms

Max-Min Algorithm: Naïve Method:

- Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

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)
```

Unit-3:Divide and Conquer Algorithms

Max-Min Algorithm: Naïve Method:

Analysis:

- The number of comparison in Naive method is 2n – 2.
  - So Time complexity O(n).

- The number of comparisons can be reduced using the divide and conquer approach.

Unit-3:Divide and Conquer Algorithms

Max-Min Algorithm: Divide and Conquer Approach

- In this approach, the array is divided into two halves.
- Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.
- In this given problem, the number of elements in an array is y−x+1, where y is greater than or equal to x.
- Max−Min(x,y) will return the maximum and minimum values of an array numbers[x...y].

Unit-3:Divide and Conquer Algorithms

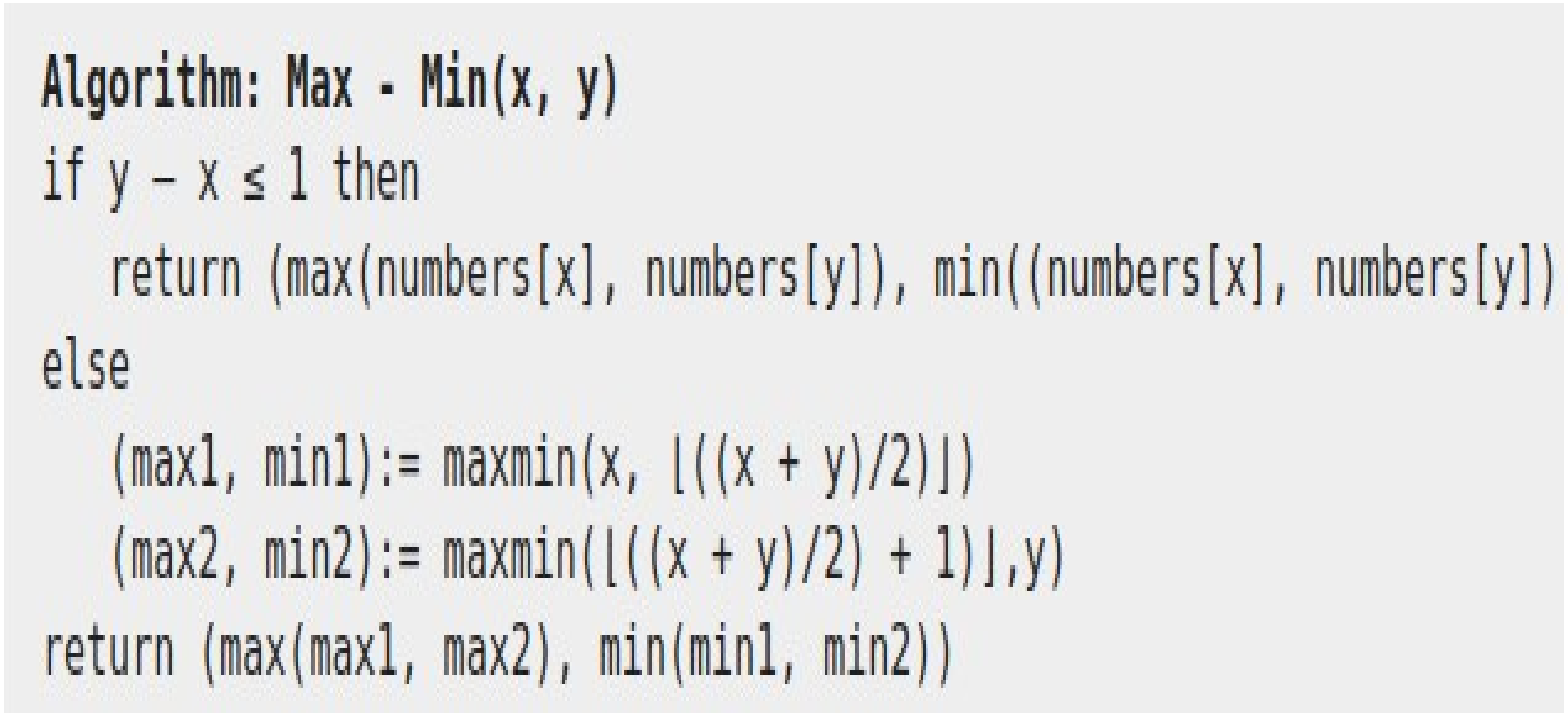
Max-Min Algorithm: Divide and Conquer Approach

```
Pair MaxMin(array, array_size)
  if array_size = 1
    return element as both max and min
  else if arry_size = 2
    one comparison to determine max and min
    return that pair
  else    /* array_size  > 2 */
    recur for max and min of left half
    recur for max and min of right half
    one comparison determines true max of the two candidates
    one comparison determines true min of the two candidates
    return the pair of max and min
```

Unit-3:Divide and Conquer Algorithms

Max-Min Algorithm: Divide and Conquer Approach

Pseudo code:

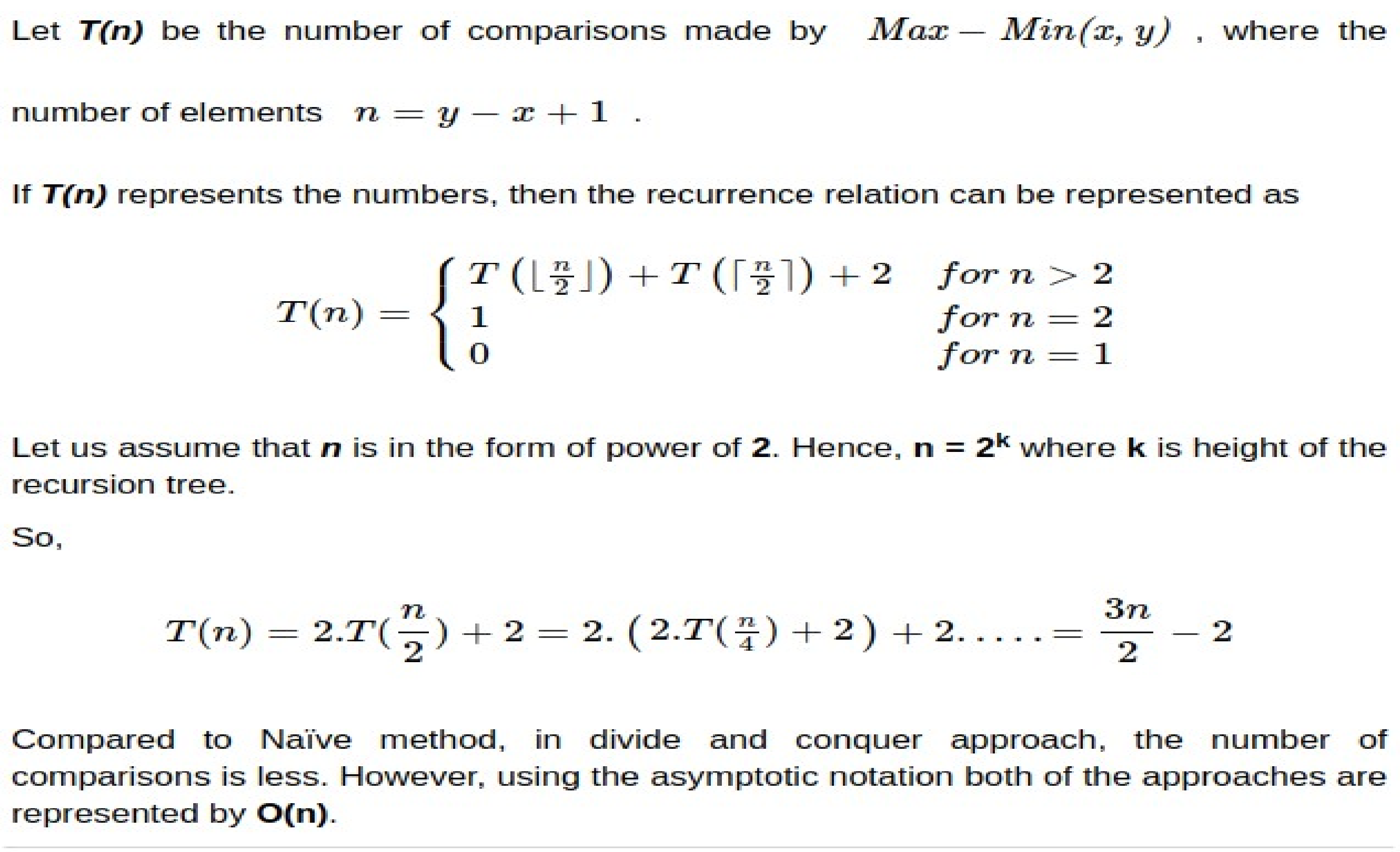


```
Algorithm: Max - Min(x, y)
if y – x ≤ 1 then
    return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y])
else
    (max1, min1):= maxmin(x, ⌊((x + y)/2)⌋)
    (max2, min2):= maxmin(⌊((x + y)/2) + 1)⌋,y)
return (max(max1, max2), min(min1, min2))
```

Unit-3:Divide and Conquer Algorithms

Max-Min Algorithm: Divide and Conquer Approach

Analysis:



Let ***T(n)*** be the number of comparisons made by $Max - Min(x, y)$, where the number of elements $n = y - x + 1$.

If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

$$T(n) = \begin{cases} T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 2 & for\ n > 2 \\ 1 & for\ n = 2 \\ 0 & for\ n = 1 \end{cases}$$

Let us assume that ***n*** is in the form of power of **2**. Hence, **n = 2^k** where **k** is height of the recursion tree.

So,

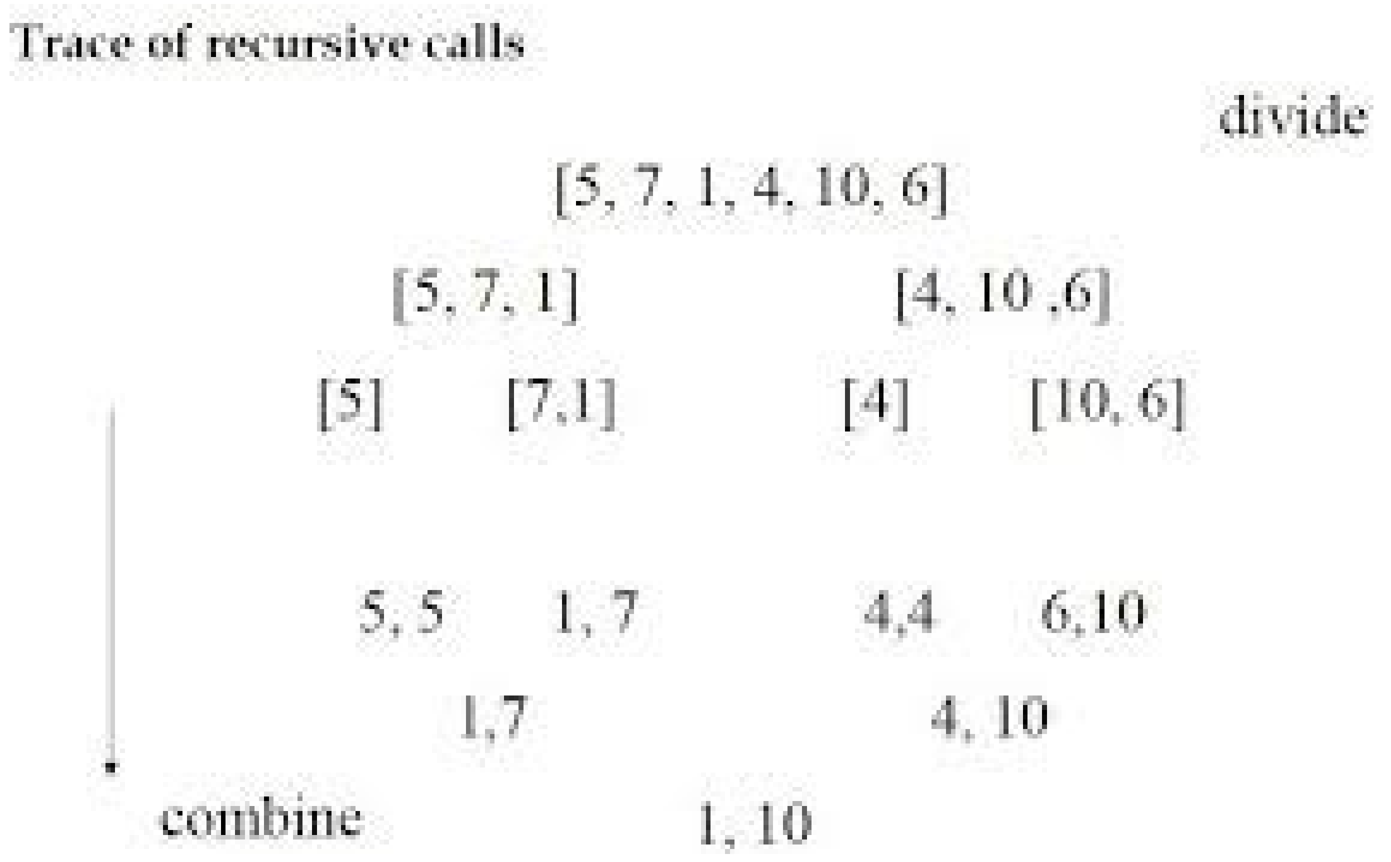
$$T(n) = 2.T(\frac{n}{2}) + 2 = 2.\left(2.T(\frac{n}{4}) + 2\right) + 2.\ldots.. = \frac{3n}{2} - 2$$

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)**.

Unit-3:Divide and Conquer Algorithms

Max-Min Algorithm: Divide and Conquer Approach

Examples:



Unit-3:Divide and Conquer Algorithms

Merge Sort:

- It is one of the well-known divide-and-conquer algorithm. This is a simple and very efficient algorithm for sorting a list of numbers.
- It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.
- The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

Unit-3:Divide and Conquer Algorithms

Merge Sort:

To sort an array A[l . . r]:

- Divide

– Divide the n-element sequence to be sorted into two subsequences of n/2 elements each

- Conquer

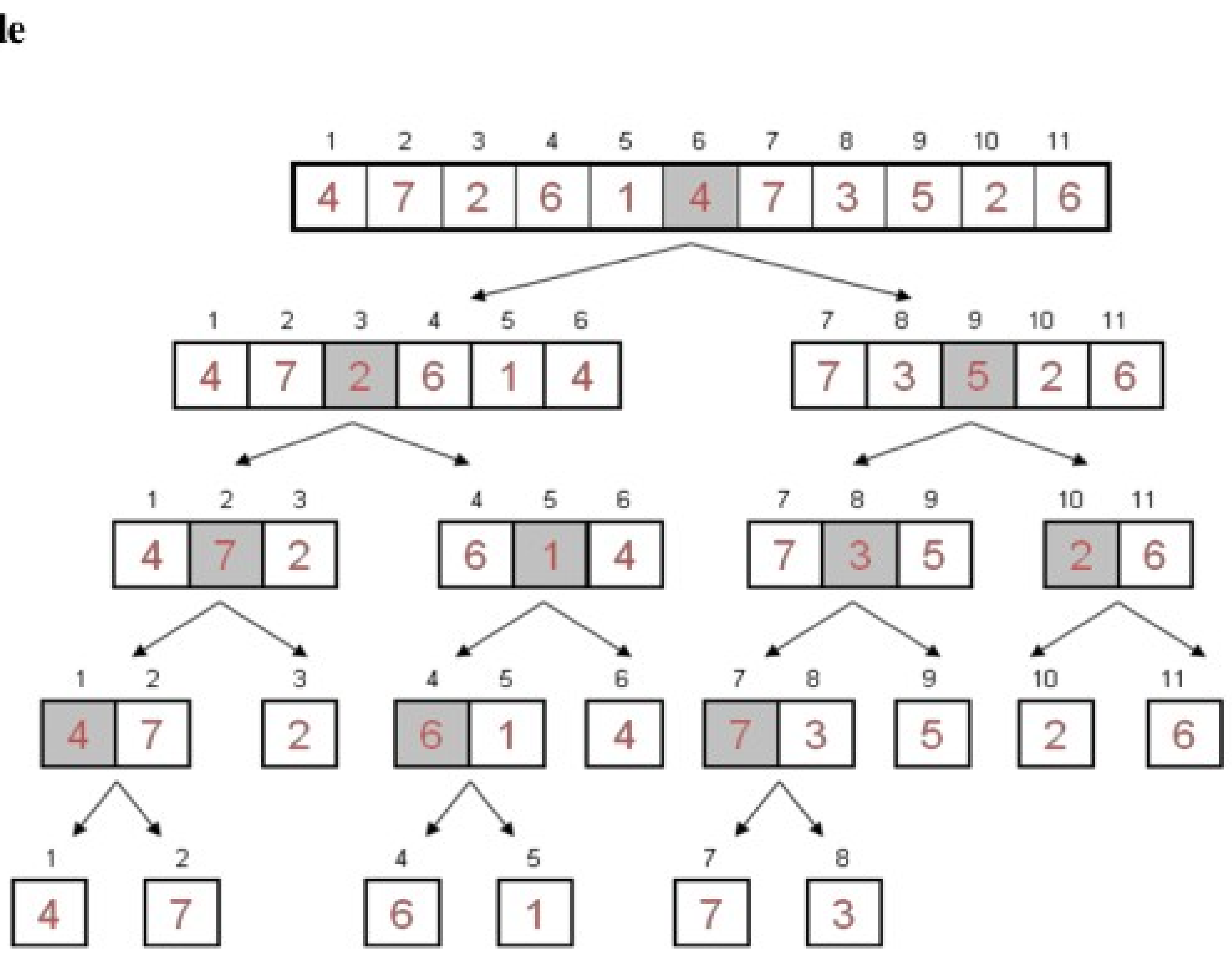
– Sort the subsequences recursively using merge sort. When the size of the sequences is 1 there is nothing more to do

- Combine

– Merge the two sorted subsequences

Unit-3:Divide and Conquer Algorithms

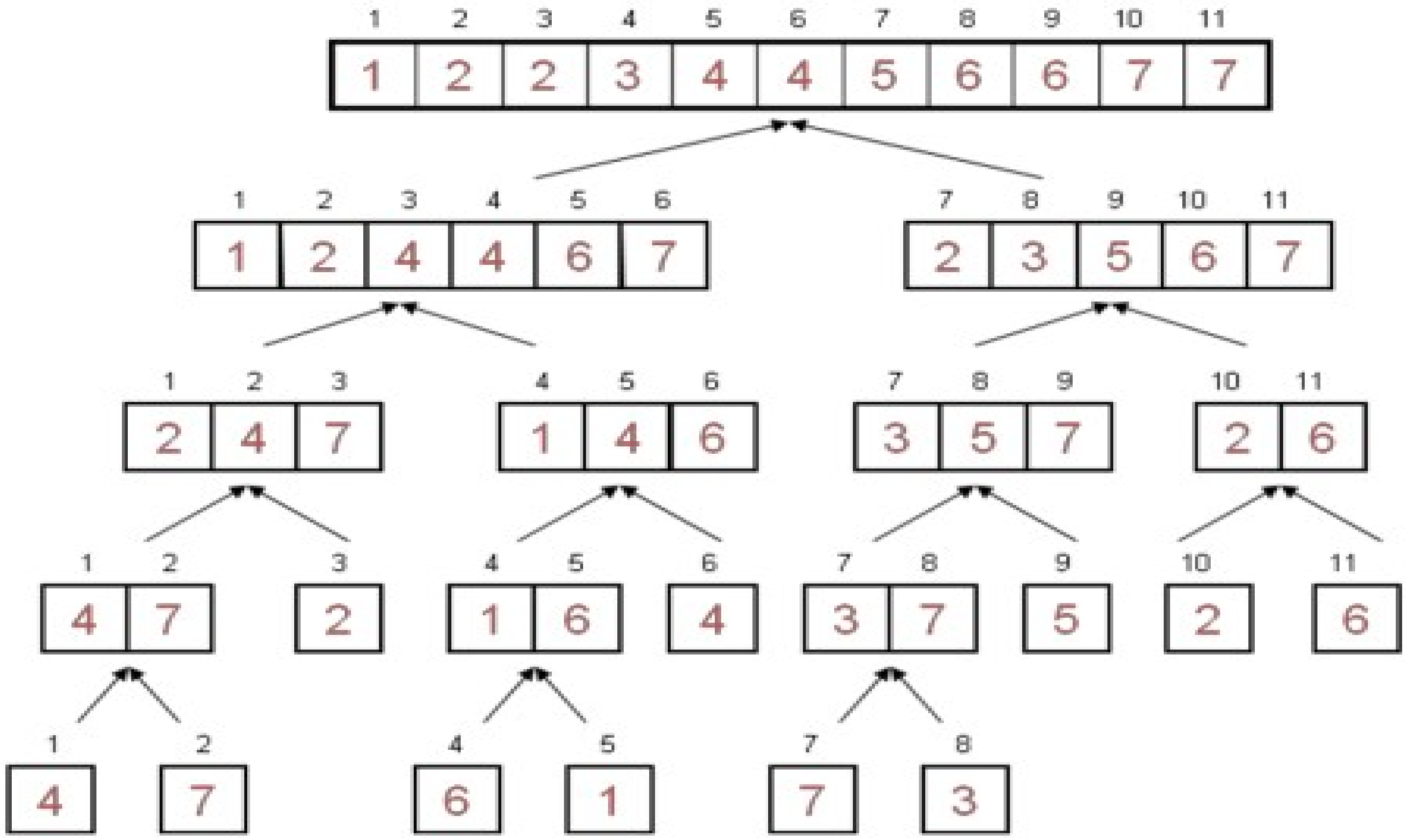
Merge Sort:



Unit-3:Divide and Conquer Algorithms

Merge Sort:

Merging:



Unit-3:Divide and Conquer Algorithms

Merge Sort: Pseudo code

```
MergeSort(A, l, r)
{
    If ( l < r)
    {                              //Check for base case
      m = ⌊(l + r)/2⌋              //Divide
        MergeSort(A, l, m)         //Conquer
        MergeSort(A, m + 1, r)     //Conquer
        Merge(A, l, m+1, r)        //Combine
    }
}
```

```
Merge(A,B,l,m,r)
{
    x=l, y=m;
    k=l;
    while(x<m && y<r)
    {
        if(A[x] < A[y])
        {
            B[k]= A[x];
            k++; x++;
        }
        else
        {
            B[k] = A[y];
            k++; y++;
        }
    }
    while(x<m)
    {
        A[k] = A[x];
        k++; x++;
    }
    while(y<r)
    {
        A[k] = A[y];
        k++; y++;
    }
    for(i=l;i<= r; i++)
    {
        A[i] = B[i]
    }
}
```

Unit-3:Divide and Conquer Algorithms

Merge Sort:

- Time Complexity:
  - NO of sub problems=2
  - Size of each subproblem =n/2
  - Dividing cost is constant for each subproblems
  - Merging cost is n
  - So recurrence relations is:
    - T(n) =1 if n=1
    - T(n) = 2T(n/2)+O(n) if n>1
  - Solving this we get, T(n)=O(nlogn)
- **Space complexity:**
  - Auxiliary Space taken is O(n)

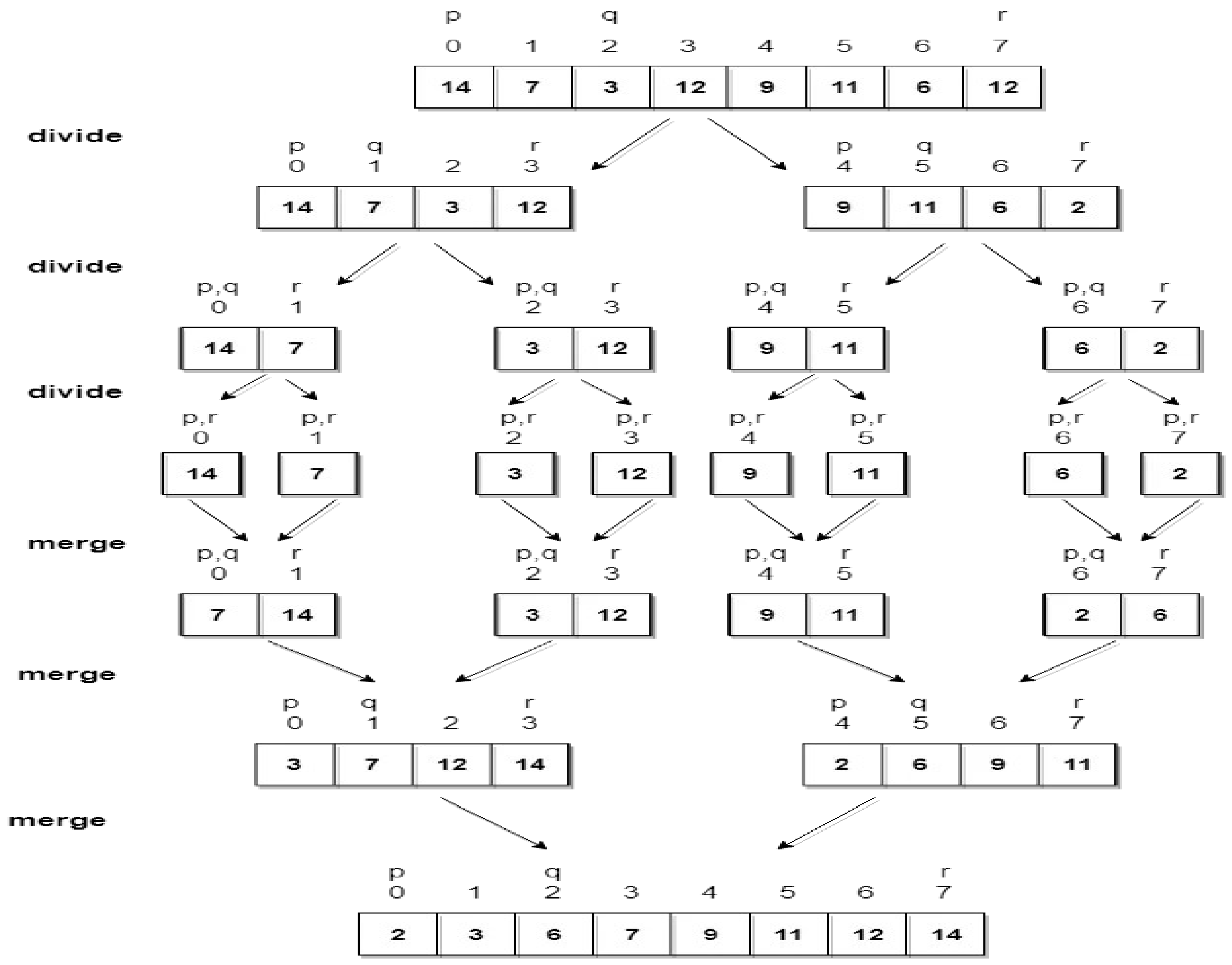
Unit-3:Divide and Conquer Algorithms

Merge Sort: Examples

- Let's consider an array with values {14, 7, 3, 12, 9, 11, 6, 12}
- Below, we have a pictorial representation of how merge sort will sort the given array.

Unit-3:Divide and Conquer Algorithms

Merge Sort: Examples



Unit-3:Divide and Conquer Algorithms

Merge Sort: Examples

In merge sort we follow the following steps:

- We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
- Then we find the middle of the array using the formula (p + r)/2 and mark the middle index as q, and break the array into two subarrays, from p to q and from q + 1 to r index.
- Then we divide these 2 subarrays again, just like we divided our main array and this continues.
- Once we have divided the main array into subarrays with single elements, then we start merging the subarrays.

Unit-3:Divide and Conquer Algorithms

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
  - Pick a random element as pivot.
  - Pick median as pivot.

Unit-3:Divide and Conquer Algorithms

Quick Sort:

- Quick Sort is one of the different Sorting Technique which is based on the concept of Divide and Conquer, just like merge sort.
- But in quick sort all the heavy lifting(major work) is done while dividing the array into subarrays, while in case of merge sort, all the real work happens during merging the subarrays.
- In case of quick sort, the combine step does absolutely nothing.
- It is also called partition-exchange sort. This algorithm divides the list into three main parts:
  - Elements less than the Pivot element
  - Pivot element(Central element)
  - Elements greater than the pivot element

Unit-3:Divide and Conquer Algorithms

Quick Sort:

- Divide

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

- 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.

Unit-3:Divide and Conquer Algorithms

Quick Sort:

Technically, quick sort follows the below steps:

- Step 1 − Make any element as pivot
- Step 2 − Partition the array on the basis of pivot
- Step 3 − Apply quick sort on left partition recursively
- Step 4 − Apply quick sort on right partition recursively

Unit-3:Divide and Conquer Algorithms

Quick Sort:

- Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as pivot.
- For example: In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take **25** as pivot. So after the first pass, the list will be changed like this.

  {6 8 17 14 **25** 63 37 52}
- Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate sunarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

Unit-3:Divide and Conquer Algorithms

Quick Sort:Pseudo Code

```
QuickSort(A,l,r)
{
    if(l<r)
    {
        p = Partition(A,l,r);
        QuickSort(A,l,p-1);
        QuickSort(A,p+1,r);
    }
}
```

```
Partition(A,l,r)
{
x =l; y =r ; p = A[l];
while(x<y)
{
        do {
                x++;
        }while(A[x] <= p);
        do {
                y--;
        } while(A[y] >=p);
        if(x<y)
                swap(A[x],A[y]);
}
A[l] = A[y]; A[y] = p; return y;   //return position of pivot
}
```

Unit-3:Divide and Conquer Algorithms

Quick Sort:

- **Time complexity:**
  - **In Worst Case:** The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

    T(n) = T(0) + T(n-1) + O(n) which is equivalent to

    - T(n) = T(n-1) + O(n)
  - The solution of above recurrence is $O(n^2)$

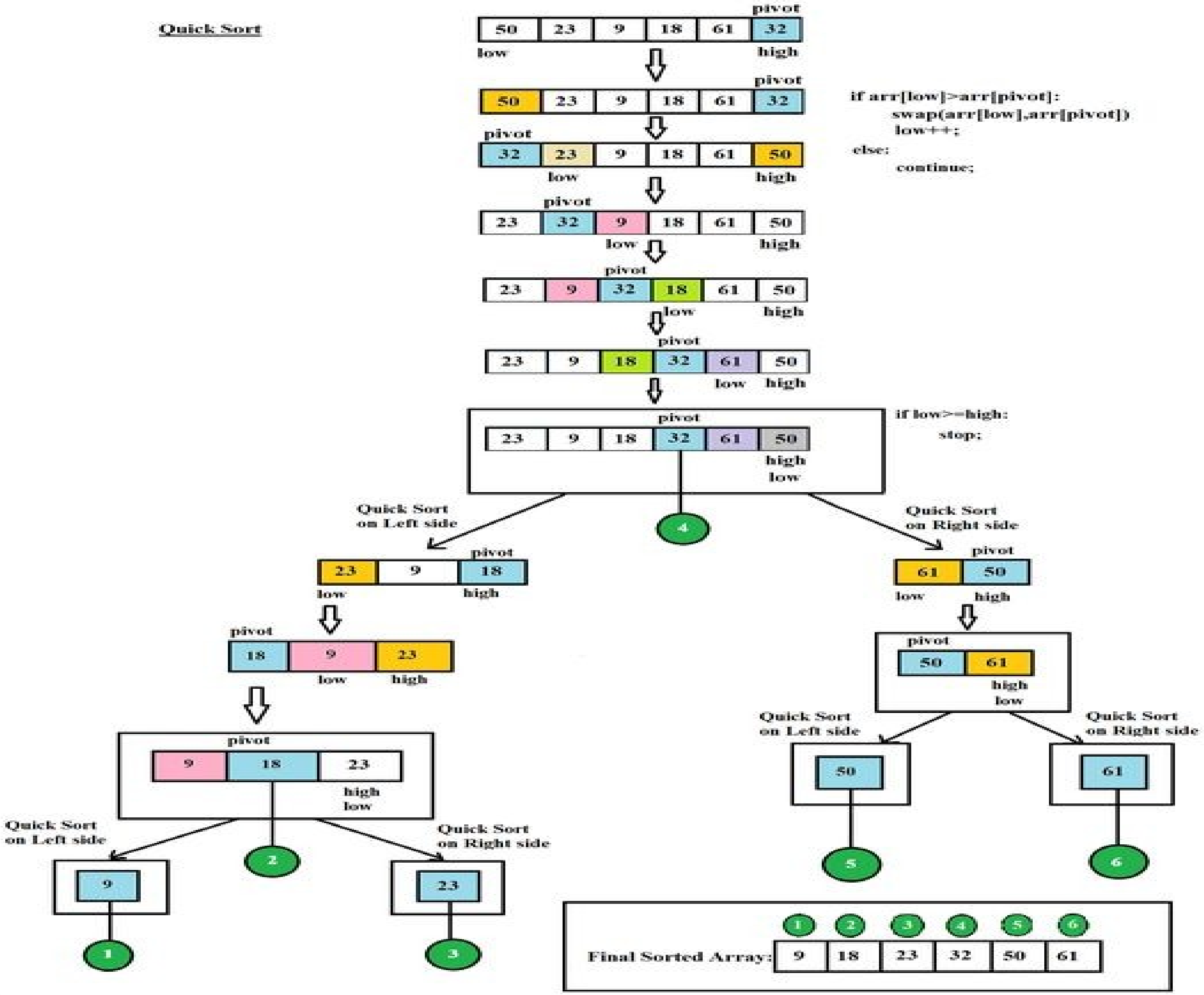
Unit-3:Divide and Conquer Algorithms

Quick Sort: Examples

Problem Statement:

- Consider the following array: 50, 23, 9, 18, 61, 32

Quick Sort



```
if arr[low]>arr[pivot]:
        swap(arr[low],arr[pivot])
        low++;
else:
        continue;
```

```
if low>=high:
    stop;
```

Final Sorted Array: 9 18 23 32 50 61

Unit-3:Divide and Conquer Algorithms

Quick Sort:Examples

Step 1:

- Make any element as pivot: Decide any value to be the pivot from the list. For convenience of code, we often select the rightmost index as pivot or select any at random and swap with rightmost. Suppose for two values “Low” and “High” corresponding to the first index and last index respectively.
- In our case low is 0 and high is 5.
- Values at low and high are 50 and 32 and value at pivot is 32.
- Partition the array on the basis of pivot: Call for partitioning which rearranges the array in such a way that pivot (32) comes to its actual position (of the sorted array). And to the left of the pivot, the array has all the elements less than it, and to the right greater than it.
- In the partition function, we start from the first element and compare it with the pivot. Since 50 is greater than 32, we don’t make any change and move on to the next element 23.
- Compare again with the pivot. Since 23 is less than 32, we swap 50 and 23. The array becomes 23, 50, 9, 18, 61, 32
- We move on to the next element 9 which is again less than pivot (32) thus swapping it with 50 makes our array as 23, 9, 50, 18, 61, 32.
- Similarly, for next element 18 which is less than 32, the array becomes 23, 9, 18, 50, 61, 32. Now 61 is greater than pivot (32), hence no changes.
- Lastly, we swap our pivot with 50 so that it comes to the correct position.
- Thus the pivot (32) comes at its actual position and all elements to its left are lesser, and all elements to the right are greater than itself.

Unit-3:Divide and Conquer Algorithms

Quick Sort:Examples

Step 2:

- The main array after the first step becomes
  - 23, 9, 18, 32, 61, 50

Step 3:

- Now the list is divided into two parts
  - Sublist before pivot element
  - Sublist after pivot element

Step 4:

- Repeat the steps for the left and right sublists recursively. The final array thus becomes
  - 9, 18, 23, 32, 50, 61.

Unit-3:Divide and Conquer Algorithms

Randomized Quick Sort:

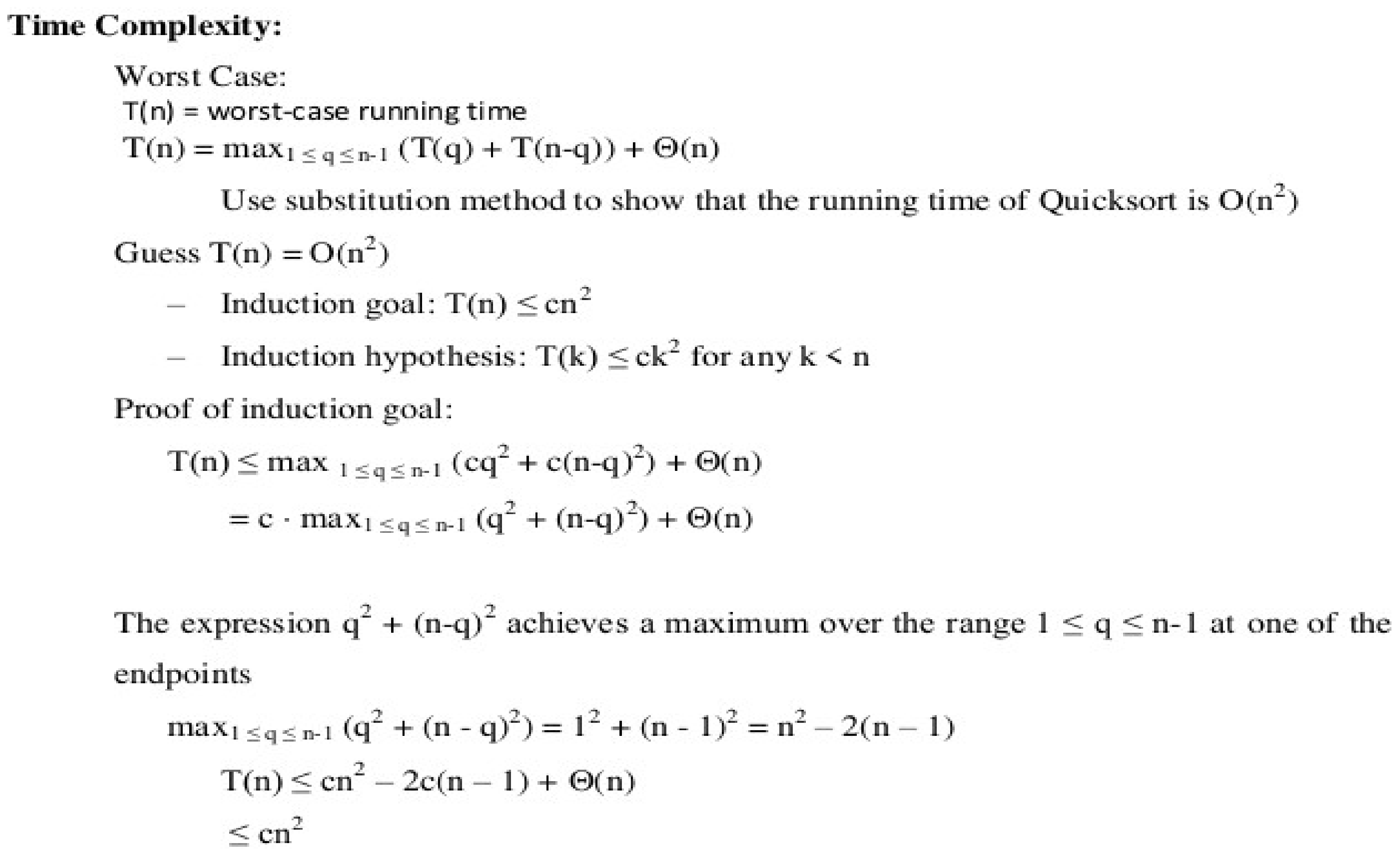
- The algorithm is called randomized if its behavior depends on input as well as random value generated by random number generator.
- The beauty of the randomized algorithm is that no particular input can produce worst-case behavior of an algorithm.
- IDEA: Partition around a random element. Running time is independent of the input order.
- No assumptions need to be made about the input distribution. No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator. Randomization cannot
- eliminate the worst-case but it can make it less likely!

Randomized Quick Sort:

```
RandQuickSort(A,l,r)
{
    if(l<r)
    {
            m = RandPartition(A,l,r);
            RandQuickSort(A,l,m-1);
            RandQuickSort(A,m+1,r);
    }
}
```

```
RandPartition(A,l,r)
{
    k = random(l,r); //generates random number between i and j including both.
    swap(A[l],A[k]);
    return Partition(A,l,r);
}
```

Randomized Quick Sort:



Time Complexity:

Worst Case:

T(n) = worst-case running time

$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n)$

Use substitution method to show that the running time of Quicksort is $O(n^2)$

Guess $T(n) = O(n^2)$

– Induction goal: $T(n) \le cn^2$

– Induction hypothesis: $T(k) \le ck^2$ for any $k < n$

Proof of induction goal:

$$T(n) \le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n)$$

$$= c \cdot \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + \Theta(n)$$

The expression $q^2 + (n-q)^2$ achieves a maximum over the range $1 \le q \le n-1$ at one of the endpoints

$$\max_{1 \le q \le n-1} (q^2 + (n - q)^2) = 1^2 + (n - 1)^2 = n^2 - 2(n - 1)$$

$$T(n) \le cn^2 - 2c(n - 1) + \Theta(n)$$

$$\le cn^2$$

i.e. $T(n) = O(n^2)$

Unit-3:Divide and Conquer Algorithms

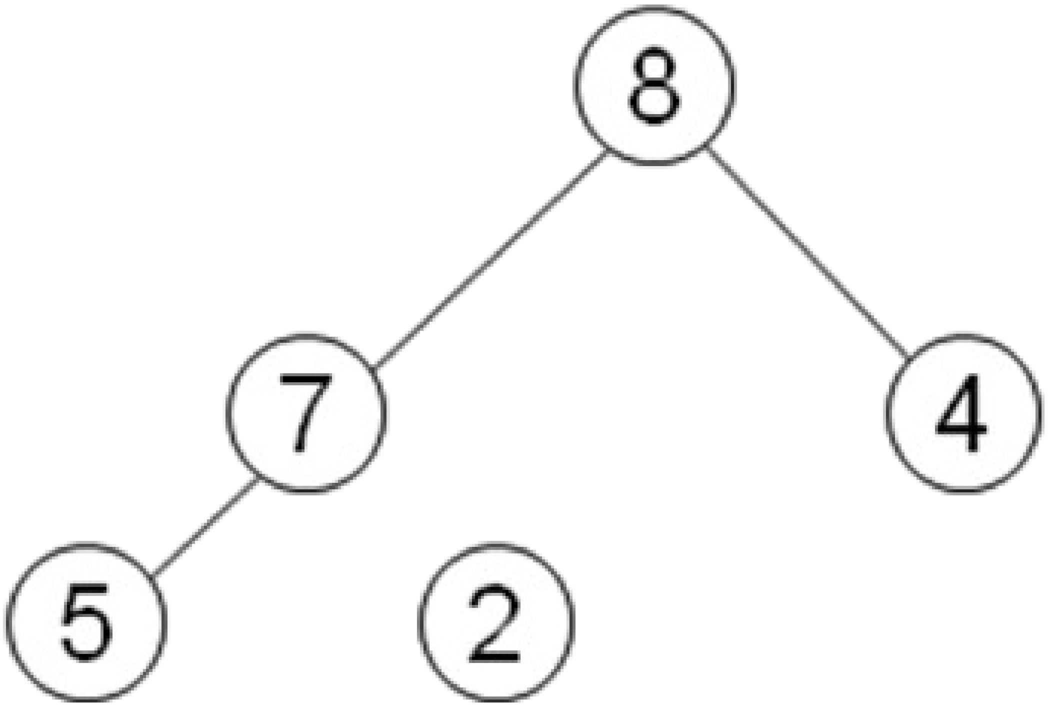
Concept of Heap Data Structures:

- Heap is a special tree-based data structure.
- A binary tree is said to follow a heap data structure if
  - it is a complete binary tree.
  - All nodes in the tree follow the property that they are greater than their children
    - i.e. the largest element is at the root and both its children and smaller than the root and so on. Such a heap is called a max-heap.
    - If instead, all nodes are smaller than their children, it is called a min-heap

Unit-3:Divide and Conquer Algorithms

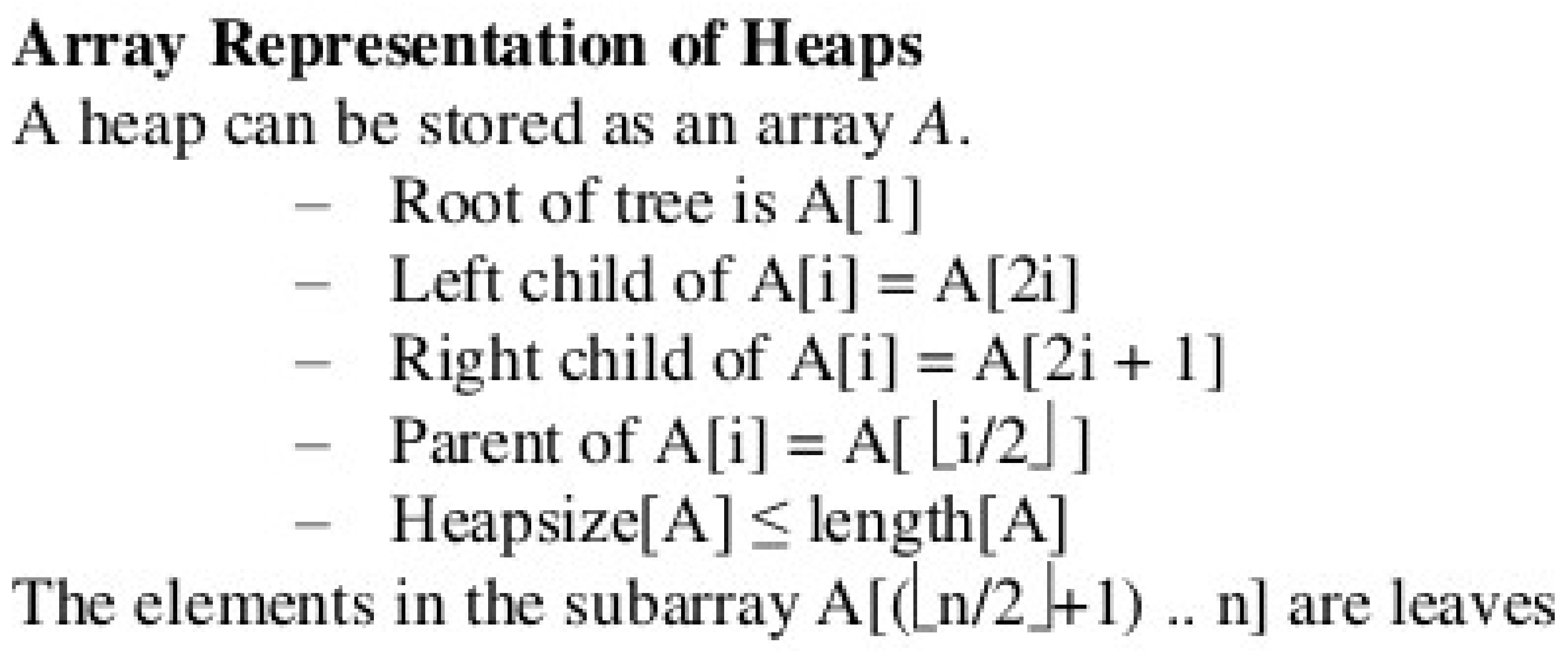
Concept of Heap Data Structures:

- Heap is a special tree-based data structure.



Unit-3:Divide and Conquer Algorithms

Array Representation of Heap:



Array Representation of Heaps

A heap can be stored as an array *A*.

- Root of tree is A[1]
- Left child of A[i] = A[2i]
- Right child of A[i] = A[2i + 1]
- Parent of A[i] = A[$\lfloor i/2 \rfloor$]
- Heapsize[A] ≤ length[A]

The elements in the subarray A[($\lfloor n/2 \rfloor$+1) .. n] are leaves

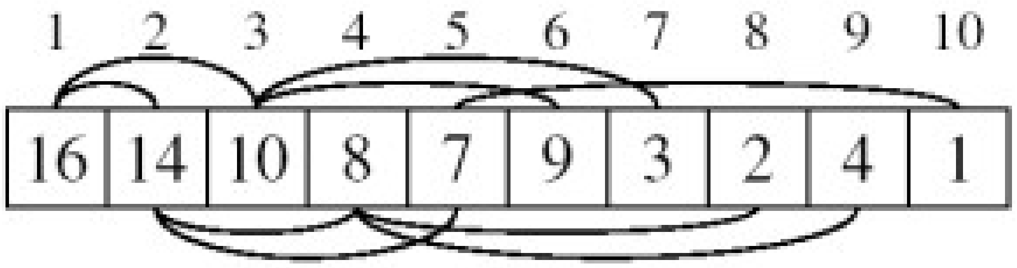
Unit-3:Divide and Conquer Algorithms

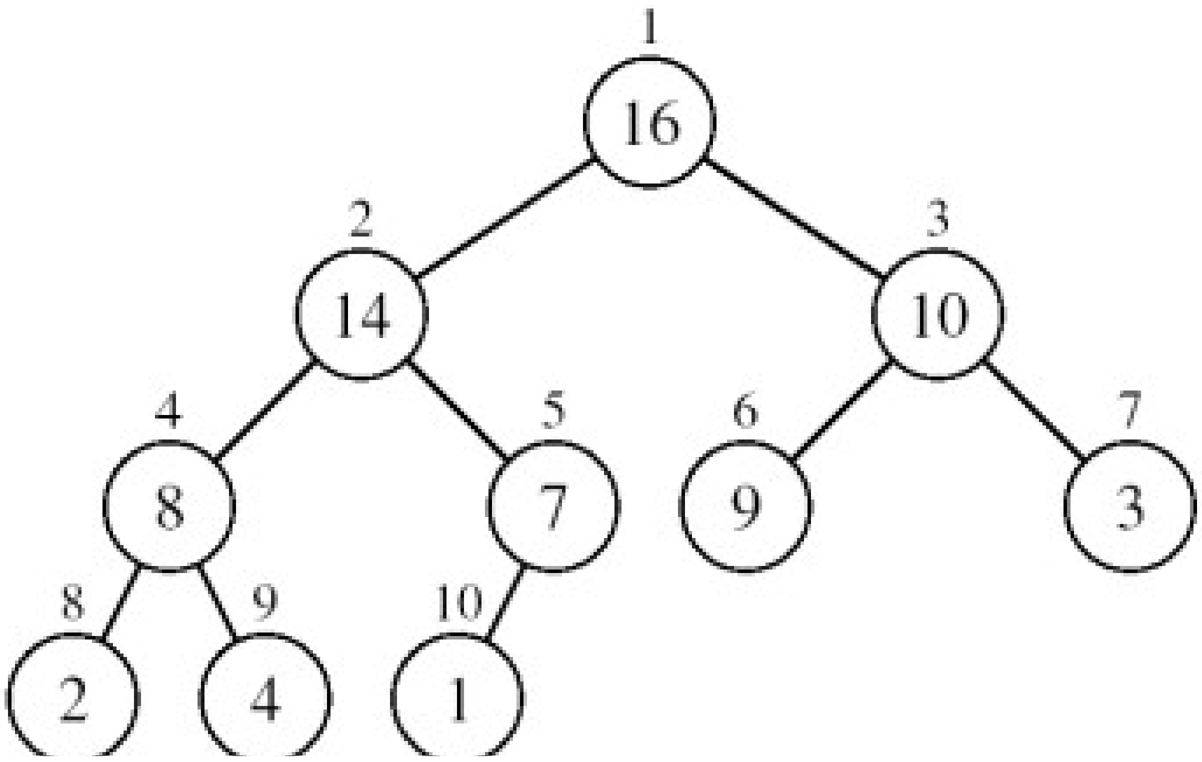
Array Representation of Heap:

- **Why array based representation for Binary Heap?**
- Since a Binary Heap is a Complete Binary Tree, it can be easily represented as an array and the array-based representation is space-efficient.
- If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and the right child by 2 * I + 2 (assuming the indexing starts at 0).

Unit-3:Divide and Conquer Algorithms

Array Representation of Heap:





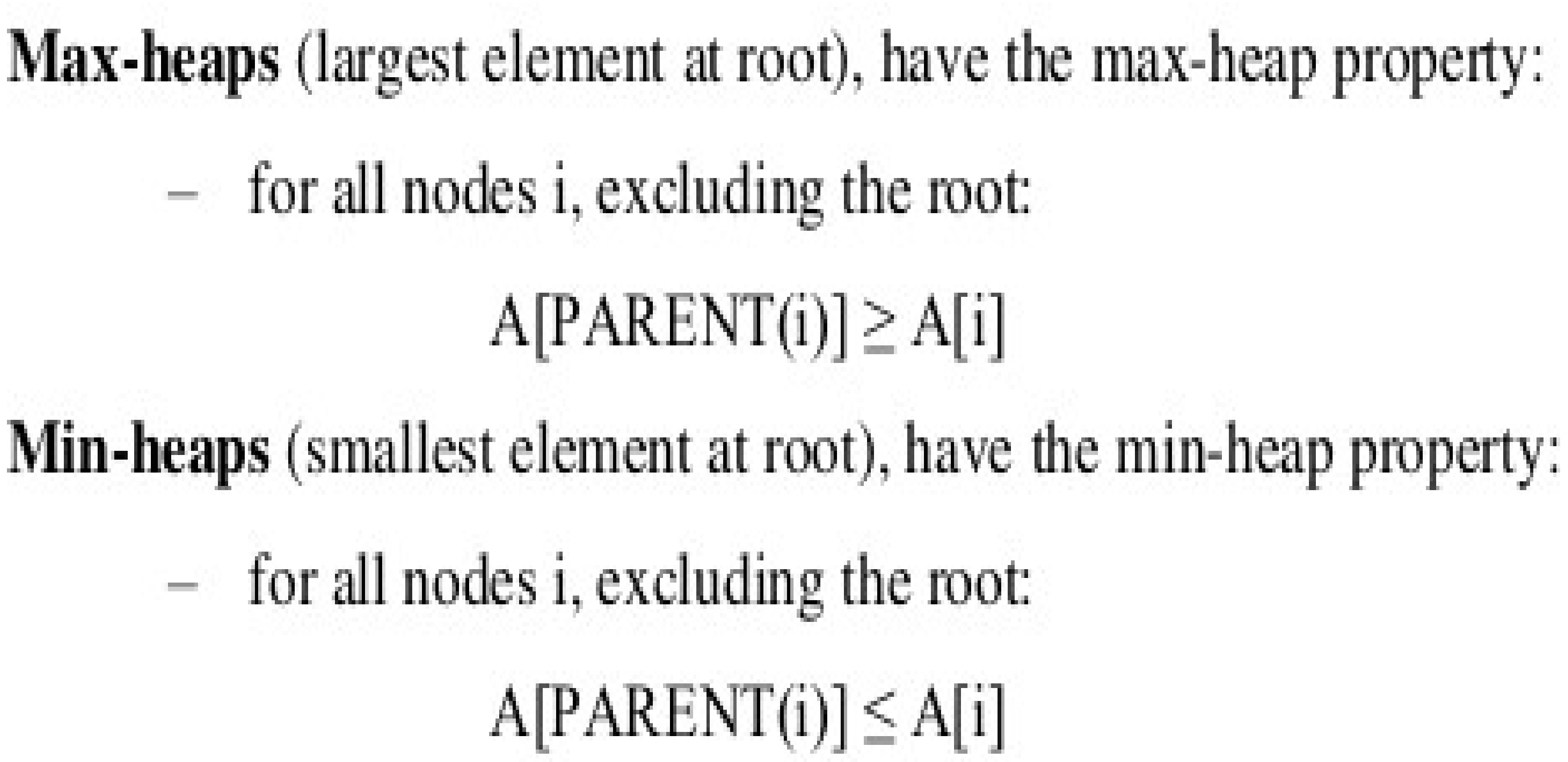
Unit-3:Divide and Conquer Algorithms

Generally, Heaps can be of two types:

- **Max-Heap:**
  - In a Max-Heap the key present at the root node must be greatest among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.
- **Min-Heap:**
  - In a Min-Heap the key present at the root node must be minimum among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.

Unit-3:Divide and Conquer Algorithms

Generally, Heaps can be of two types:



Max-heaps (largest element at root), have the max-heap property:

– for all nodes i, excluding the root:

$$A[\text{PARENT}(i)] \geq A[i]$$

Min-heaps (smallest element at root), have the min-heap property:

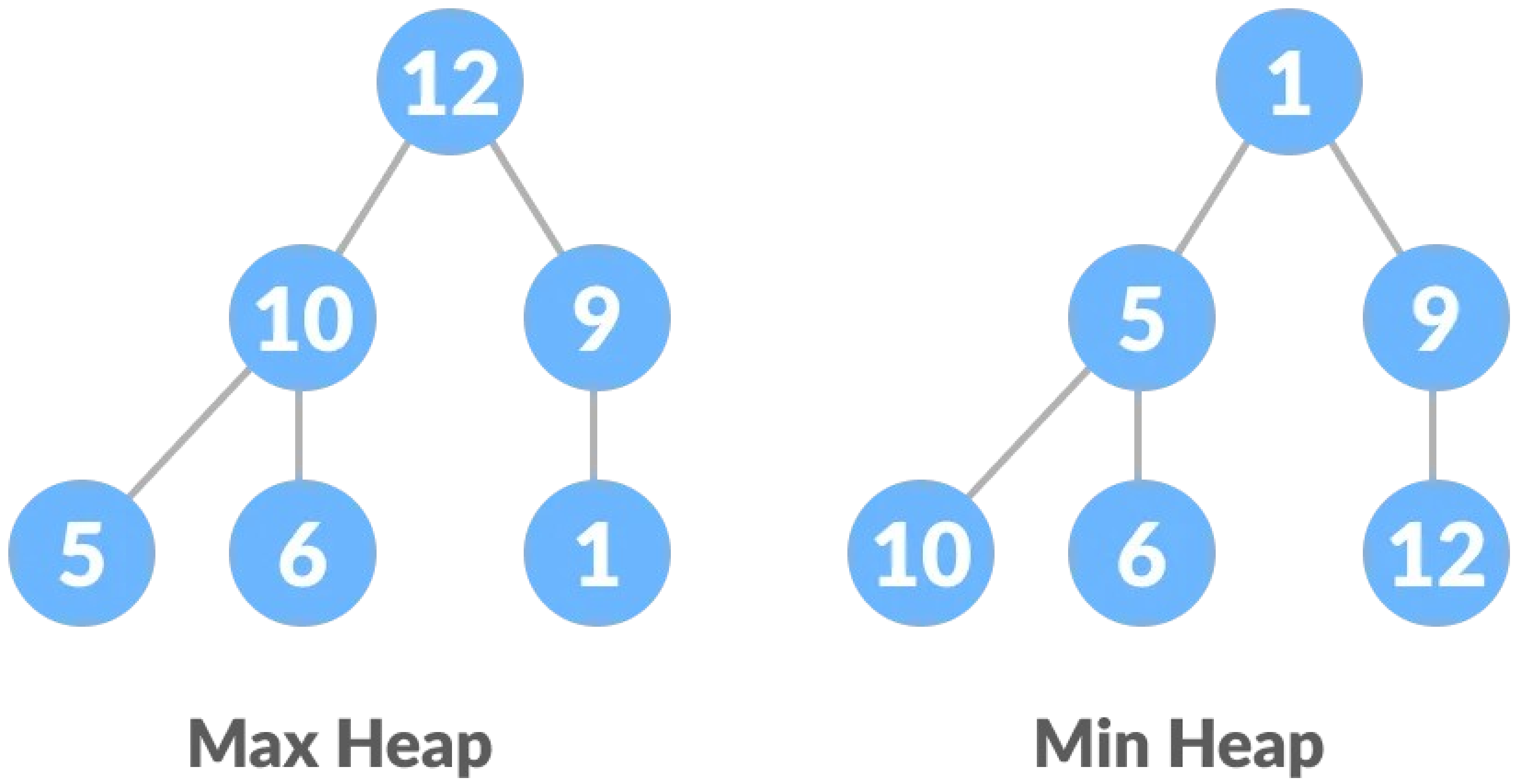
– for all nodes i, excluding the root:

$$A[\text{PARENT}(i)] \leq A[i]$$

Unit-3:Divide and Conquer Algorithms

Concept of Heap Data Structures:

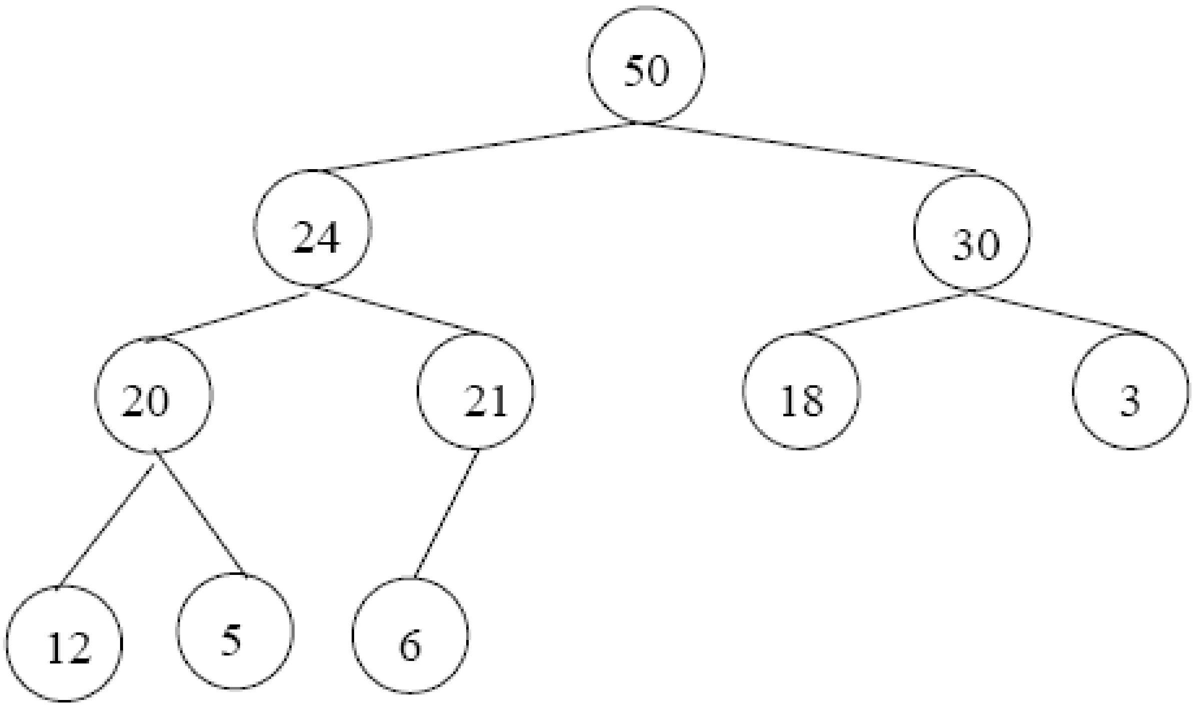
The following example diagram shows Max-Heap and Min-Heap.



Unit-3:Divide and Conquer Algorithms

Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right) and nodes are removed from the bottom level (right to left).



Unit-3:Divide and Conquer Algorithms

Operations on Heaps

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT

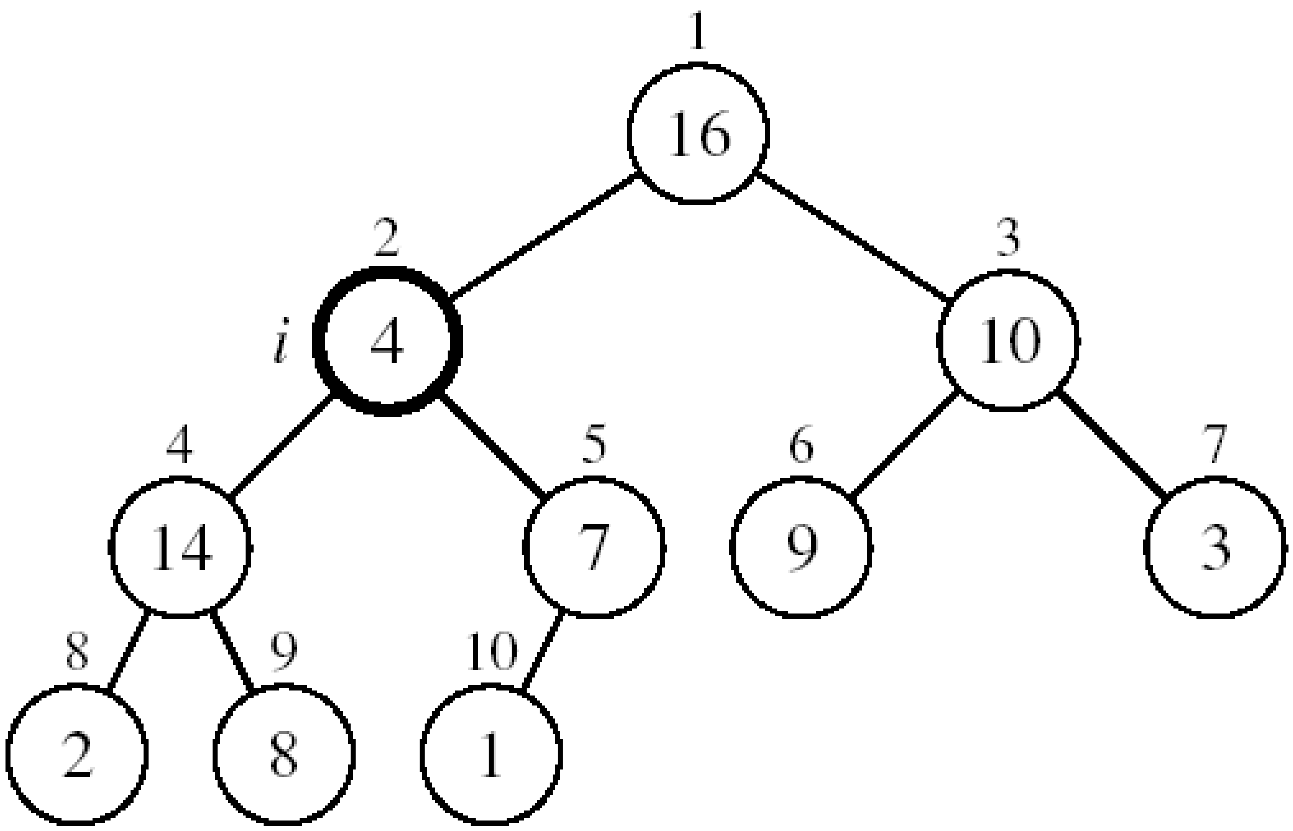
Unit-3:Divide and Conquer Algorithms

Maintaining the Heap Property

- Suppose a node is smaller than a child and Left and Right subtrees of i are max-heaps. To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children

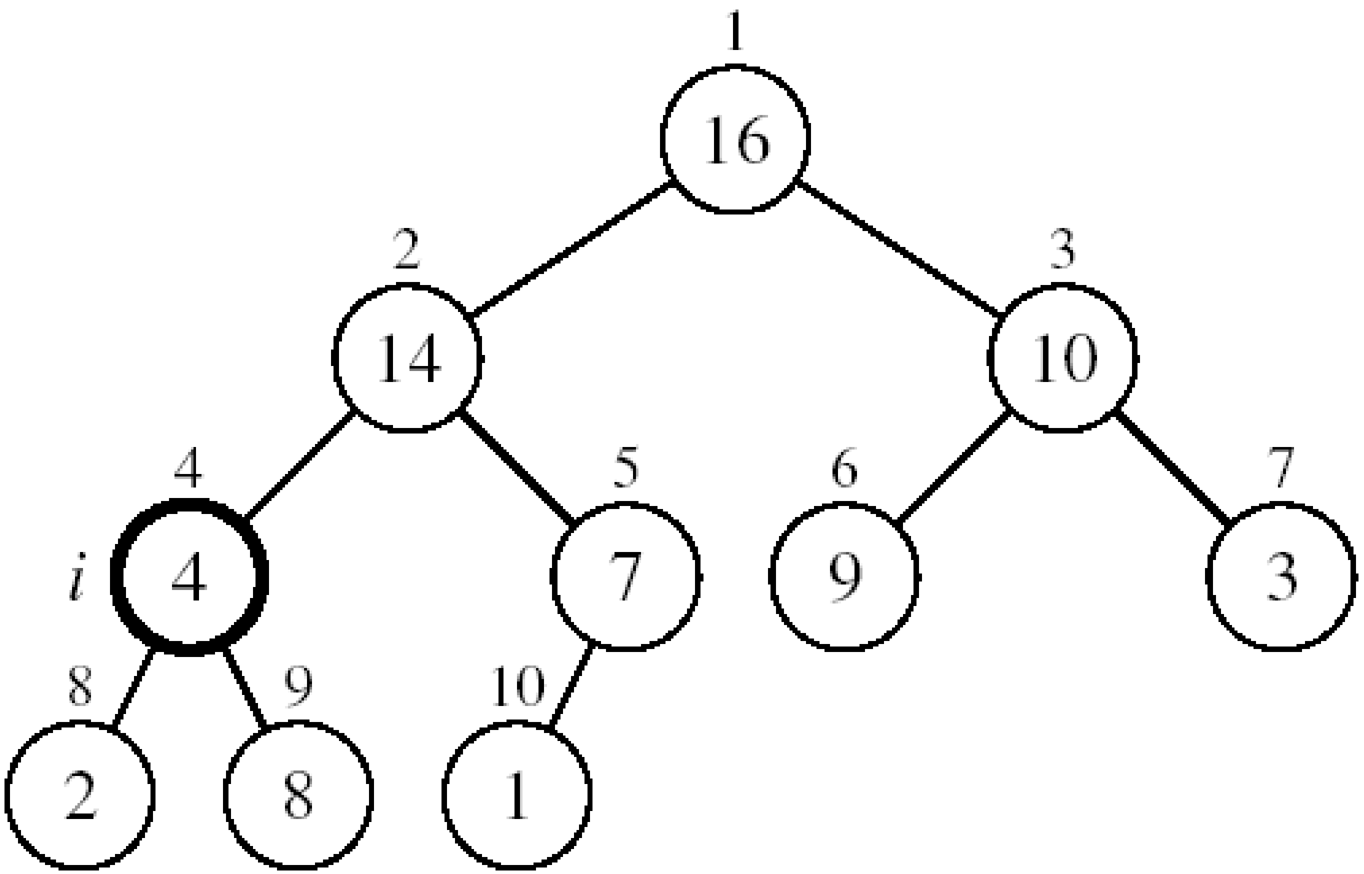
Unit-3:Divide and Conquer Algorithms

Maintaining the Heap Property



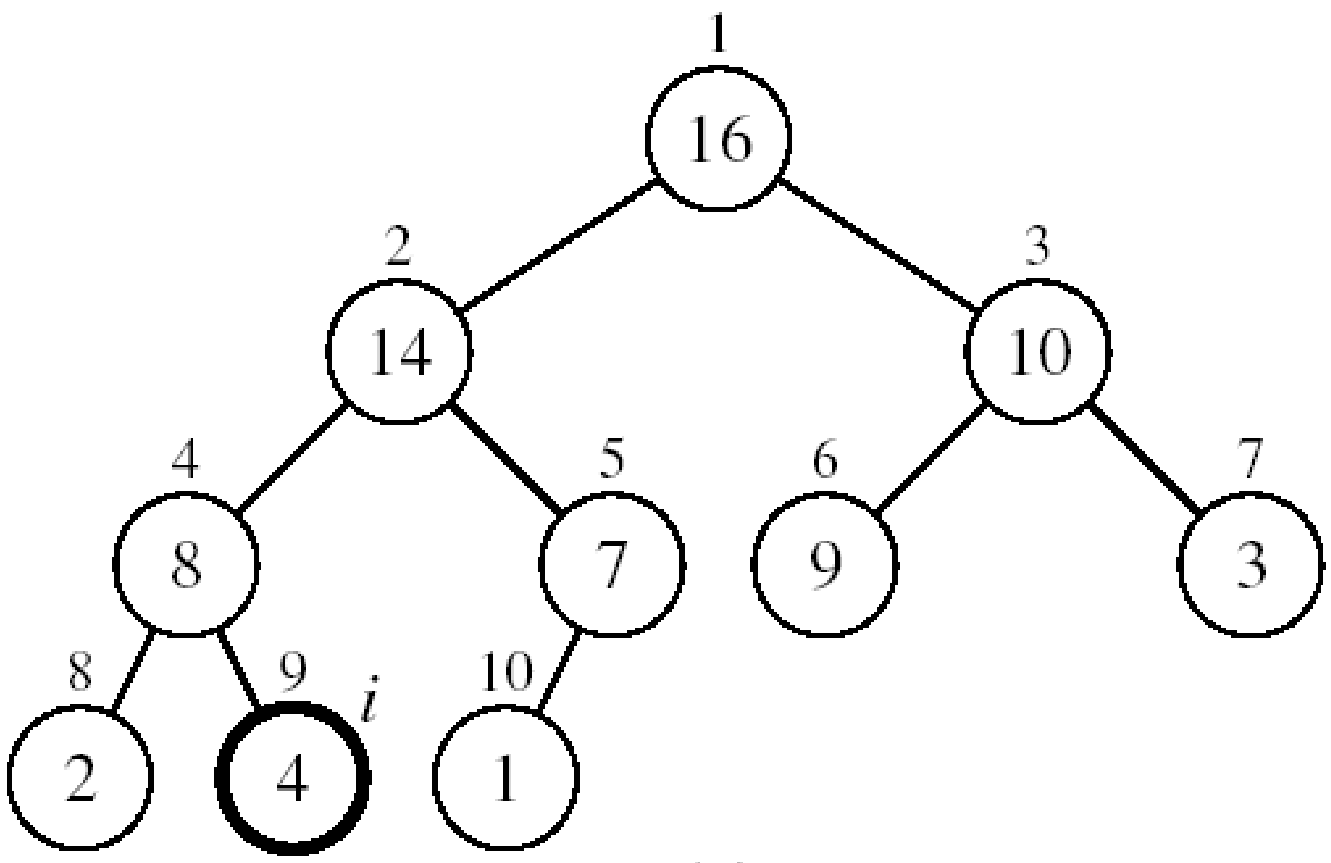
Unit-3:Divide and Conquer Algorithms

Maintaining the Heap Property



Unit-3:Divide and Conquer Algorithms

Maintaining the Heap Property



Unit-3:Divide and Conquer Algorithms

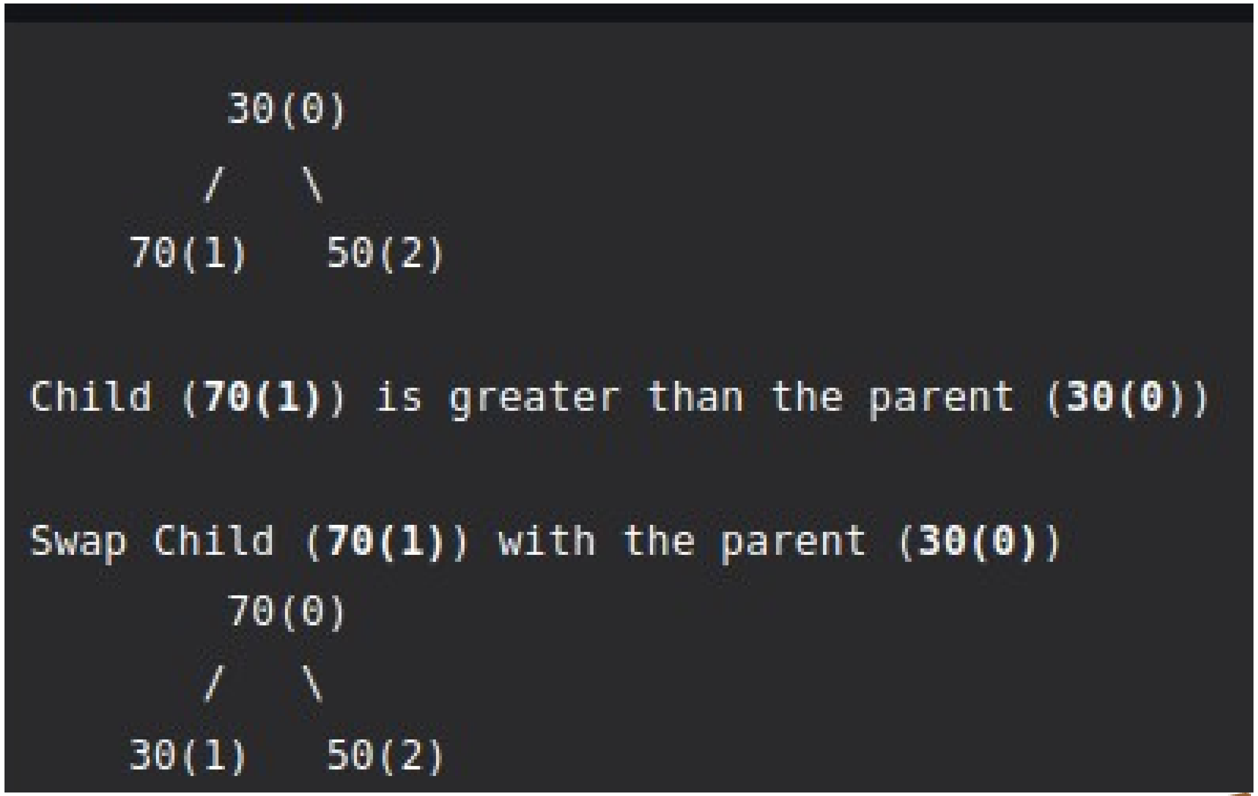
How to “heapify” a tree?

- The process of reshaping a binary tree into a Heap data structure is known as ‘heapify’.
- A binary tree is a tree data structure that has two child nodes at max. If a node’s children nodes are ‘heapified’, then only ‘heapify’ process can be applied over that node.
- A heap should always be a complete binary tree.
- Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called ‘heapify’ on all the non-leaf elements of the heap. i.e. ‘heapify’ uses recursion.

Unit-3:Divide and Conquer Algorithms

Algorithm for “heapify”:

```
heapify(array)
   Root = array[0]
   Largest = largest( array[0] , array [2 * 0 + 1]. array[2 * 0 + 2])
   if(Root != Largest)
       Swap(Root, Largest)
```



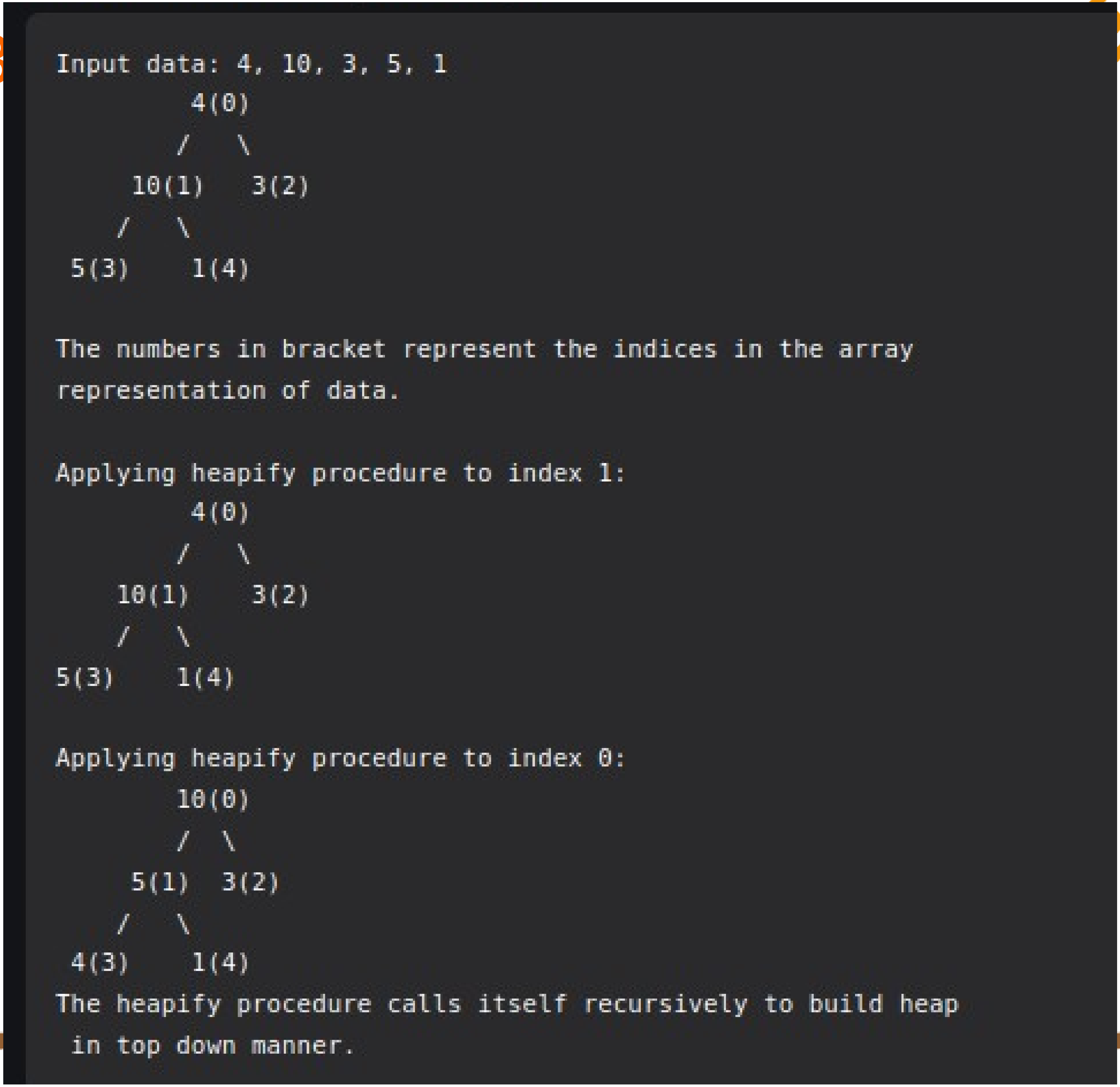
```
        30(0)
        /   \
    70(1)   50(2)

Child (70(1)) is greater than the parent (30(0))

Swap Child (70(1)) with the parent (30(0))
        70(0)
        /   \
    30(1)   50(2)
```

Unit-3

Example



```
Input data: 4, 10, 3, 5, 1
         4(0)
        /   \
    10(1)   3(2)
    /   \
 5(3)    1(4)

The numbers in bracket represent the indices in the array
representation of data.

Applying heapify procedure to index 1:
         4(0)
        /   \
    10(1)    3(2)
    /   \
5(3)    1(4)

Applying heapify procedure to index 0:
        10(0)
        /  \
     5(1)  3(2)
    /   \
 4(3)    1(4)
The heapify procedure calls itself recursively to build heap
 in top down manner.
```

Unit-3:Divide and Conquer Algorithms

Heapify: Pseudo code:

```
Max-Heapify(A, i, n)
{
    l = Left(i)
    r = Right(i)
    largest=l;
    if l ≤ n and A[l] > A[largest]
        largest = l
    if r ≤ n and A[r] > A[largest]
        largest = r
    if largest !=i
        exchange (A[i] , A[largest])
        Max-Heapify(A, largest, n)
}
```

Heapify:

- **Time Complexity Analysis:**
  - In the worst case Max-Heapify is called recursively h times, where h is height of the heap and since each call to the heapify takes constant time.
  - Time complexity = O(h) = O(logn)

Unit-3:Divide and Conquer Algorithms

Building a Heap:

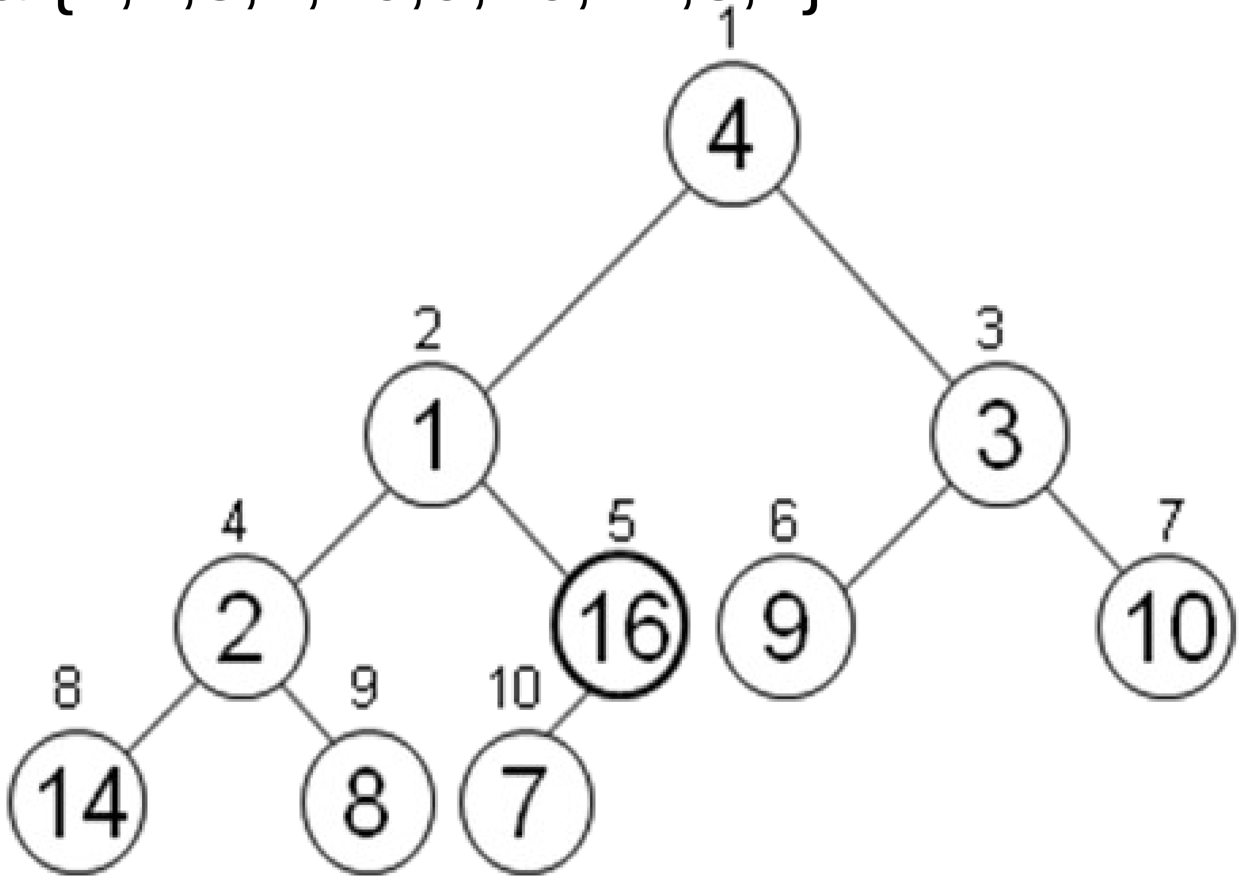
- To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied to all the elements including the root element.

- Consider the following examples
  - Given data sequence{4,1,3,2,16,9,10,14,8,7}
  - Here we need to construct binary tree first and
  - We need to carry out heapify operations on every non leaf nodes to build the Max-heap.

Unit-3:Divide and Conquer Algorithms

Building a Heap:

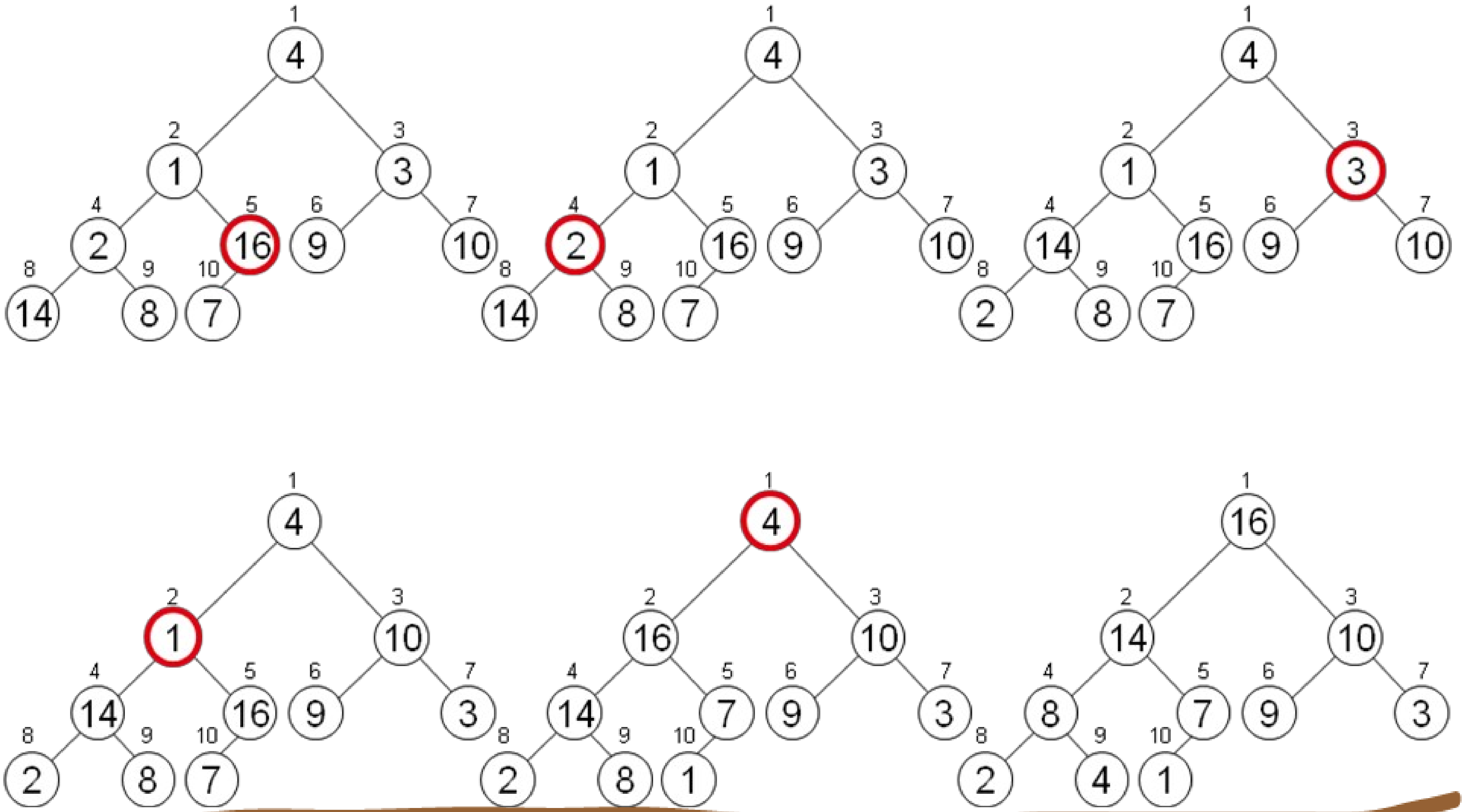
– Here we need to construct binary tree first for the given data {4,1,3,2,16,9,10,14,8,7}



Unit-3:Divide and Conquer Algorithms

Building a Heap:

– Then We need to carry out heapify operations to build the Max-heap.



Unit-3:Divide and Conquer Algorithms

Building a Heap: Pseudo Code

```
Build-Max-Heap(A)
    n = length[A]
    for i ← $\lfloor n/2 \rfloor$ downto 1
    {
        MAX-HEAPIFY(A, i, n)
    }
```

- **Time Complexity:**
- Running time: Loop executes O(n) times and complexity of Heapify is O(logn), therefore complexity of Build-Max-Heap is O(nlogn).

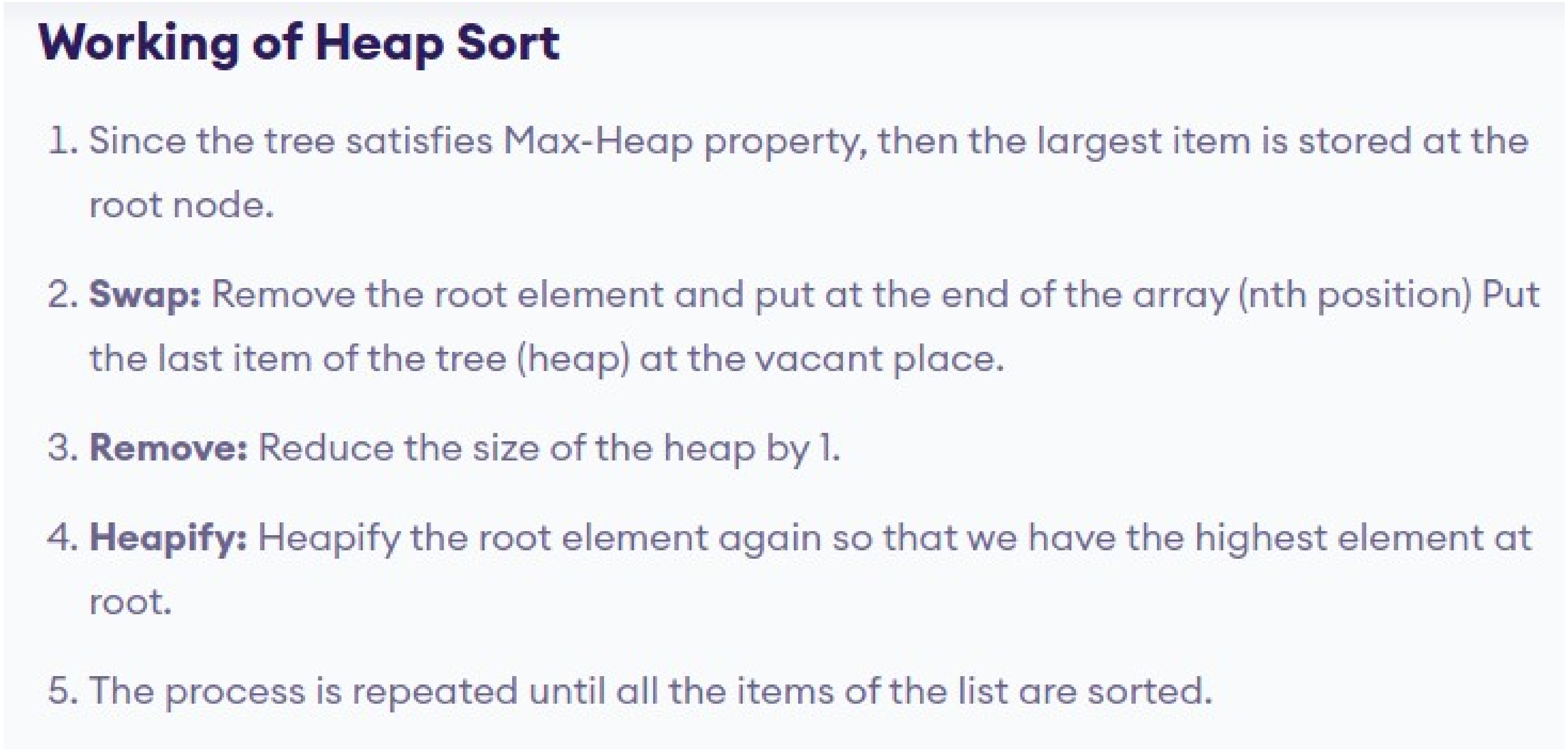
Unit-3:Divide and Conquer Algorithms

Heap Sort:

- Heap sort is a comparison-based sorting technique based on Binary Heap data structure.
- It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning.
- We repeat the same process for the remaining elements.
- Steps involved to sort n elements:
  - Build a max-heap from the array
  - Swap the root (the maximum element) with the last element in the array
  - “Discard” this last node by decreasing the heap size
  - Call Max-Heapify on the new root
  - Repeat this process until only one node remains

Unit-3:Divide and Conquer Algorithms

Heap Sort:

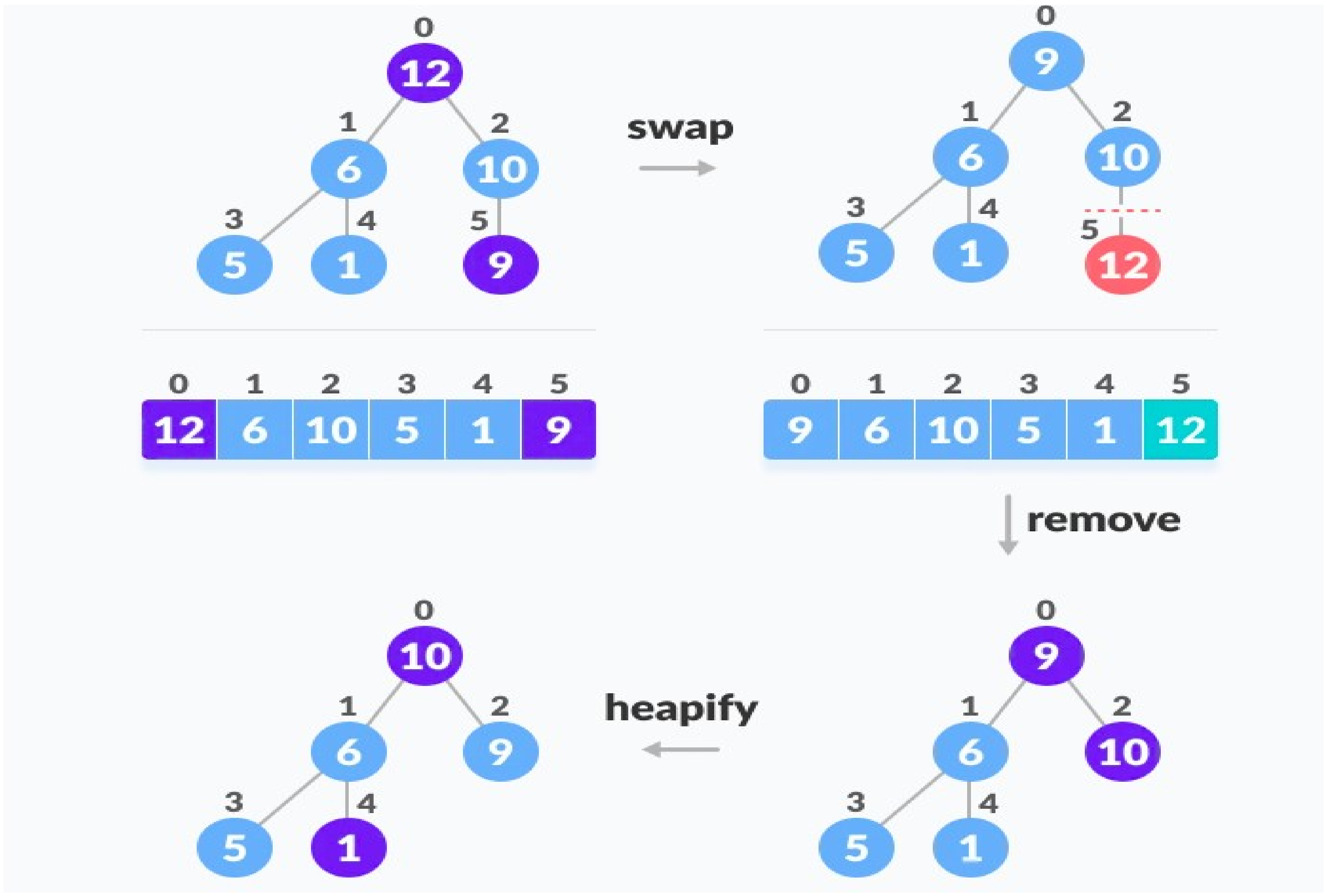


Working of Heap Sort

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.

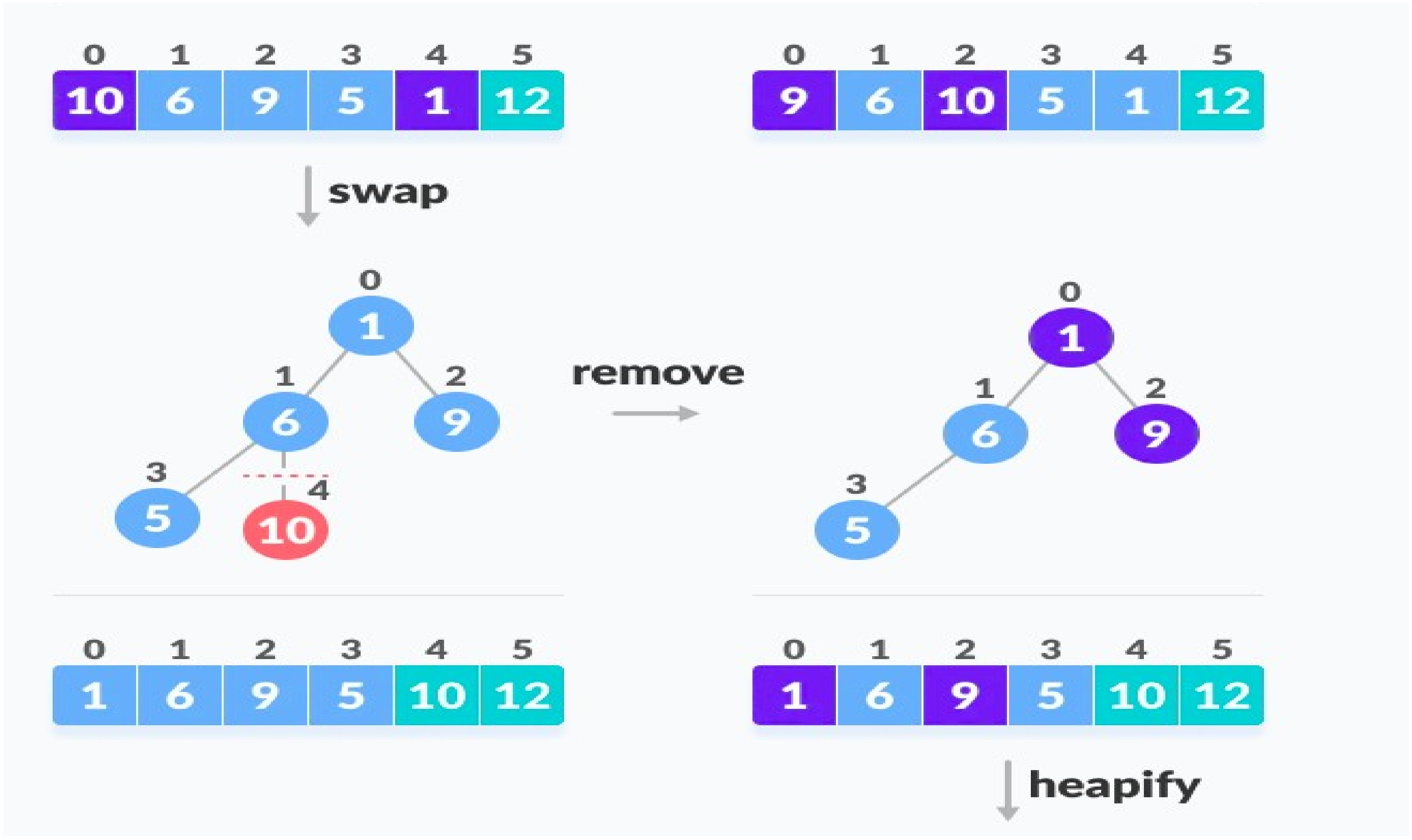
Unit-3:Divide and Conquer Algorithms

Heap Sort: Example



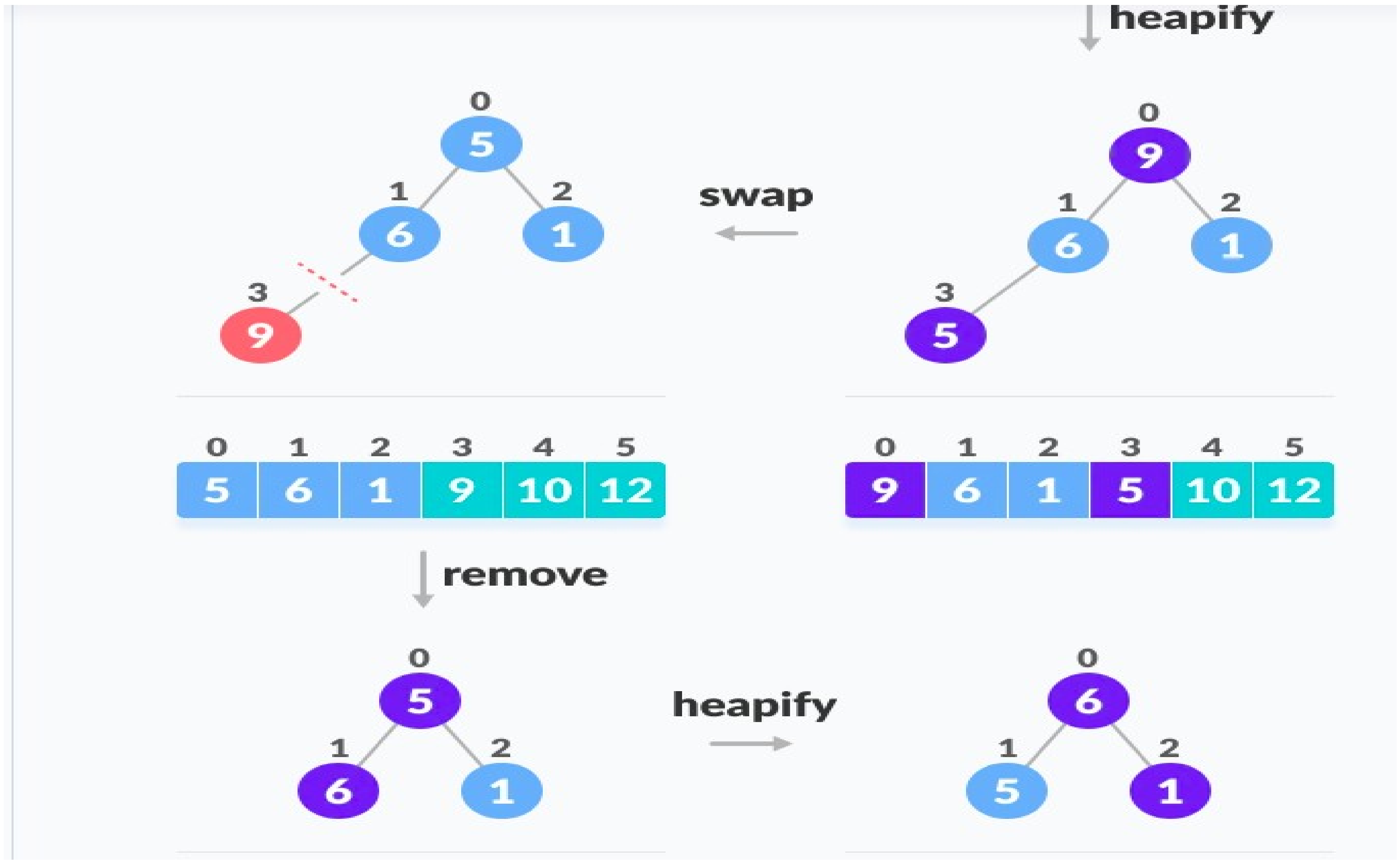
Unit-3:Divide and Conquer Algorithms

Heap Sort: Example



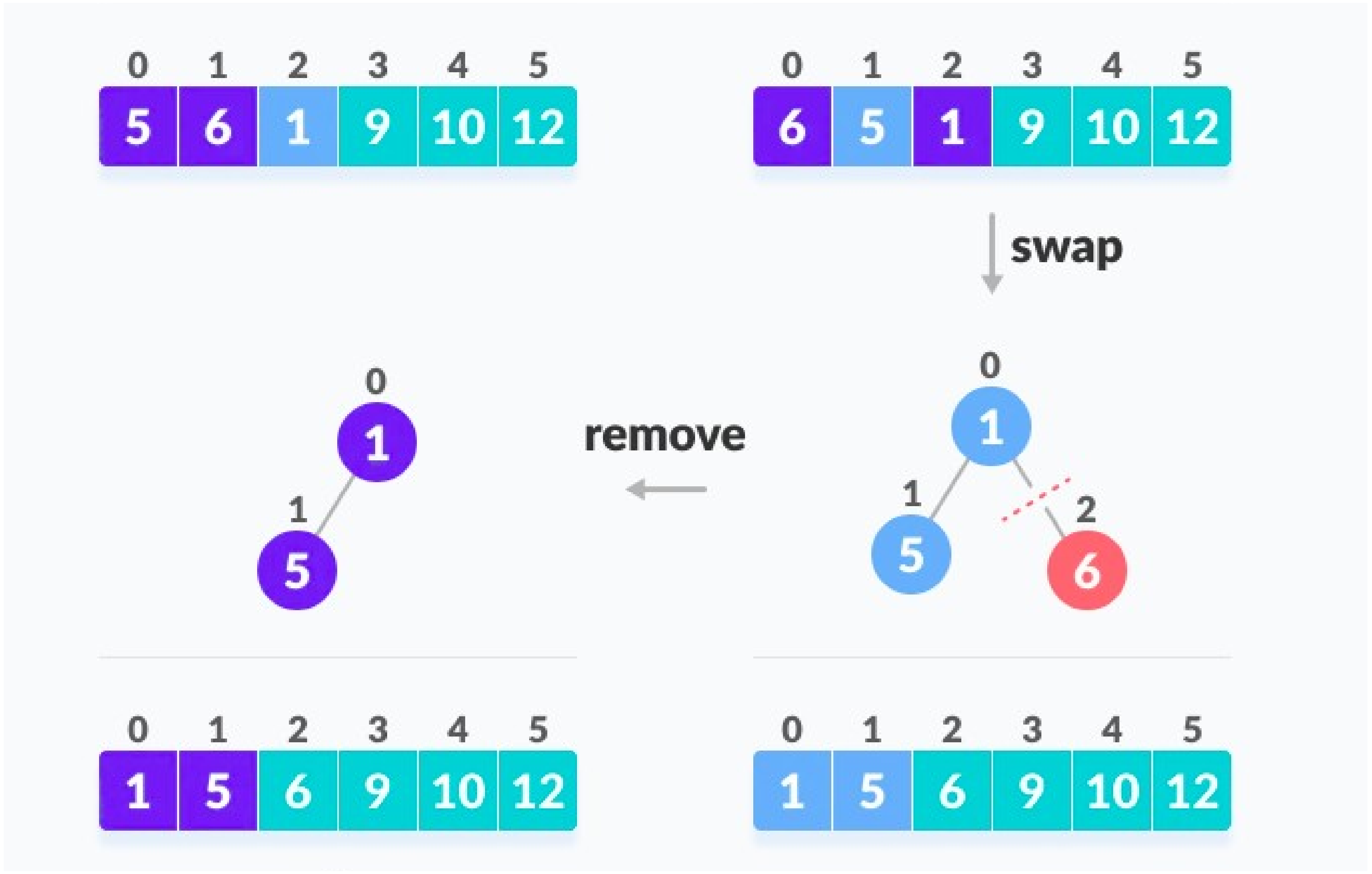
Unit-3:Divide and Conquer Algorithms

Heap Sort: Example



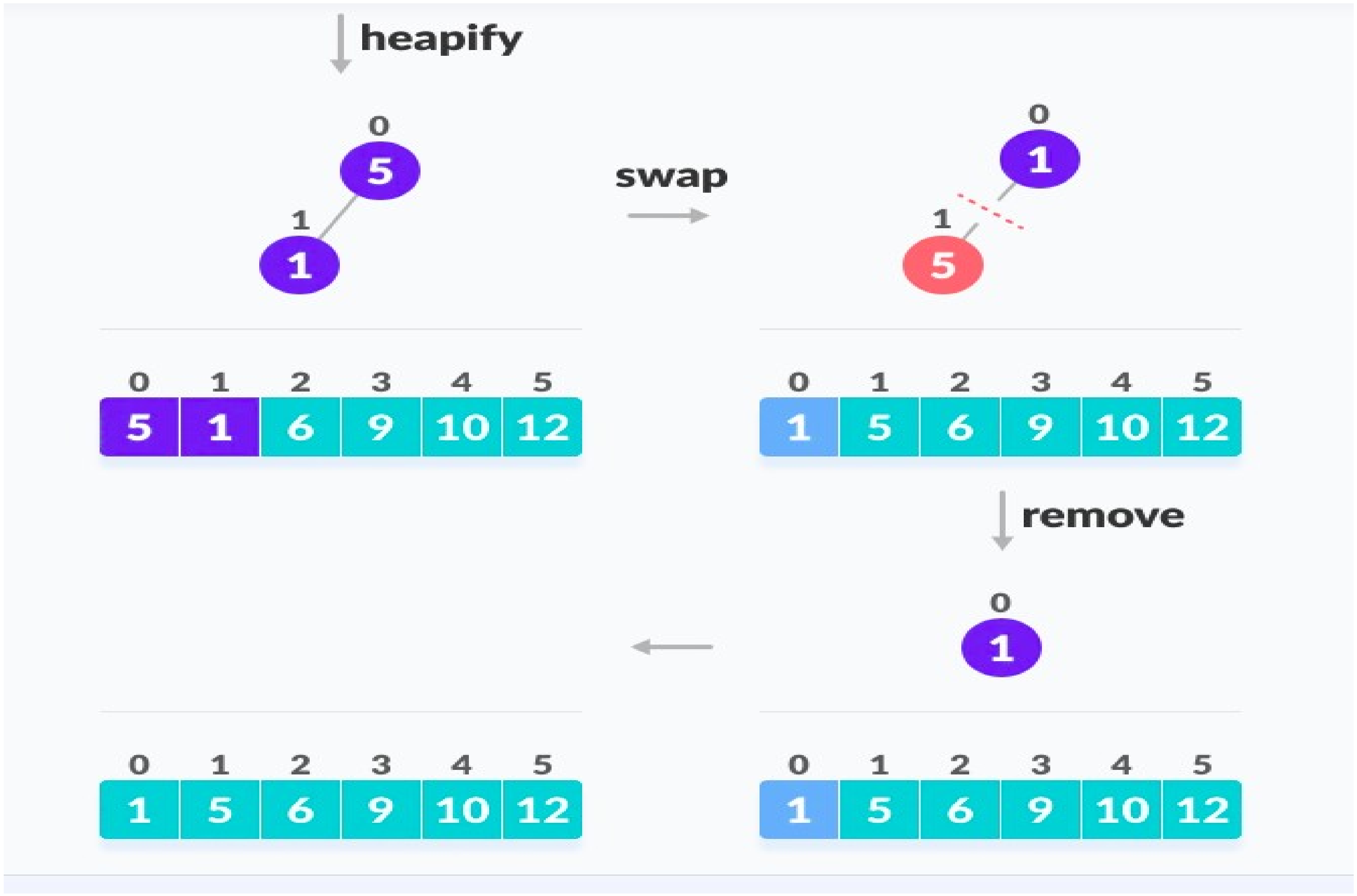
Unit-3:Divide and Conquer Algorithms

Heap Sort: Example



Unit-3:Divide and Conquer Algorithms

Heap Sort: Example



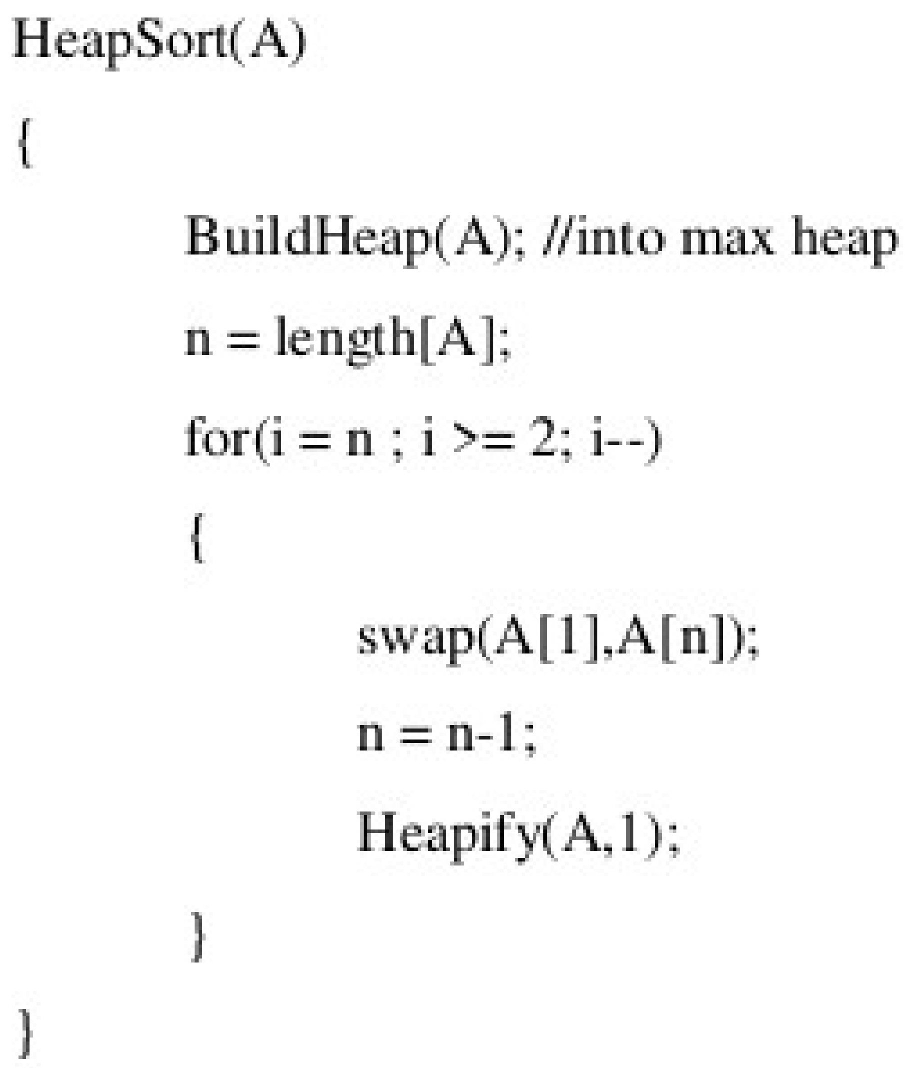
Unit-3:Divide and Conquer Algorithms

Heap Sort: Examples (do at your own)

- Sort the following elements using heap sort
  - [1,4,2,7,3]
- First construct the binary tree and construct the heap of recently constructed binary tree.
- Then apply heap sort.

Unit-3:Divide and Conquer Algorithms

Heap Sort: Pseudo Code



```
HeapSort(A)
{
        BuildHeap(A); //into max heap
        n = length[A];
        for(i = n ; i >= 2; i--)
        {
                swap(A[1],A[n]);
                n = n-1;
                Heapify(A,1);
        }
}
```

- Analysis:
  - Building heap takes O(n)
  - Loop executes n times
  - Heapify operations takes O(logn)
  - So total T(n)=O(nlogn)

Unit-3:Divide and Conquer Algorithms

Order statistics

- Order statistics are sample values placed in ascending order. The study of order statistics deals with the applications of these ordered values and their functions.
- Let’s say you had three weights:
  - X1 = 22 kg, X2 = 44 kg, and X3 = 12 kg.
- To get the order statistics (Yn), put the items in numerical increasing order:
  - Y1 = 12 kg
  - Y2 = 22 kg
  - Y3 = 44 kg
- The kth smallest X value is normally called the kth order statistic.
- More formally,
  - If X1, X2,…, Xn are random iid observations taken from a population with n continuous observations, then
  - the random variables Y1 < Y2 < …, < Yn denote the sample’s order statistics.

Unit-3:Divide and Conquer Algorithms

Order statistics

- The ith order statistic of a set of n elements is the ith smallest element. For example, the minimum of a set of elements is the first order statistic (i = 1), and the maximum is the nth order statistic (i = n).

- **Median order**
- A median, informally, is the "halfway point" of the set.
- When n is odd, the median is unique, occurring at i = (n + 1)/2. When n is even, there are two medians, occurring at i = n/2 and i = n/2 + 1.
- Thus, regardless of the parity of n, medians occur at i = lower bound(n + 1)/2 and i = upper bound(n + 1)/2.

Unit-3:Divide and Conquer Algorithms

Brute Force Algorithm:

- This is the most basic and simplest type of algorithm.
- A Brute Force Algorithm is the straightforward approach to a problem i.e., the first approach that comes to our mind on seeing the problem.
- More technically it is just like iterating every possibility available to solve that problem.
- For Example:
  - If there is a lock of 4-digit PIN.
  - The digits to be chosen from 0-9 then the brute force will be trying all possible combinations one by one like 0001, 0002, 0003, 0004, and so on until we get the right PIN.
  - In the worst case, it will take 10,000 tries to find the right combination.

Unit-3:Divide and Conquer Algorithms

Brute Force Algorithm:

- Below given are some features of the brute force algorithm are:
  - It is an intuitive, direct, and straightforward technique of problem-solving in which all the possible ways or all the possible solutions to a given problem are enumerated.
  - Many problems solved in day-to-day life using the brute force strategy, for example exploring all the paths to a nearby market to find the minimum shortest path.
  - Arranging the books in a rack using all the possibilities to optimize the rack spaces, etc.
  - In fact, daily life activities use a brute force nature, even though optimal algorithms are also possible.

Unit-3:Divide and Conquer Algorithms

Brute Force Algorithm:

- A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved.
- Examples:

1. Computing an (a > 0, n a non negative integer)
2. Computing n!
3. Multiplying two matrices
4. Searching for a key of a given value in a list

Brute-Force Sorting Algorithm

- Selection Sort
  - Scan the array to find its smallest element and swap it with the first element.
  - Starting with the second element, scan the elements after it to find the smallest among them and swap it with the second elements.
  - Generally, on pass i ($0 \le i \le n-2$), find the smallest element after A[i] and swap it with A[i]:

    $A[0] \le \ . \ . \ . \le A[i-1] \mid A[i], \ . \ . \ . \ , A[min], \ . \ . \ ., A[n-1]$

    in their final positions
- Example: 7 3 2 5

Unit-3:Divide and Conquer Algorithms

Brute Force Algorithm:

- Brute force algorithm is a technique that guarantees solutions for problems of any domain helps in solving the simpler problems and also provides a solution that can serve as a benchmark for evaluating other design techniques, but takes a lot of run time and inefficient.

Unit-3:Divide and Conquer Algorithms

Selection in Expected Linear Time

- The general selection problem appears more difficult than the simple problem of finding a minimum, yet, surprisingly, the asymptotic running time for both problems is the same: (n).
- Here this approach is a divide-and-conquer algorithm for the selection problem.
- The algorithm RANDOMIZED-SELECT is modeled after the quicksort algorithm. As in quicksort, the idea is to partition the input array recursively.
- But unlike quicksort, which recursively processes both sides of the partition, RANDOMIZED-SELECT only works on one side of the partition.
- This difference shows up in the analysis: whereas quicksort has an expected running time of (nlogn), the expected time of RANDOMIZED-SELECT is (n^2).

Unit-3:Divide and Conquer Algorithms

Selection in Expected Linear Time

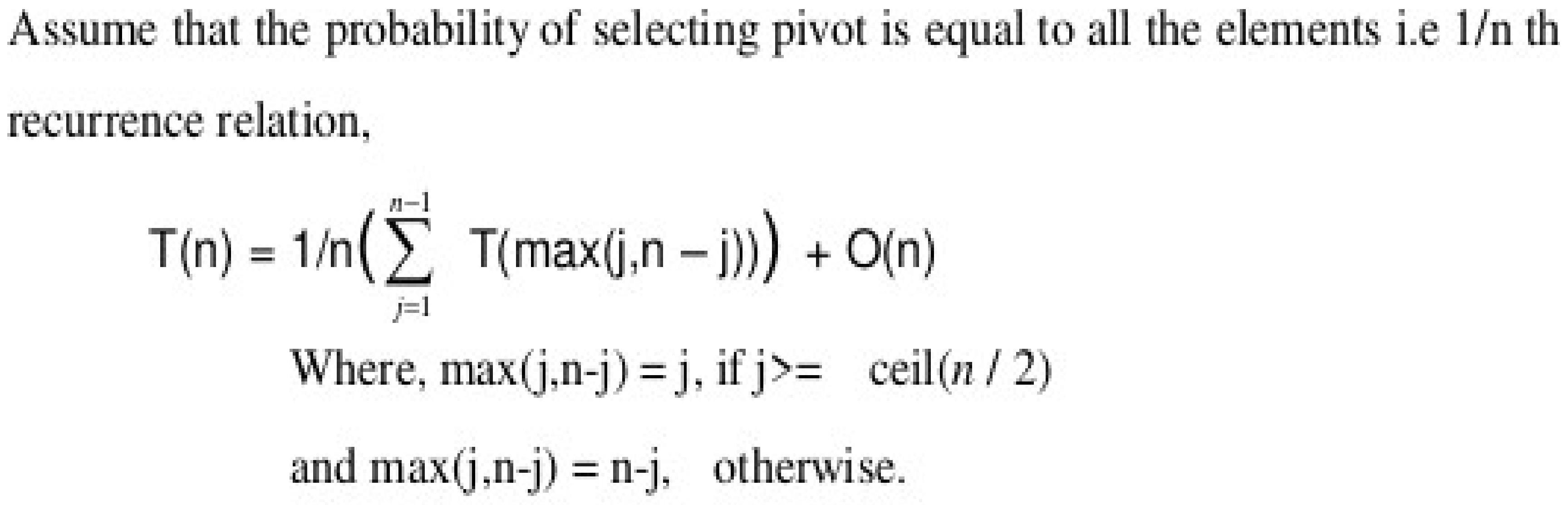
- This problem is solved by using the “divide and conquer” method. The main idea for this problem solving is to partition the element set as in Quick Sort where partition is randomized one.
- Pseudo Code:

```
RandSelect(A,l,r,i)
  {
    if(l = =r )
        return A[p];
    p = RandPartition(A,l,r);
    k = p – l + 1;
    if(i <= k)
        return RandSelect(A,l,p-1,i);
    else
        return RandSelect(A,p+1,r,i - k);
    }
```

Unit-3:Divide and Conquer Algorithms

Analysis:

- Since our algorithm is randomized algorithm no particular input is responsible for worst case however the worst case running time of this algorithm is O(n 2).
- This happens if every time unfortunately the pivot chosen is always the largest one (if we are finding minimum element).



Assume that the probability of selecting pivot is equal to all the elements i.e 1/n th

recurrence relation,

$$T(n) = 1/n\left(\sum_{j=1}^{n-1} T(\max(j, n-j))\right) + O(n)$$

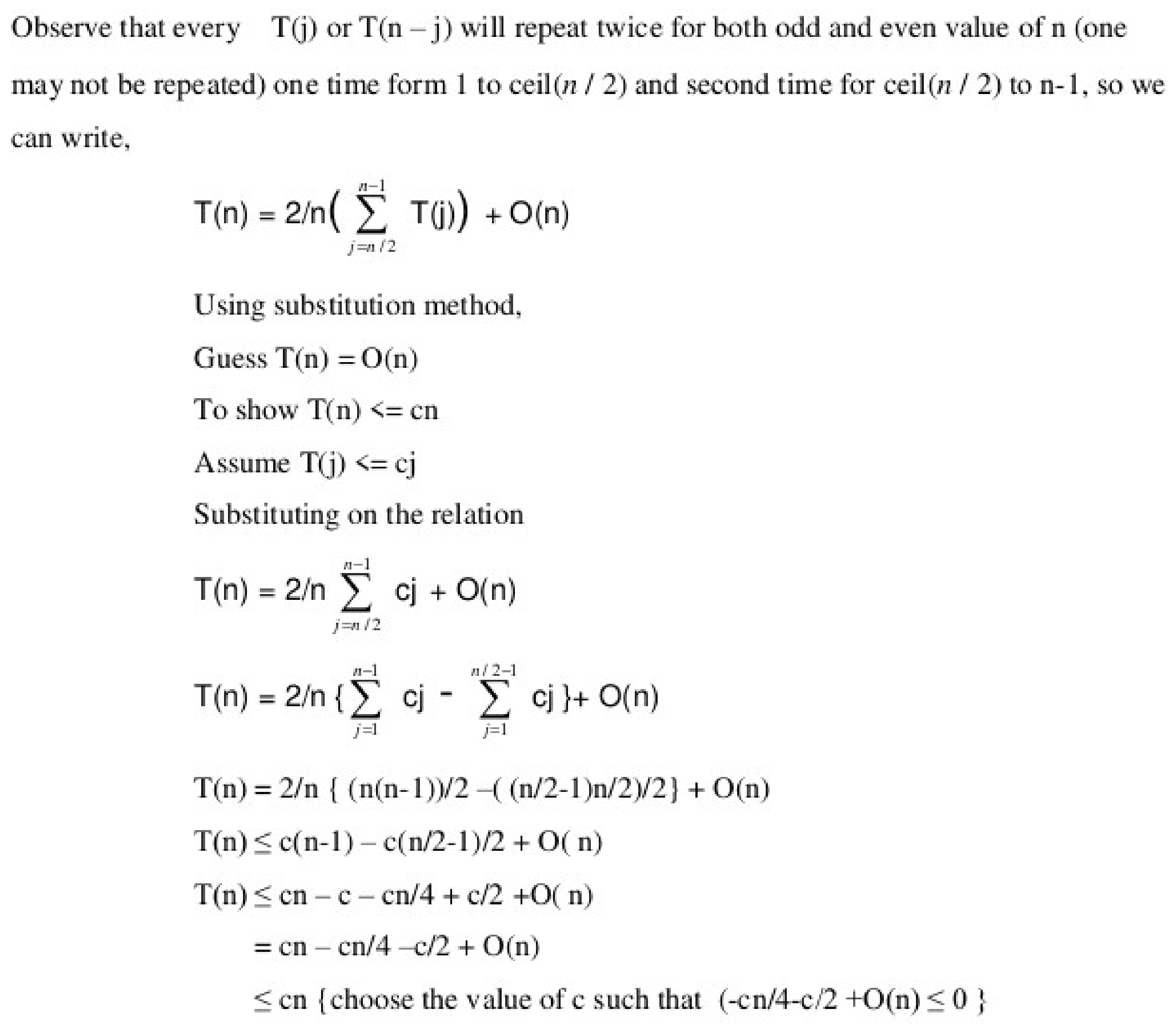
Where, max(j,n-j) = j, if j>= ceil(n / 2)

and max(j,n-j) = n-j, otherwise.

Unit-3:Divide and Conquer Algorithms

Analysis:

- T(n)=O(n)



Observe that every T(j) or T(n – j) will repeat twice for both odd and even value of n (one may not be repeated) one time form 1 to ceil(n / 2) and second time for ceil(n / 2) to n-1, so we can write,

$$T(n) = 2/n\left(\sum_{j=n/2}^{n-1} T(j)\right) + O(n)$$

Using substitution method,

Guess T(n) = O(n)

To show T(n) <= cn

Assume T(j) <= cj

Substituting on the relation

$$T(n) = 2/n \sum_{j=n/2}^{n-1} cj + O(n)$$

$$T(n) = 2/n \{\sum_{j=1}^{n-1} cj - \sum_{j=1}^{n/2-1} cj \} + O(n)$$

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

$$T(n) \le c(n-1) - c(n/2-1)/2 + O(n)$$

$$T(n) \le cn - c - cn/4 + c/2 +O(n)$$

$$= cn - cn/4 -c/2 + O(n)$$

$$\le cn \text{ \{choose the value of c such that } (-cn/4-c/2 +O(n) \le 0 \text{ \}}$$

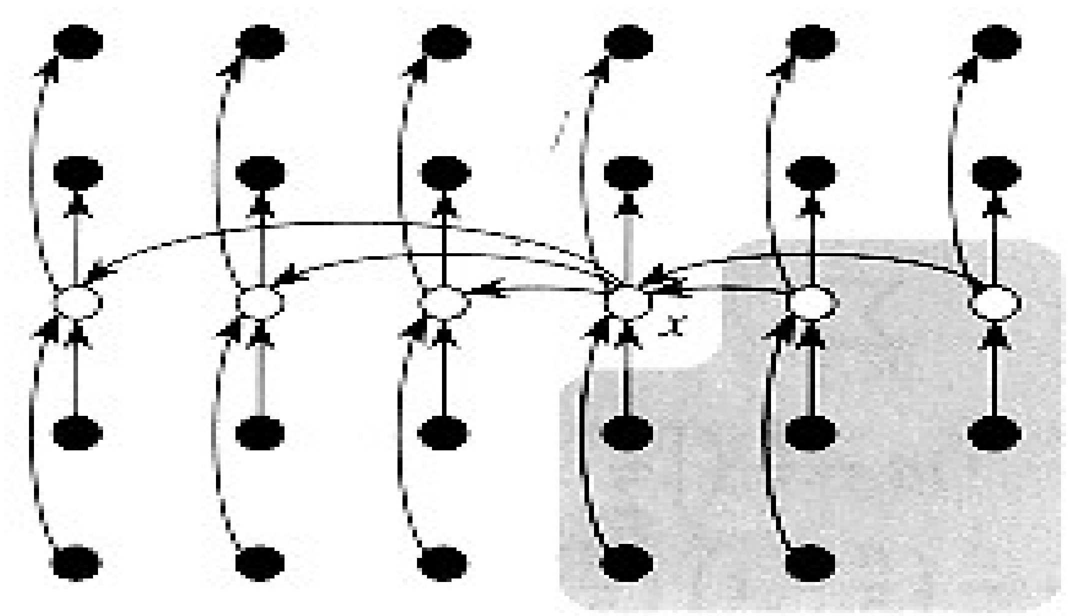
Unit-3:Divide and Conquer Algorithms

Selection in Worst Case Linear Time algorithm

- We now examine a selection algorithm whose running time is O(n) in the worst case. Like RANDOMIZED-SELECT, the algorithm SELECT finds the desired element by recursively partitioning the input array.
- The idea behind the algorithm, however, is to guarantee a good split when the array is partitioned.
- SELECT uses the deterministic partitioning algorithm PARTITION from quicksort, modified to take the element to partition around as an input parameter.
-

Unit-3:Divide and Conquer Algorithms

Selection in Worst Case Linear Time algorithm

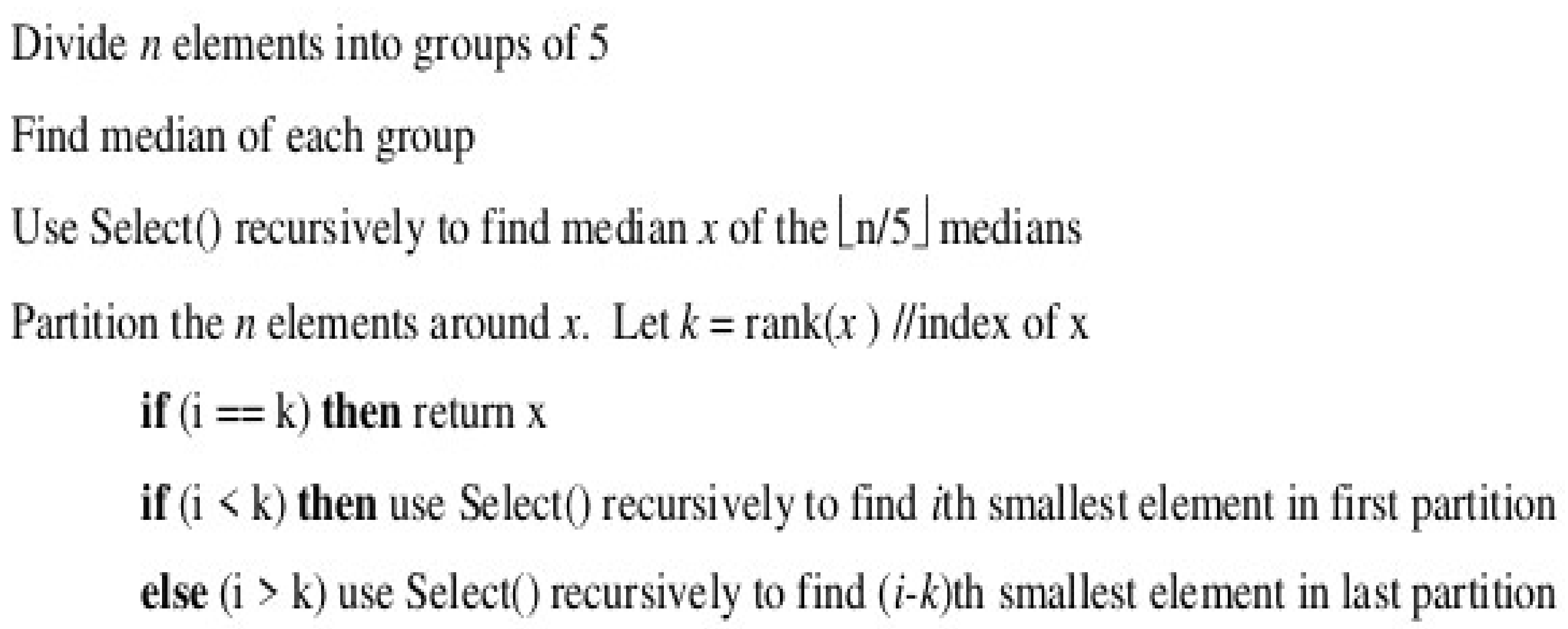


- Here, The n elements are represented by small circles, and each group occupies a column.
- The medians of the groups are whitened, and the median-of-medians x is labeled. Arrows are drawn from larger elements to smaller, from which it can be seen that 3 out of every group of 5 elements to the right of x are greater than x, and 3 out of every group of 5 elements to the left of x are less than x.
- The elements greater than x are shown on a shaded background.

Unit-3:Divide and Conquer Algorithms

Selection in Worst Case Linear Time algorithm

- **Algorithms:**

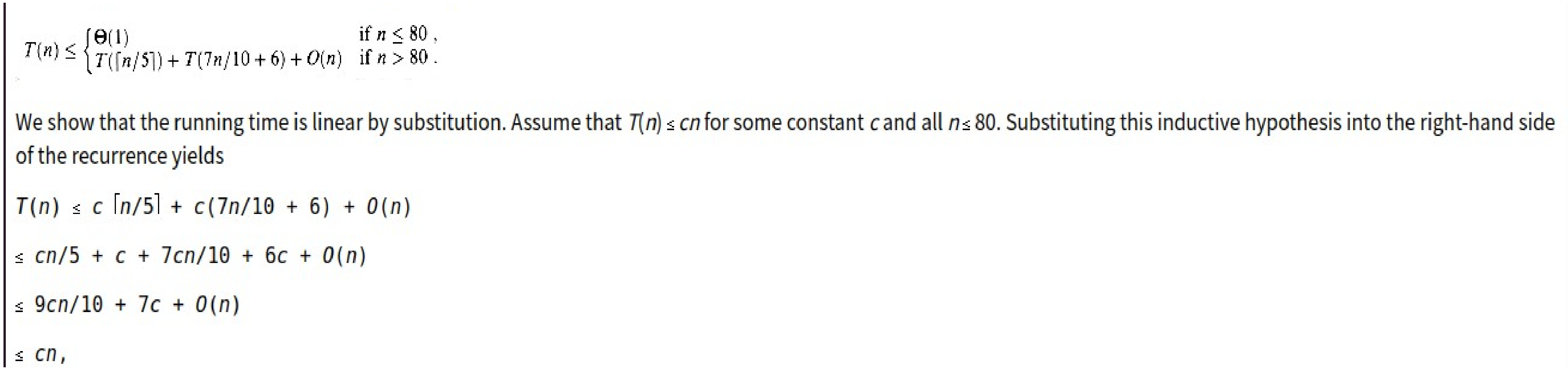


```
Divide $n$ elements into groups of 5
Find median of each group
Use Select() recursively to find median $x$ of the $\lfloor n/5 \rfloor$ medians
Partition the $n$ elements around $x$.  Let $k$ = rank($x$ ) //index of x
          if (i == k) then return x
          if (i < k) then use Select() recursively to find $i$th smallest element in first partition
          else (i > k) use Select() recursively to find ($i$-$k$)th smallest element in last partition
```

Unit-3:Divide and Conquer Algorithms

Selection in Worst Case Linear Time algorithm

- Analysis:
  - Here at least half of the medians are <=x, since there are at least n/5 medians
  - So (n/5)/2 medians are <=x, i.e. n/10 medians are<=x
  - Since each medians contributes 3 elements which are<=x i.e. 3n/10 elements are <=x
  - So out of n elements 7n/10 elements are>=x
  - Now recurrence relation is:



$$T(n) \le \begin{cases} \Theta(1) & \text{if } n \le 80\,, \\ T(\lceil n/5 \rceil) + T(7n/10+6) + O(n) & \text{if } n > 80\,. \end{cases}$$

We show that the running time is linear by substitution. Assume that $T(n) \le cn$ for some constant c and all $n \le 80$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$T(n) \le c\lceil n/5 \rceil + c(7n/10 + 6) + O(n)$

$\le cn/5 + c + 7cn/10 + 6c + O(n)$

$\le 9cn/10 + 7c + O(n)$

$\le cn,$

Unit-3:Divide and Conquer Algorithms

Assignment:

- Discuss some implementation issues that may arise in divide and conquer algorithms.
- Trace the algorithms for the following array of elements elements by using recursive approach of Min-Max algorithms.

  [6,5,3,8,11,2,99,35,7]
- Trace for key =7 in [-1,5,6,7,18,20,25,27,39,91,119,121] using binary search.
- Introduce the concepts of partitioning and analyze the best, average and worst case time complexity of quick sort algorithm based on divide and conquer approach. Sort the following data using quick sort: [5,3,2,6,4,1,3,7]

Unit-3:Divide and Conquer Algorithms

Assignment:

- Sort the following elements using heap sort:

{4,1,3,2,16,9,10,14,8,7}