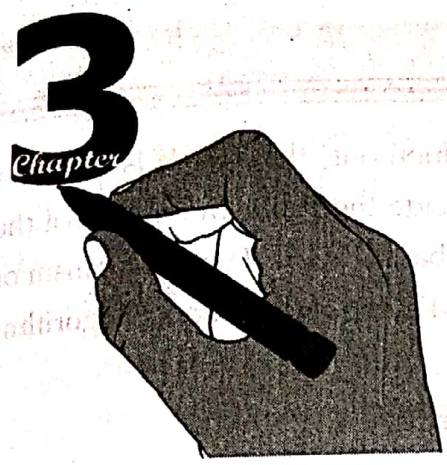


DIVIDE AND CONQUER ALGORITHMS

CHAPTER OUTLINE

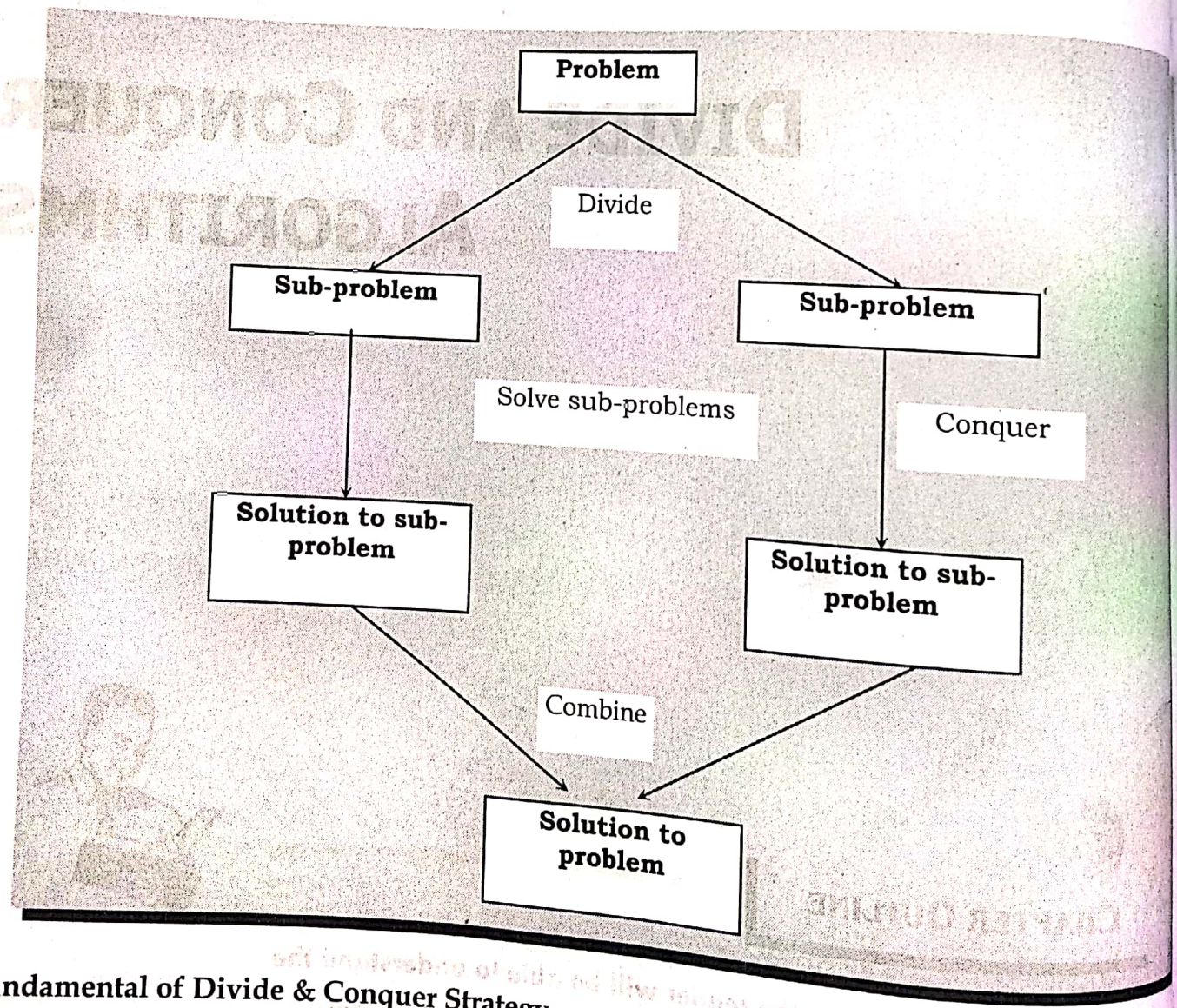
After studying this chapter, the reader will be able to understand the

- Searching Algorithms: Binary Search, Min-Max Finding and their Analysis
- Sorting Algorithms: Merge Sort and Analysis, Quick Sort and Analysis (Best Case, Worst Case and Average Case), Heap Sort (Heapify, Build Heap and Heap Sort Algorithms and their Analysis), Randomized Quick sort and its Analysis
- Order Statistics: Selection in Expected Linear Time, Selection in Worst Case Linear Time and their Analysis.

Introduction

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism solving the problem is called the Divide & Conquer Strategy. Divide and Conquer algorithm consists of a dispute using the following three steps.

- **Divide** the original problem into a set of sub-problems.
- **Conquer:** Solve every sub-problem individually, recursively.
- **Combine:** Put together the solutions of the sub-problems to get the solution to the whole problem.



Fundamental of Divide & Conquer Strategy

There are two fundamental of Divide & Conquer Strategy:

- Relational Formula
- Stopping Condition

1. **Relational Formula**

   It is the formula that we generate from the given technique. After generation of Formula we apply divide and conquer strategy, i.e. we break the problem recursively & solve the broken sub-problems.

2. **Stopping Condition**

   When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of divide and conquer is called as Stopping Condition

Searching Algorithms

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:

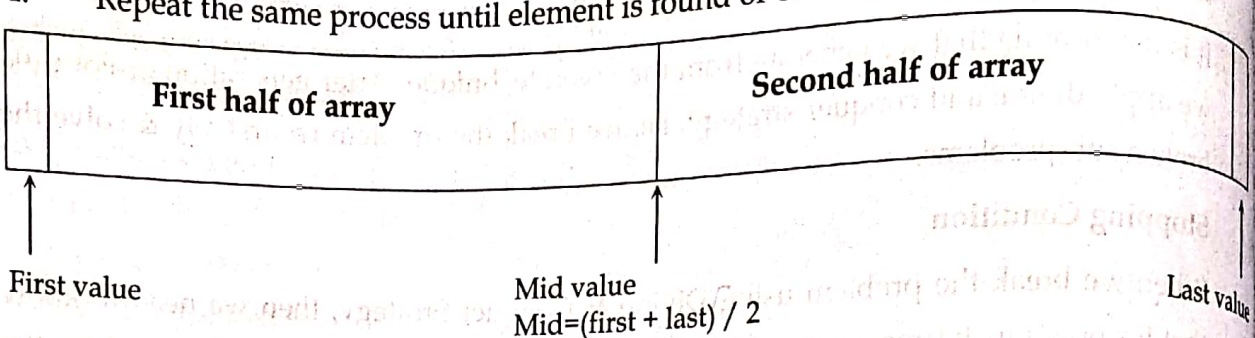
- Linear Search or Sequential Search
- Binary Search

Binary Search

Binary search algorithm finds given element in a list of elements with **O(log n)** time complexity where **n** is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of element which are already arranged in an order. The binary search cannot be used for list of element which is in random order. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sub list of the middle element. If the search element is larger, then we repeat the same process for right sub list of the middle element. We repeat this process until we find the search element in the list or until we left with a sub list of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list". The logic behind this technique is given below:

1. First find the middle element of the array
2. Compare the middle element with an item.
3. There are three cases:
   a. If it is a desired element then search is successful
   b. If it is less than desired item then search only the first half of the array.
   c. If it is greater than the desired element, search in the second half of the array.

4. Repeat the same process until element is found or exhausts in the search area.



Tracing: Search for key= 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114} by using Binary search.

Solution:

Initially

-1	5	6	18	19	25	46	78	102	114
0	1	2	3	4	5	6	7	8	9

Step 1: Since [middle element (19) > key (6)]

-1	5	6	18	19	25	46	78	102	114
0	1	2	3	4	5	6	7	8	9

Step 2: Since [middle element (5) < key (6)]

~~-1~~	~~5~~	6	18	~~19~~	~~25~~	46	~~78~~	~~102~~	~~114~~
~~0~~	~~1~~	2	3	4	~~5~~	~~6~~	~~7~~	~~8~~	~~9~~

Step 1: Since [middle element (6) > key (6)]

~~-1~~	~~5~~	6	~~18~~	~~19~~	~~25~~	~~46~~	~~78~~	~~102~~	~~114~~
~~0~~	~~1~~	2	~~3~~	4	~~5~~	~~6~~	~~7~~	~~8~~	~~9~~

Algorithm

1. Start
2. Read the search element from the user
3. Find the middle element in the sorted list
4. Compare, the search element with the middle element in the sorted list.
5. If both are matching, then display "Given element found!!!" and terminate the function

6. If both are not matching, then check whether the search element is smaller or larger than middle element.
7. If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sub list of the middle element.
8. If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sub list of the middle element.
9. Repeat the same process until we find the search element in the list or until sub list contains only one element.
10. If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.
11. Stop

Pseudo code

```
BinarySearch(a, l,  r, key)
{
        int m;
        int flag=0;
        if(l<=r)
        {
                m =(l + r)/2 ;
                if(key==a[m])
                    flag=m;
                 else if (key<a[m])
                    return BinarySearch(a, l, m-1, key) ;
                else
                    return BinarySearch(a, m+1, r, key) ;
        }
        else
        return flag;
 }
```

Analysis

From the above algorithm we can say that the running time of the algorithm is

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

By any one of the recurrence relation solving technique we get,

$T(n) = O(\log n)$

In the best case output is obtained at one run i.e. O(1) time if the key is at middle.

In the worst case the output is at the end of the array so running time is O(log n) time.

In the average case also running time is O(log n).

Min-Max Finding

To find the maximum and minimum numbers in a given array of elements of size n is called min-max finding algorithm. There are two method for finding minimum and maximum element from given array of elements. Namely iterative approach and divide and conquer approach. First we are representing the iterative method and then we will present divide and conquer approach.

Iterative Method

This 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[0]
Min = numbers[0]
for i = 1 to n do
        if numbers[i] > max then
                Max = numbers[i]
        if numbers[i] < min then
                Min = numbers[i]
Return (max, min)
```

Analysis

The number of comparison in this method is 2n - 2.

= O(1) x O(n) – O(1)

= O(n) – O(1)

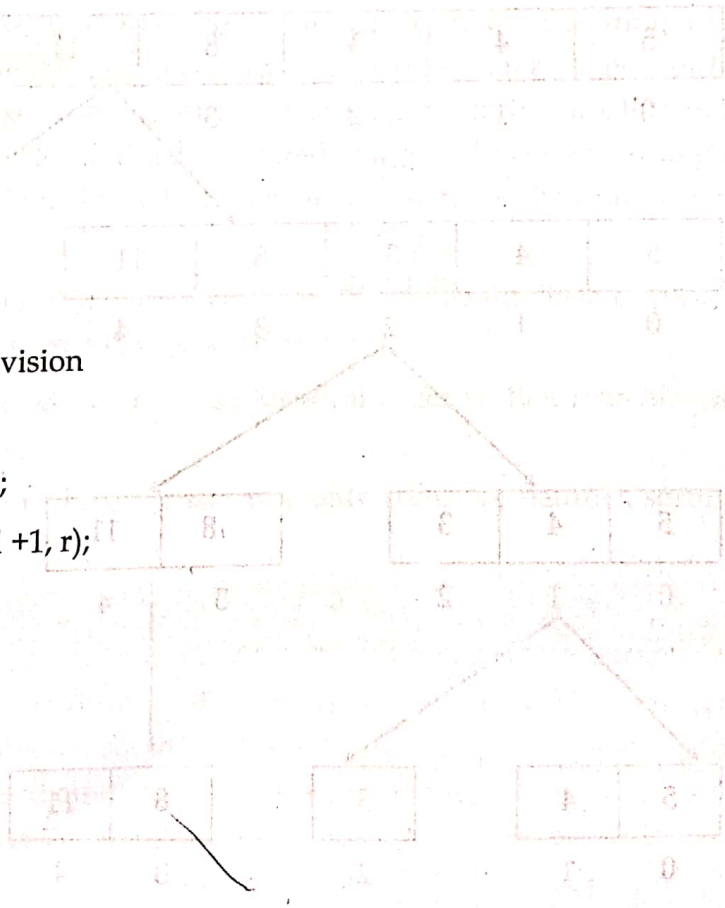
= O(n)

The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

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. The main idea behind the algorithm is: if the number of elements is 1 or 2 then max and min are obtained trivially. Otherwise split problem into approximately equal part and solved recursively.

```
MinMax(l, r)
{
    if(l = = r)
    max = min = A[l];
    else if(l = r-1)
    {
        if(A[l] < A[r])
        {
            max = A[r];
            min = A[l];
        }
        else
        {
            max = A[l];
            min = A[r];
        }
    }
    else
    {
        //Divide the problems
        mid = (l + r)/2; //integer division
        //solve the subproblems
        {min, max}=MinMax(l, mid);
        {min1, max1}= MinMax(mid +1, r);
        //Combine the solutions
        if(max1 > max)
            max = max1;
        if(min1 < min)
            min = min1;
    }
}
```



Analysis

We can give recurrence relation as below for MinMax algorithm in terms of number of comparisons. Since, there are two sub-problems each of size n/2. In every time problem divided into two equal halves and we do not need merging the solution of sub problems.

So size of sub problems=n/2

Dividing and merging sub problems= constant time = O(1)

Then we can define their recurrence relation as,

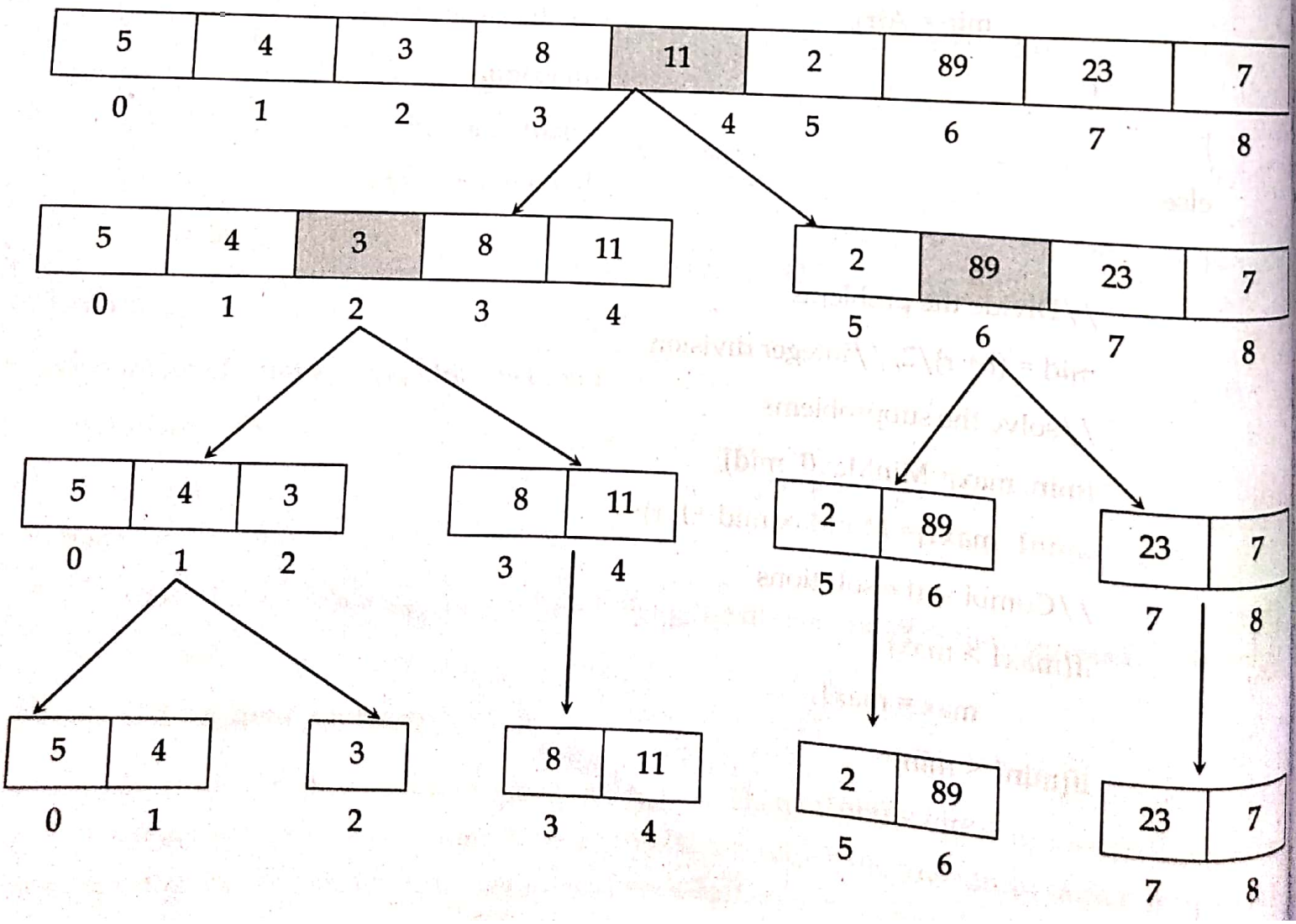
$$T(n) = 2T(n/2) + 1, \text{ if } n>2$$

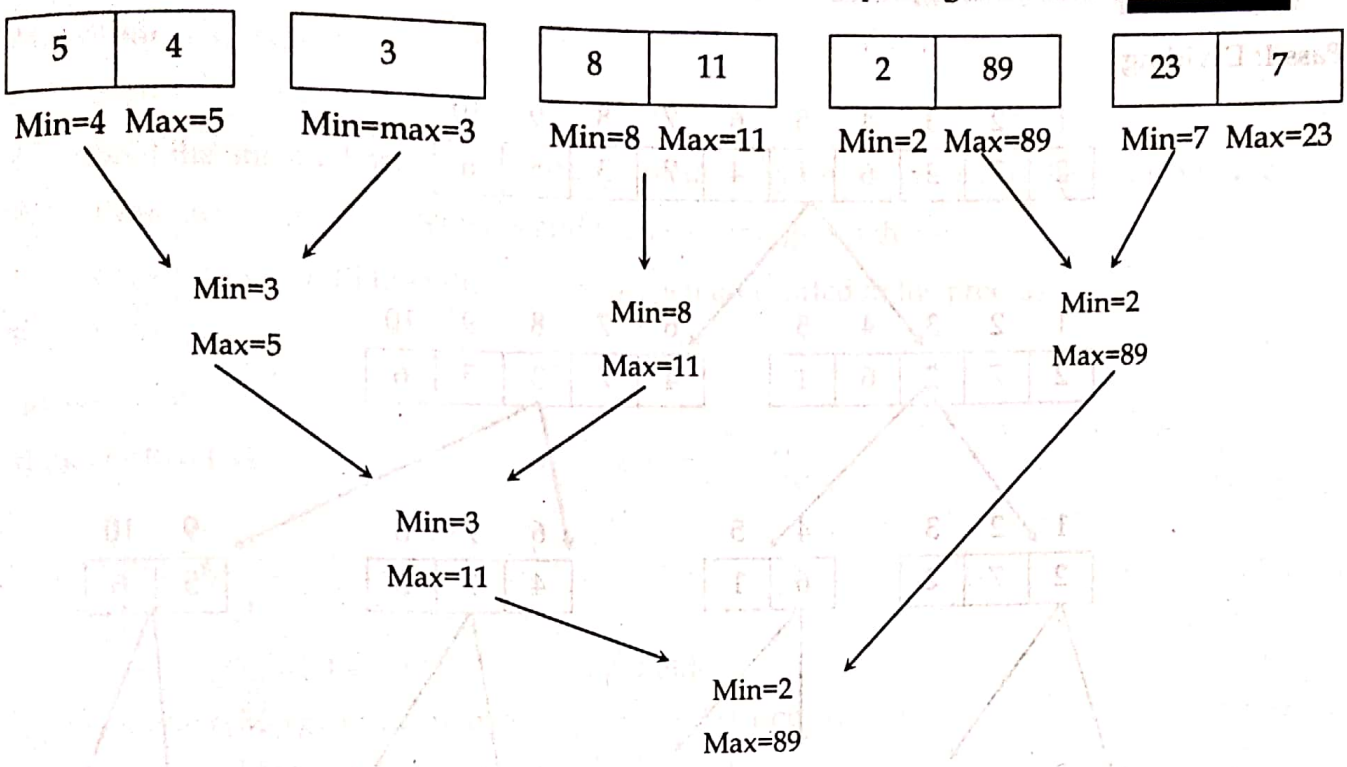
$$T(n) = 1, \text{ if } n \leq 2$$

Solving the recurrence by using any one of the recurrence relation solving technique we get, T(n)=O(n)

Tracing: Trace the algorithm for following array of elements by using recursive approach of min-max algorithm.

A[ ] = {5, 4, 3, 8, 11, 2, 89, 23, 7}





Sorting Algorithms

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure. The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios:

- **Telephone Directory:** The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary:** The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

There are a lot of searching techniques but here we describe only recursive nature's sorting algorithms.

Merge Sort

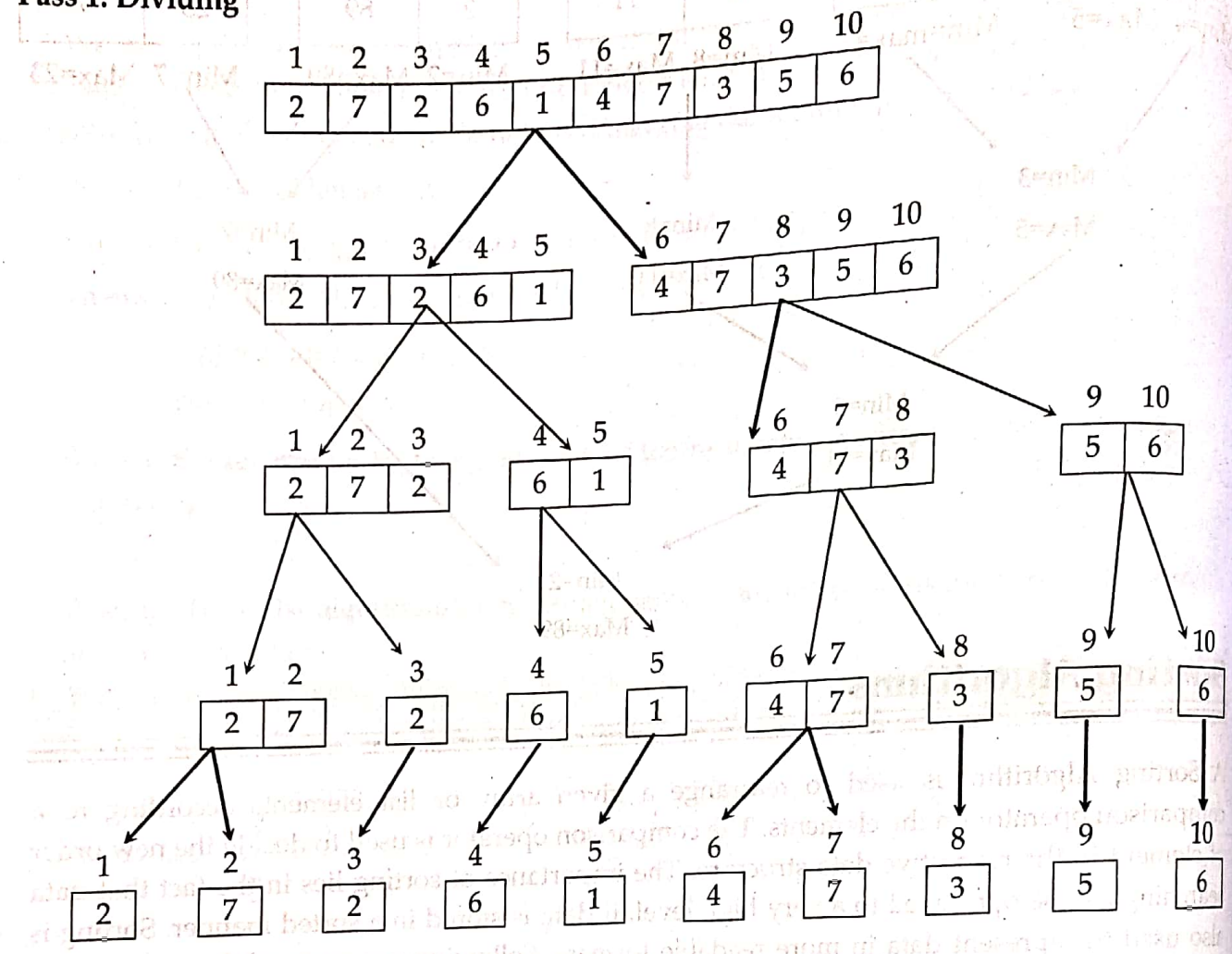
Merge sort is an efficient sorting algorithm which involves merging two or more sorted files into a third sorted file. Merging is the process of combining two or more sorted files into a third sorted file. The merge sort algorithm is based on divide and conquer method.

The process of merge sort can be formalized into three basic operations.

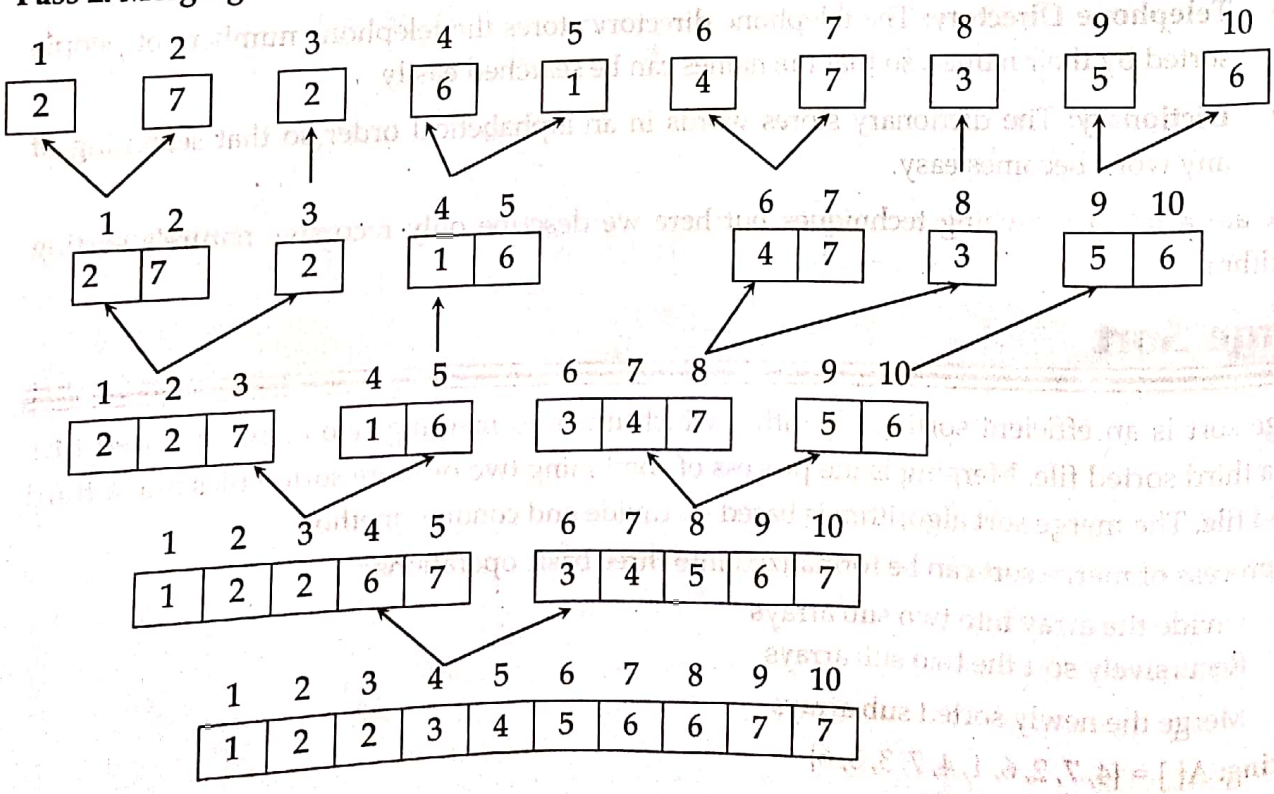
1. Divide the array into two sub arrays
2. Recursively sort the two sub arrays
3. Merge the newly sorted sub arrays

Tracing: A[ ] = {4, 7, 2, 6, 1, 4, 7, 3, 5, 6}

Pass 1: Dividing



Pass 2: Merging



Algorithm

1. Start
2. Split the unsorted list into two groups recursively until there is one element per group
3. Compare each of the elements and then sort and group them
4. Repeat step 2 until the whole list is merged and sorted in the process
5. Stop

Pseudo code

```
MergeSort(A, l, r)
{
        If( l< r)
        {
                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]
}
```

Time Complexity

No of sub-problems=2

Size of each subproblem=n/2

Dividing cost=constant

Merging cost=n

Thus recurrence relation for Merge sort is;

$T(n) = 1$ if n=1

$T(n) = 2\,T(n/2) + O(n)$ if n>1

By solving given recurrence relation, we get,

Time Complexity = $T(n) = O(n \log n)$

Quick Sort

As its name implies, **quick sort** is a fast divide-and-conquer algorithm. Its average running time is O(n logn) . Its speed is mainly due to a very tight and highly optimized inner loop. It has quadratic worst-case performance, which can be made statistically unlikely to occur with a little effort. On the one hand, the quick sort algorithm is relatively simple to understand and prove correct because it relies on recursion. On the other hand, it is a tricky algorithm to implement because minute changes in the code can make significant differences in running time. It has two phases:

- The partition phase and
- The sort phase

As we will see, most of the work is done in the partition phase - it works out where to divide the work. The sort phase simply sorts the two smaller problems that are generated in the partition phase. This makes Quick sort a good example of the **divide and conquers** strategy for solving problems.

- **Divide**

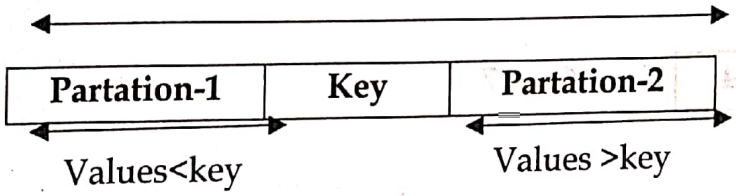
  Partition the array A[l…r] into two sub-arrays A[l…pivot] and A[pivot+1...r], such that each element of A[l...pivot] is smaller than or equal to each element in A[pivot+1...r].

- **Conquer**

  Recursively sort A[l…pivot] and A[pivot+1…r] using Quick sort

- **Combine**

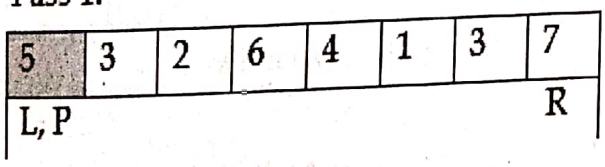
  Trivial: the arrays are sorted in place. No additional work is required to combine them.

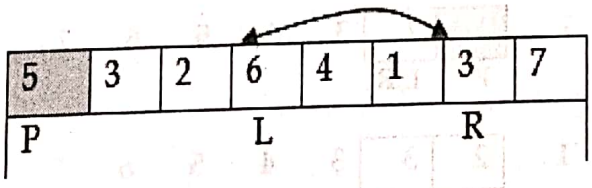


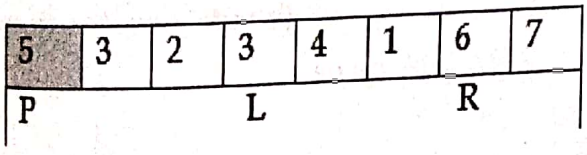
Tracing: Sort the following data items by using Quick sort

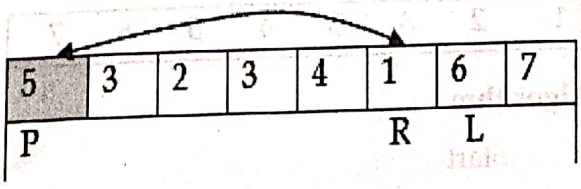
A[ ]={5, 3, 2, 6, 4, 1, 3, 7}

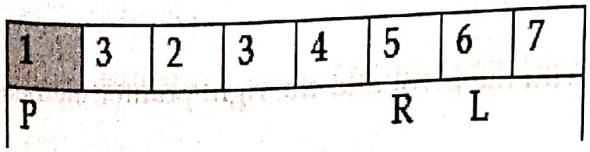
Pass 1:

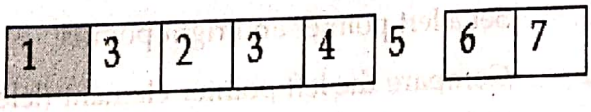




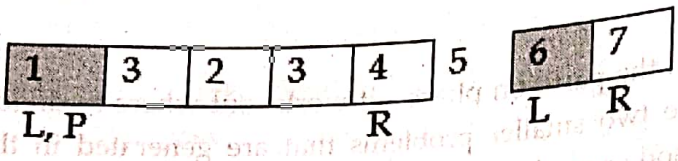


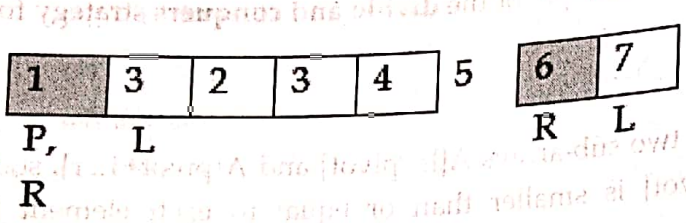




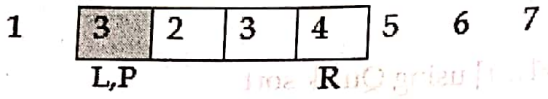


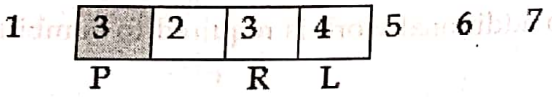
Pass 2:



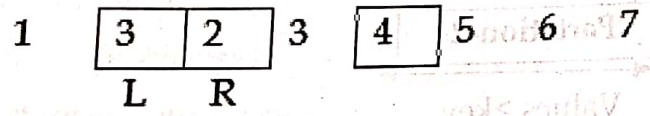


Pass 3:

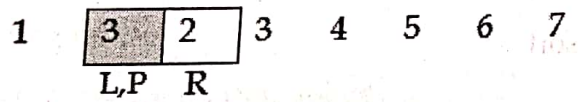


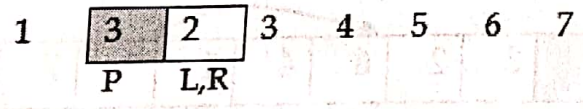


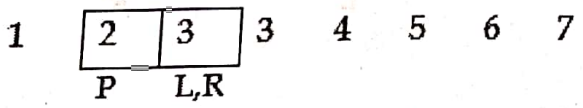
Pass 4:

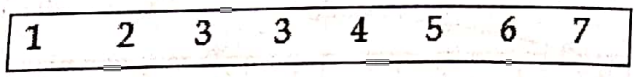


Pass 5:









Algorithm

1. Start
2. Choose a pivot
3. Set a left pointer and right pointer
4. Compare the left pointer element (lelement) with the pivot and the right pointer element (relement) with the pivot.

5. Check if lelement<pivot and relement>pivot:
   a. If yes, increment the left pointer and decrement the right pointer
   b. If not, swap the lelement and relement
6. When left >= right, swap the pivot with either left or right pointer.
7. Repeat steps 1 - 5 on the left half and the right half of the list till the entire list is sorted.
8. Stop

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

Time Complexity

Best Case

Quick sort gives best time complexity when elements are divided into two partitions of equal size; therefore recurrence relation for this case is;

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

By solving this recurrence, we get,

$$T(n)=O(n \log n)$$

Worst Case

Quick sort gives worst case when elements are already sorted. In this case one partition contains the n-1 elements and another partition contains no element. Therefore, its recurrence relation is;

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

By solving this recurrence relation, we get

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

Average Case

It is the case between best case and worst case. All permutations of the input numbers are equally likely. On a random input array, we will have a mix of well balanced and unbalanced splits. Good and bad splits are randomly distributed across throughout the tree. Suppose we are alternate: Balanced, Unbalanced, Balanced.

$B(n)= 2UB(n/2) + \Theta(n)$ Balanced

$UB(n)= B(n -1) + \Theta(n)$ Unbalanced

Solving: $B(n) = 2(B(n/2 -1) + \Theta(n/2)) + \Theta(n)$

$= 2B(n/2 -1) + \Theta(n)$

$= \Theta(n \log n)$

Heaps

A heap is an almost complete binary tree whose elements have keys that satisfy the following heap property: the keys along any path from root to leaf are descending (i.e. non-increasing). Heaps could represent family descendant trees because the heap property means that every parent is older than its children.

In brief, a heap is an almost complete binary tree of n nodes such that the value of each node is less than or equal to the value in parent node. This type of heap is called max heap. By default the heap is max heap. There are two types of heap:

1. Max heap and
2. Min heap

Heaps are used to implement priority queues and to the heap sort algorithm.

Example: Construct a heap from a set of 6 elements {15, 19, 10, 7, 17, 16}

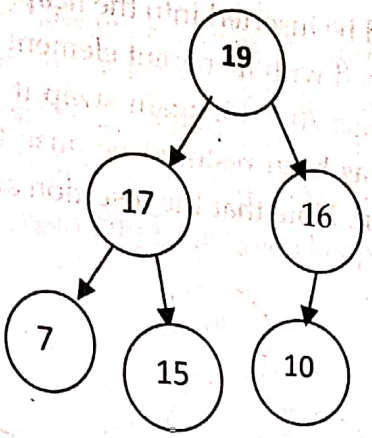
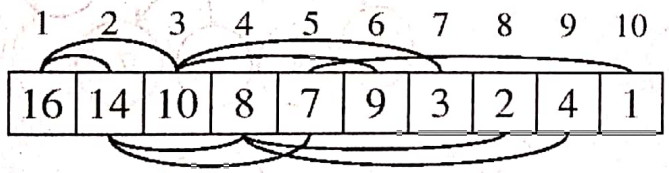


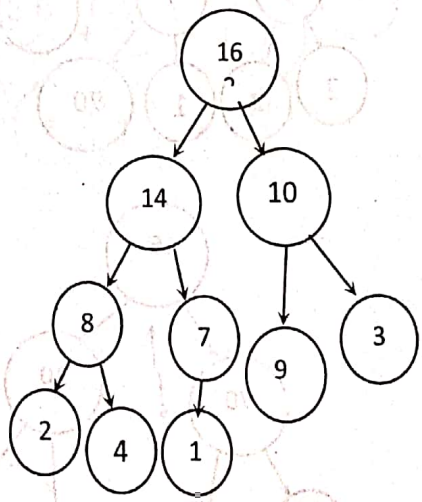
Fig: Heap of given set of elements

Array Representation of Heaps

Every complete binary tree has a natural mapping into an array. The mapping is obtained from a level-order traversal of the tree. In the resulting array, the parent of the element at index i is at index $i/2$, and the children are at indexes $2i$ and $2i+1$.

- 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 sub-array A[($\lfloor n/2 \rfloor$+1) ..... n] are leaves



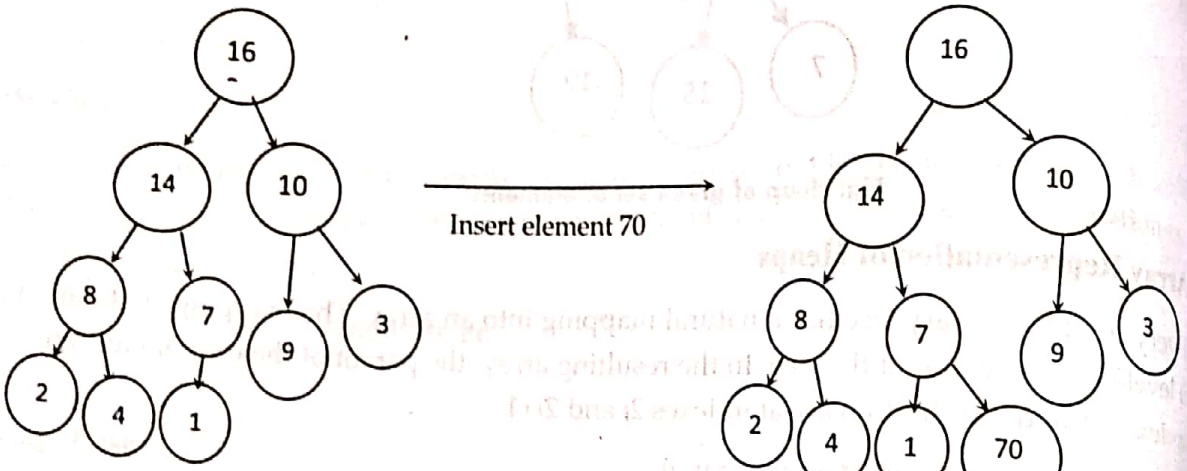


Inserting element to an existing heap

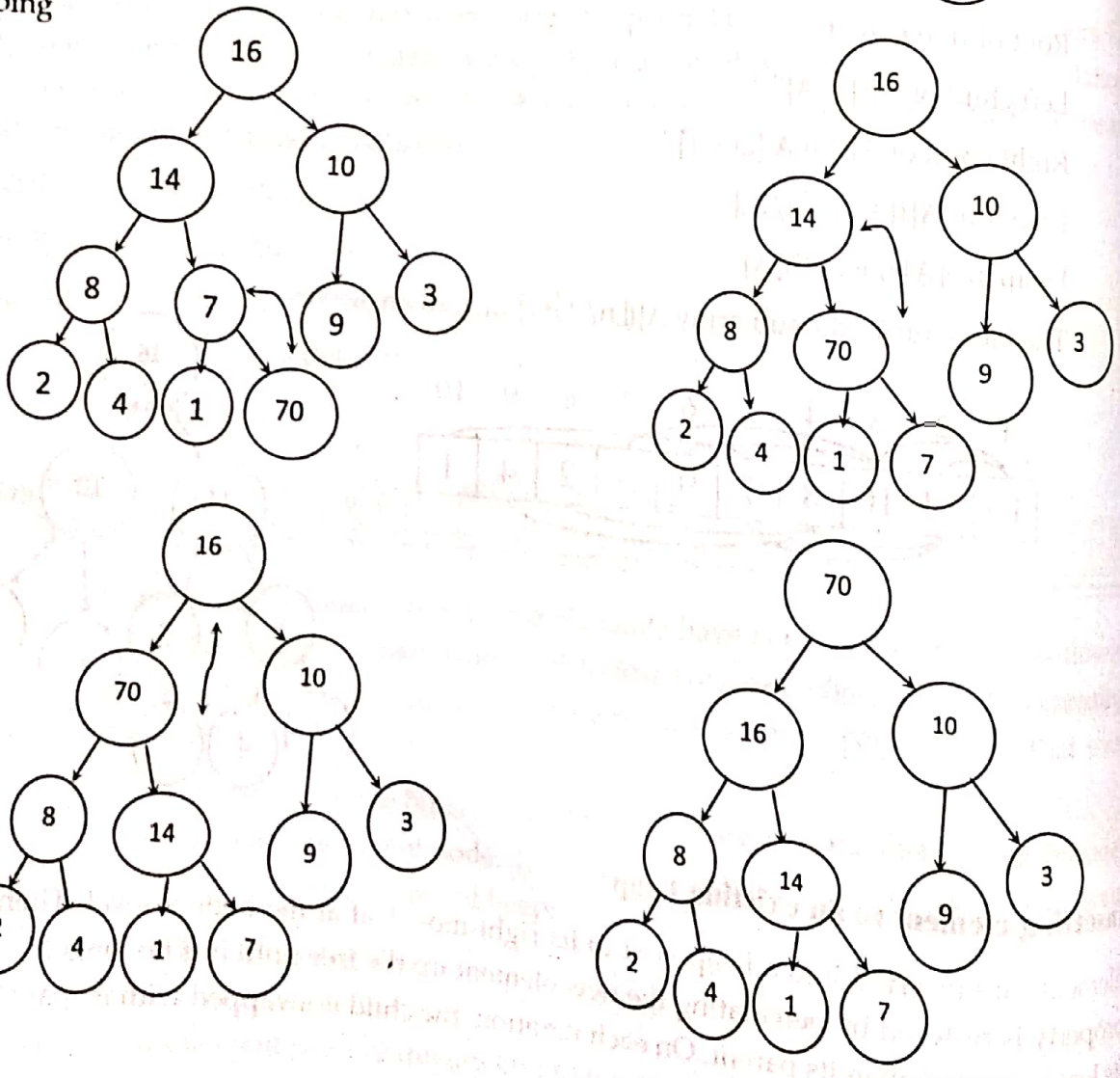
Elements are inserted into a heap next to its right-most leaf at the bottom level. Then the heap property is restored by percolating the new element up the tree until it is no longer "older" (i.e., its key is greater) than its parent. On each iteration, the child is swapped with its parent.

Example: Insert element 70 to given heap

Figure below shows how the key 70 would be inserted into the heap. The element 70 is added to the tree as a new last leaf. Then it is swapped with its parent element 7 because 70 > 7. Then it is swapped with its parent element 14 because 70 > 14, again swap it with its parent element 16 because 70>16. Now the heap property has been restored because the new element 70 is less than its parent and greater than its children. Note that the insertion affects only the nodes along a single root-to-leaf path.



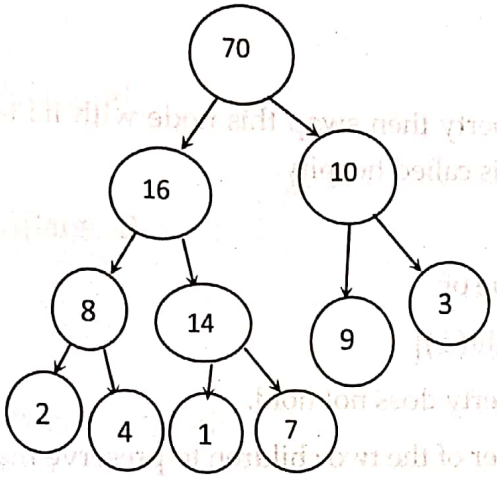
Swapping



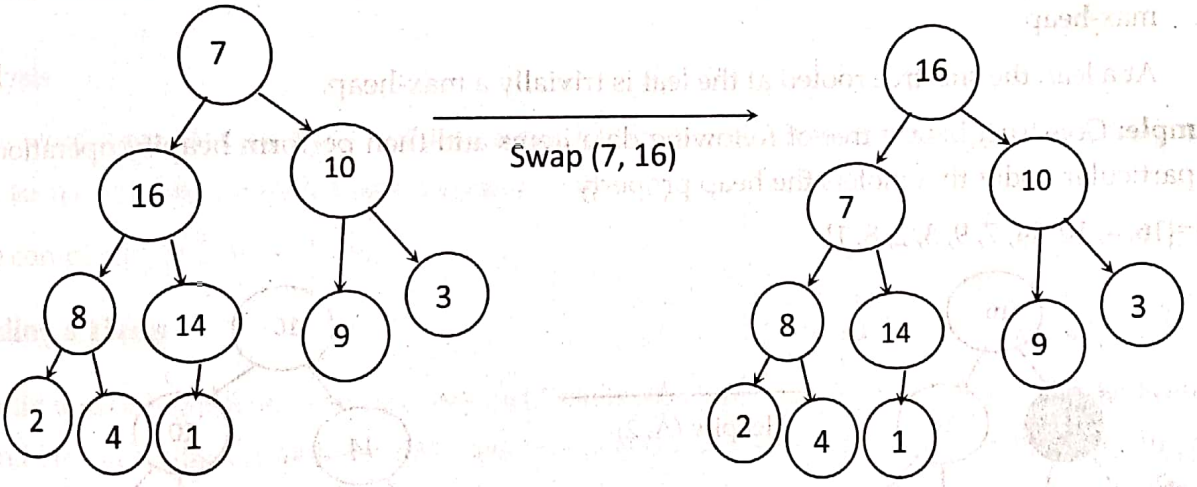
Deleting element from an existing heap

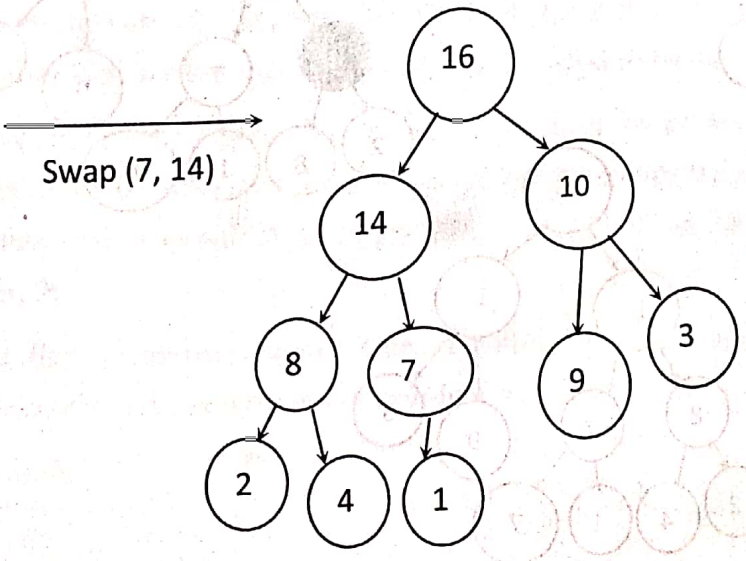
The heap removal algorithm always removes the root element from the tree. This is done by moving the last leaf element into the root element and then restoring the heap property by percolating the new root element down the tree until it is no longer "younger" (i.e., its key is less) than its children. On each iteration, the parent is swapped with the older of its two children.

Example: Removing root element (70) from given heap shown below



Solution:





Operations on Heaps

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

Heapify Property

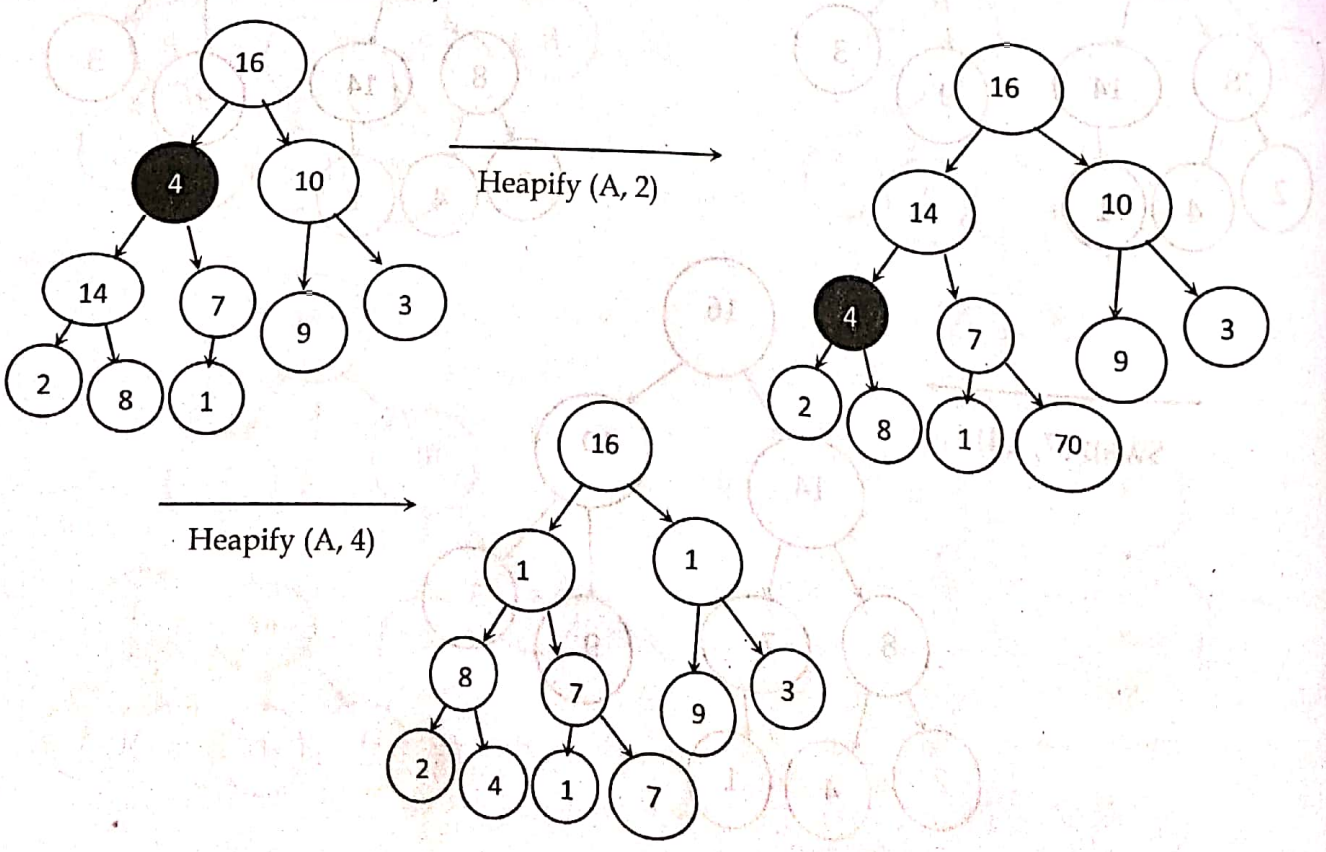
If any node violets the heap property then swap this node with its larger children to maintain the heap property, this operation is called heapify.

MAX-HEAPIFY operation

1. Find location of largest value of:
   A[i], A[Left(i)] and A[Right(i)]
2. If not A[i], max-heap property does not hold.
3. Exchange A[i] with the larger of the two children to preserve max-heap property.
4. Continue this process of compare/exchange down the heap until sub-tree rooted at *i* is a max-heap.
5. At a leaf, the sub-tree rooted at the leaf is trivially a max-heap.

Example: Construct binary tree of following data items and then perform heapify operation on the particular nodes that violets the heap property.

A[ ]={16, 4, 10, 14, 7, 9, 3, 2, 8, 1}



Algorithm

```
Max-Heapify(A, i, n)
{
    l = Left(i)
    r = Right(i)
    largest=i;
    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)
}
```

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)

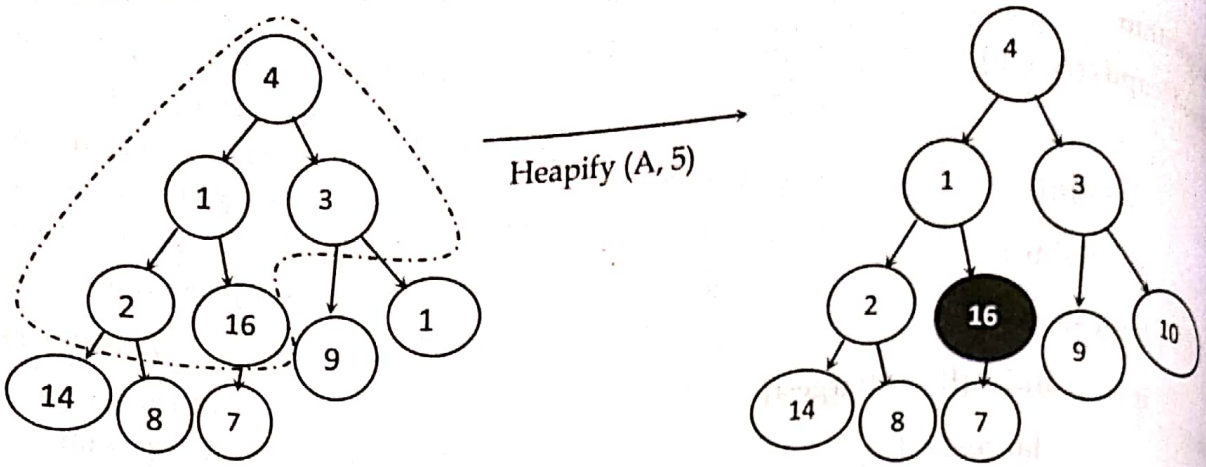
Building a Heap

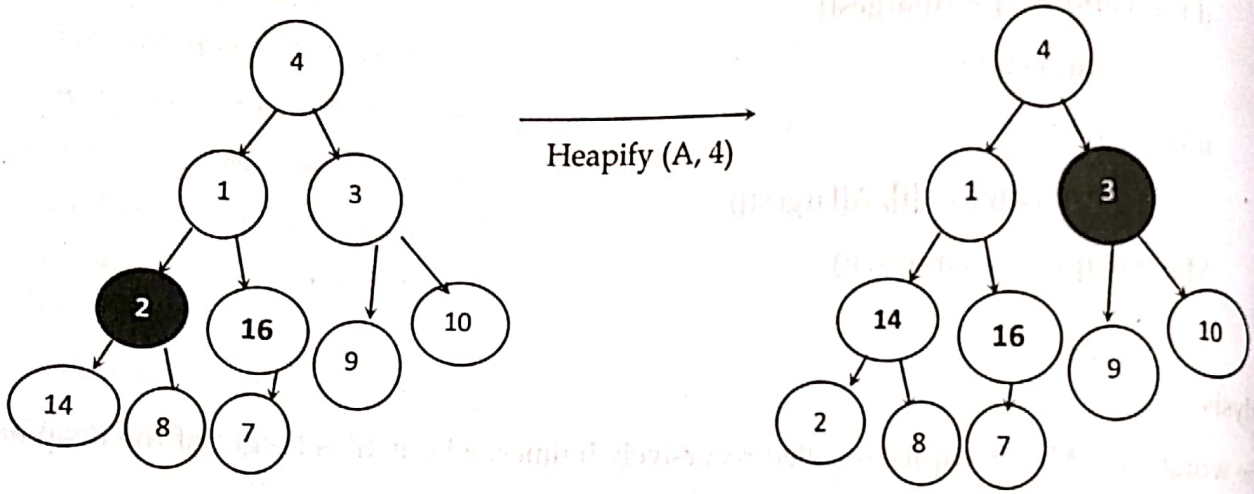
To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied on all the elements including the root element. In the case of complete tree, the first index of non-leaf node is given by n/2 - 1. All other nodes after that are leaf-nodes and thus don't need to be heapified.

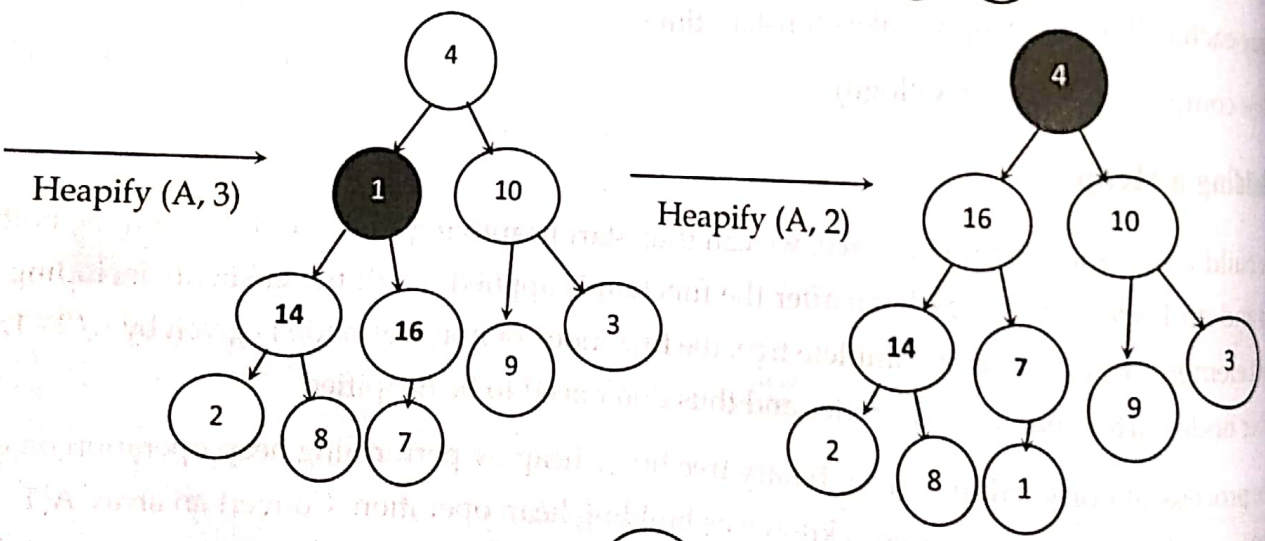
The process of converting a given binary tree into a heap by performing heap operation on each of the non-leaf node of the tree is known as building heap operation. Convert an array A[1 … n] into a max-heap (n = length[A]). Apply MAX-HEAPIFY on elements between 1 and floor function of (n/2).

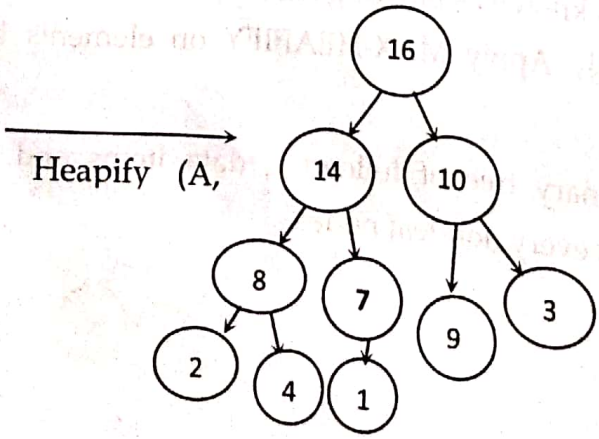
Example: At first Construct binary tree of following data items and then build heap by performing heapify operation on every non-leaf nodes.

A[ ]={4, 1, 3, 2, 16, 9, 10, 14, 8, 7}









Algorithm

```
Build-Max-Heap(A)
{
        n = length[A]
        for (i= floor(n/2); i>= 1; i--)
        {
        MAX-HEAPIFY(A, i, n);
        }
}
```

Heapsort

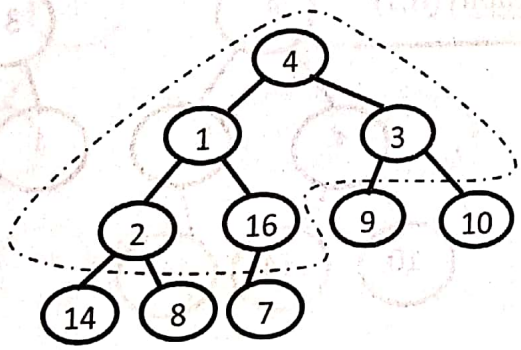
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element. To sort given set of n elements by using heap sort, it follow following steps

- 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
- Perform Max-Heapify operation on the new root node
- Repeat this process until only one node remains

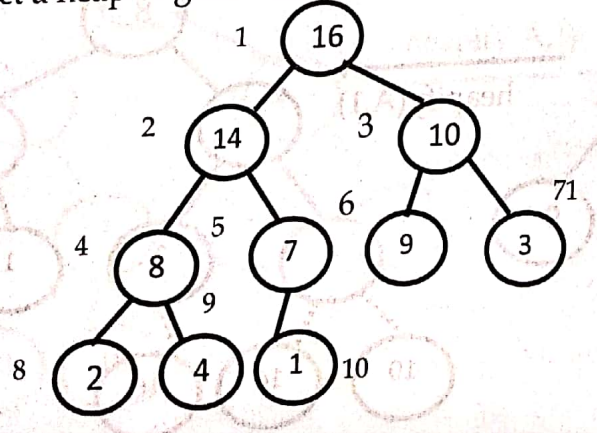
Example: Sort following elements by using heap sort

A[ ] ={4, 1, 3, 2, 16, 9, 10, 14, 8, 7}

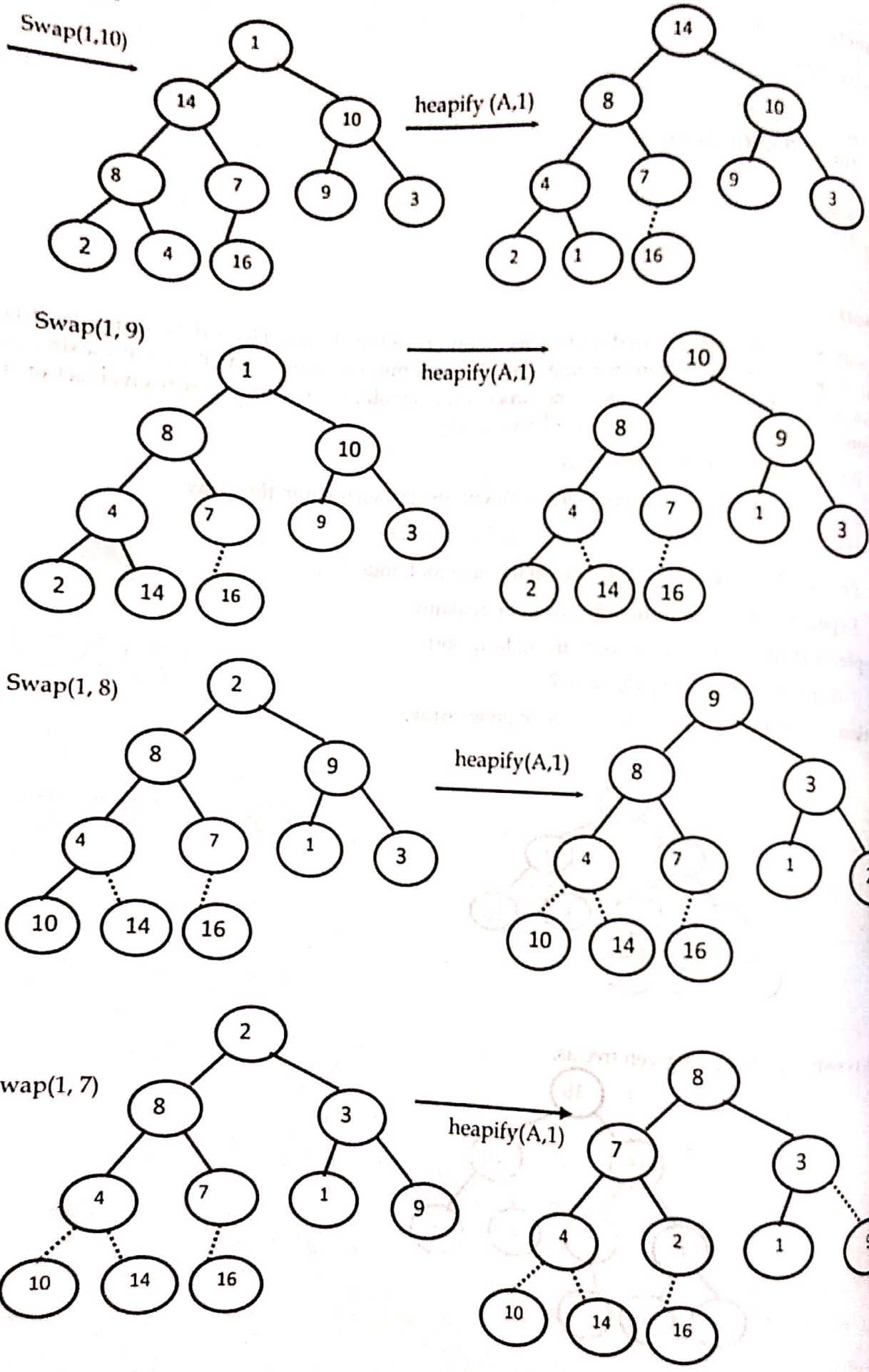
Solution: At first construct a binary tree of given array,

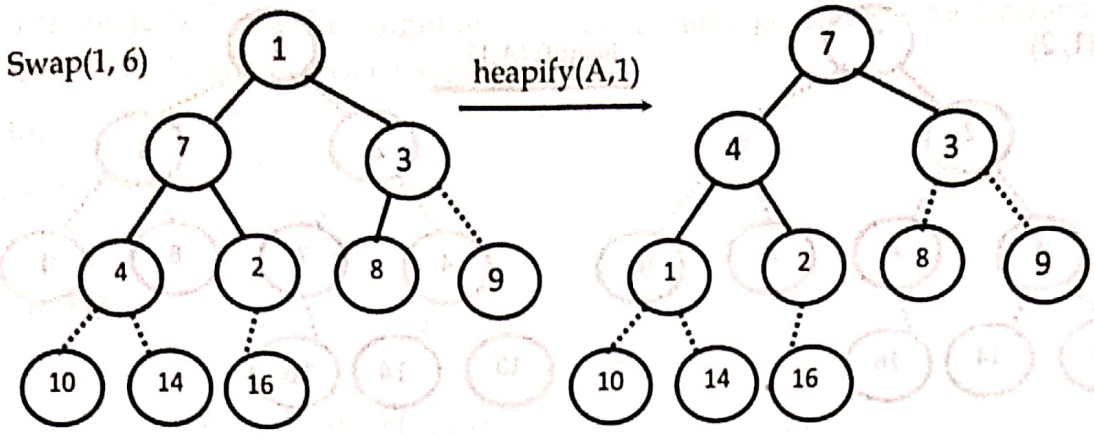


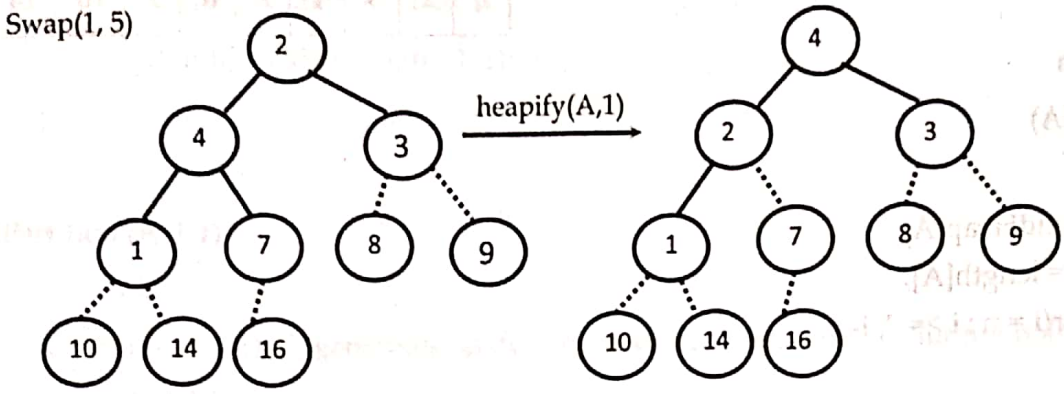
Now construct a heap of given tree as,

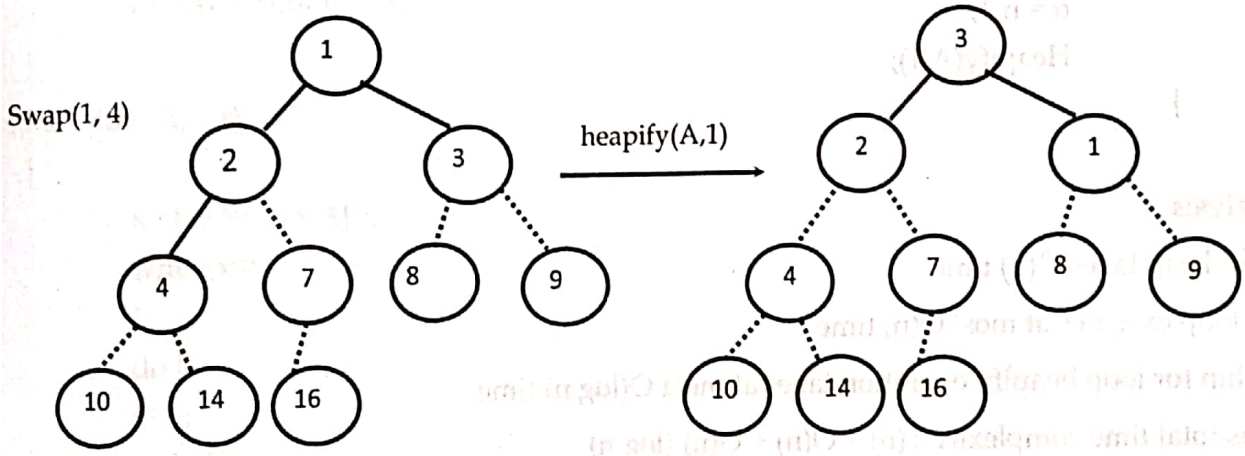


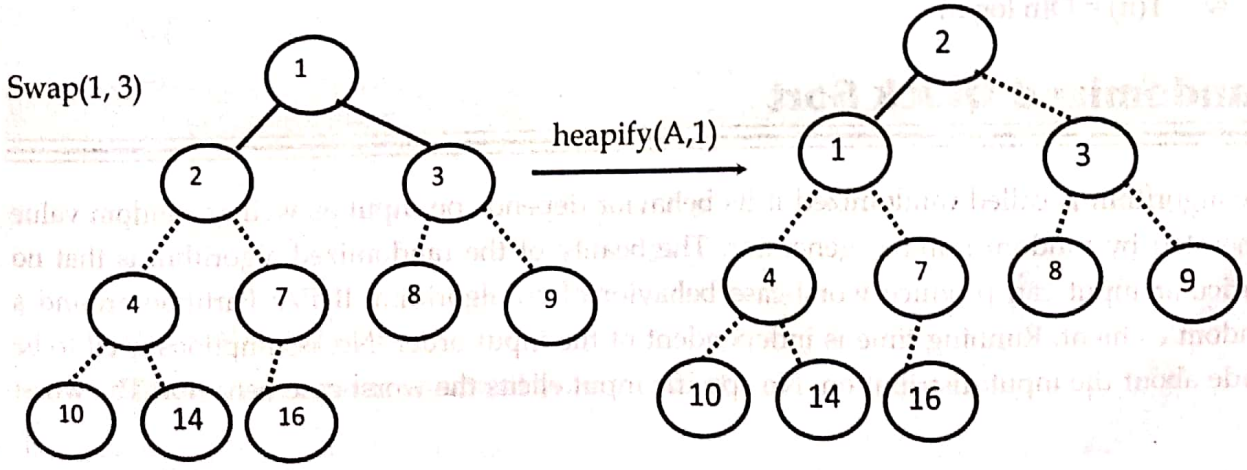
Heap sort

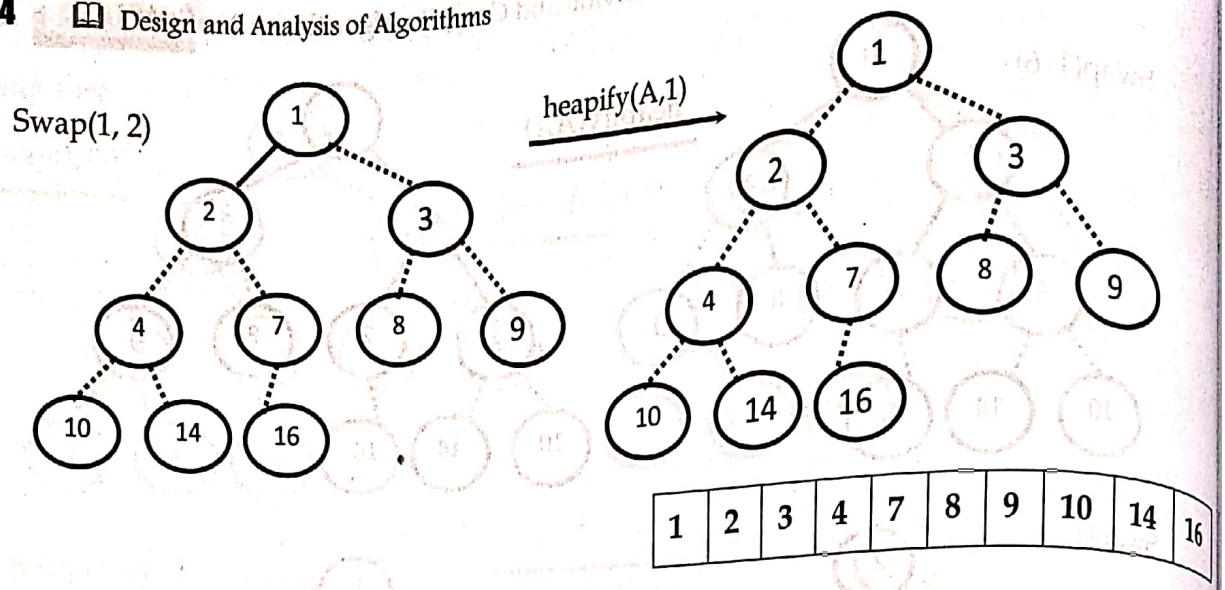












Algorithm

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

Analysis

Build heap takes O(n) time

For loop executes at most O(n) time

Within for loop heapify operation takes at most O(log n) time

Thus total time complexity T(n) = O(n) + O(n) (log n)

⇨ T(n) = O(n log n)

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!

Algorithm:

```
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);
}
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
}
```

Time Complexity

Worst Case

T(n) = worst-case running time

Let k be the partition element then there are two sub problems of size k and (n-k),

Since there are n elements so we need at most O(n) time for dividing.

Thus their recurrence relation can be defined as,

$$T(n) = \max_{1 \le k \le n-1} (T(k) + T(n-k)) + O(n) \quad \ldots\ldots(1)$$

Where, k is some partitioned point produced by random number generator

Now, by using substitution method to show that the running time of Quick sort is $O(n^2)$

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

$\Rightarrow$ $T(n) \le cn^2$ .....(2)

Now proof this by using mathematical induction

Basic step: for n=1,

$T(1) \le c.\ 1^2$

Or $1 \le c$ which is true for $c > 0$

Inductive step:

Let's assume that it is true for all $k < n$

I.e. $T(k) \le ck^2$ for any $k < n$

It is also true for k=n-k,

I.e. $T(n-k) \le c\ (n-k)^2$

Now equation 1 becomes,

$$T(n) \le \max_{1 \le k \le n-1} (ck^2 + c(n-k)^2) + O(n)$$

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

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

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

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

$$\le cn^2$$

$\Rightarrow$ $T(n) = O(n^2)$

Average Case

To analyze average case, assume that all the input elements are distinct for simplicity. If we are to take care of duplicate elements also the complexity bound is same but it needs more intricate analysis. Consider the probability of choosing pivot from n elements is equally likely i.e. 1/n.

Now we give recurrence relation for the algorithm as

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

For some k = 1, 2... n-1, T(k) and T(n-k) is repeated two times

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

$$nT(n) = 2\sum_{k=1}^{n-1} T(k) + O(n^2) \quad \ldots\ldots(1)$$

Similarly,

$$(n-1)T(n-1) = 2\sum_{k=1}^{n-2} T(k) + O(n-1)^2 \quad \ldots\ldots(2)$$

Subtracting equation 1 from 2 we get,

$$nT(n) - (n-1)T(n-1) = 2\sum_{k=1}^{n-1} T(k) + O(n^2) - 2\sum_{k=1}^{n-2} T(k) + O(n-1)^2$$

$$\text{or, } nT(n) - (n-1)T(n-1) = 2\sum_{k=1}^{n-1} T(k) - 2\sum_{k=1}^{n-2} T(k) + O(n^2) - O(n-1)^2$$

$$= 2T(n-1) + n^2 - n^2 + 2n - 1$$

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

$$\text{or, } nT(n) = (n-1)T(n-1) + 2T(n-1) + (2n-1)$$

$$= T(n-1)[(n-1) + 2] + (2n-1)$$

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

Or, $nT(n) - (n+1)T(n-1) = 2n-1$

Dividing both sides by n (n+1) we get

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

Let $A_n = T(n)/(n+1)$

$A_n = A_{n-1} + (2n-1)/n(n+1)$

$$A_n = \sum_{i=1}^{n} 2i - 1/i(i+1)$$ [since recurrence rel^n of sum of first n natural number is $s_n = s_{n-1} + n$]

$$An \approx \sum_{i=1}^{n} 2i / i(i+1)$$

$$An \approx 2\sum_{i=1}^{n} 1/(i+1)$$

This is a Harmonic series,

Hence An ≈ 2logn.....(3)

Since An = T(n) /(n+1)

Or, 2 log(n) = T(n) /(n+1)

⇨ T(n) = 2 (n+1) log(n)

Or, T(n) = 2n logn +2logn

⇨ T(n) = O(n log n)

Comparison of various sorting algorithms

In brief the worst, best and average case complexities of various sorting algorithms are tabulated below;

Sorting technique	Worst case	Average case	Best case	Comment
Insertion sort	$O(n^2)$	$O(n^2)$	O(n)	
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Unstable
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	
Merge sort	O(n log n)	O(n log n)	O(n log n)	Require extra memory
Heap sort	O(n log n)	O(n log n)	O(n log n)	Large constant
Quick sort	$O(n^2)$	O(n log n)	O(n log n)	Small constant

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. i^{th} order statistic of a set of elements gives i^{th} largest (smallest) element. In general let's think of i^{th} order statistic gives i^{th} smallest. Then minimum is first order statistic and the maximum is last order statistic. Similarly a median is given by i^{th} order statistic where $i = \frac{(n+1)}{2}$ for odd n and $i = \frac{n}{2}$ and $\frac{n}{2+1}$ for even n. This kind of problem commonly called selection problem.

Notation and examples

For example, suppose that four numbers are observed or recorded, resulting in a sample of size 4. If the sample values are

6, 9, 3, 8

They will usually be denoted

$x_1 = 6, x_2 = 9, x_3 = 3, x_4 = 8$

Where the subscript i in x_i indicates simply the order in which the observations were recorded and is usually assumed not to be significant. A case when the order is significant is when the observations are part of a time series.

The order statistics would be denoted

$x_{(1)}=3, x_{(2)}=6, x_{(3)}=8, x_{(4)}=9$

Where the subscript (i) enclosed in parentheses indicates the i^{th} order statistic of the sample. The first order statistic (or smallest order statistic) is always the minimum of the sample, that is,

$x_{(1)}= \min \{ x_1,.....x_4\}$

Where, following a common convention, we use upper-case letters to refer to random variables, and lower-case letters (as above) to refer to their actual observed values. Similarly, for a sample of size n, the nth order statistic (or largest order statistic) is the maximum, that is,

$x_{(n)}= \max \{ x_1,.....x_4\}$

The sample range is the difference between the maximum and minimum. It is a function of the order statistics:

$\text{Range}\{ x_1,.....x_4\} = x_{(n)} - x_{(1)}$

Nonlinear general selection algorithm

We can construct a simple, but inefficient general algorithm for finding the k^{th} smallest or k^{th} largest item in a list. This is efficient when k is small. To accomplish this, we simply find the most extreme value and move it to the beginning until we reach our desired index.

```
Select(A, k, n)
{
    for( i=0; i < k; i++)
    {
        minindex = i;
        minvalue = A[i];
        for(j=i+1; j < n; j++)
        {
```

```
                    if(A[j] < minvalue)
                    {
                        minindex = j;
                        minvalue = A[j];
                    }
                    swap(A[i], A[minindex]);
                }
            }
            return A[k];
        }
```

Analysis

When i=0, inner loop executes n-1 times

When i=1, inner loop executes n-2 times

When i=2, inner loop executes n-3 times

.....

When i=k-1 inner loop executes n-(k-1+1) times

Thus, Time Complexity = (n-1) + (n-2) + ...(n-k)

In worst case if k=n then,

T(n)=0+1+2+3+4+.....+(n-2)+(n-1)

=n(n-1)/2 [since s_n=n(n+1)/2]

= $O(n^2)$

Selection in Expected Linear Time

The general selection problem appears more difficult than the simple problem of finding a minimum. In this section, we present a divide-and-conquer algorithm for the selection problem. As in quick sort, the idea is to partition the input array recursively. But unlike quick sort, which recursively processes both sides of the partition, Randomized-Select only works on one side of the partition. The main idea for this problem solving is to partition the element set as in Quick Sort where partition is randomized one.

Algorithm

Let A be the array of n elements, l be the leftmost index of A, r be the rightmost index of A and i be the index of element to be select.

RandSelect(A, l, r, i)

```
{
        if(l = =r )
                return A[ l];
        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);
}
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);
}
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
}
```

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 then we have the recurrence relation,

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

Where, max(j, n-j) = j, if $j >= \lceil n / 2 \rceil$

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

Here every T(j) or T(n – j) will repeat twice, one time from 1 to $\lceil n / 2 \rceil$ and second time from $\lceil n / 2 \rceil$ to (n-1), so we can write,

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

Using substitution method,

Let's guess T(n) = O(n)

Then we have to show that $T(n) \le c\, n$ .....(2)

Basic step: for n=1,

$T(1) \le c \cdot 1$

⇨ $1 \le c$ which is true for all $c > 0$

Inductive step: let's assume that it is true for all $j<n$

Then $T(j) \le c\, j$ .....(3)

Substituting on the relation (1) we get,

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

$$\text{Or,} \quad T(n) \le 2/n \{\sum_{j=1}^{n-1} cj - \sum_{j=1}^{\frac{n}{2}-1} cj\} + O(n)$$

$$= 2/n \{ c.\frac{n(n-1)}{2} - \frac{(\frac{n}{2}-1)(n/2-1+1)}{2}.c \} + O(n)$$

$$= 2/n \{ c.\frac{n(n-1)}{2} - \frac{n(\frac{n}{2}-1)}{4}.c \} + O(n)$$

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

$$= c\,n - c - c\,n/4 + c/2 + c\,n$$

$$= c\,n - [c + c\,n/4 - c/2 - c\,n]$$

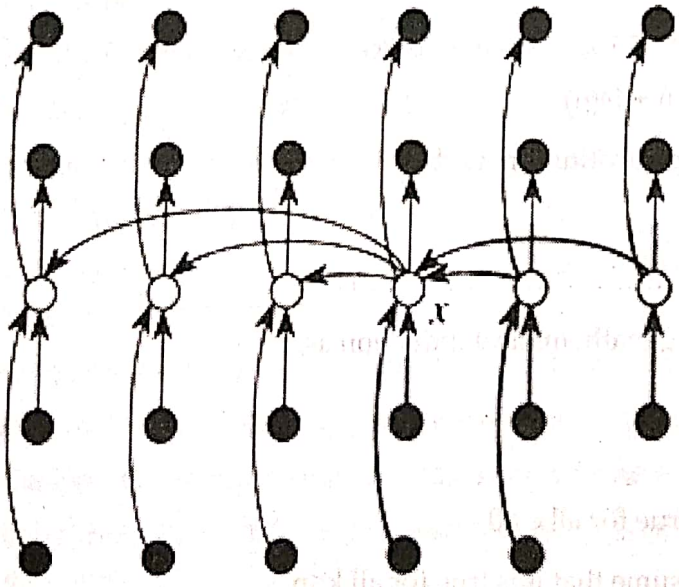
$$\leq c\,n$$

$\Rightarrow T(n) = O(n)$

Selection in Worst Case Linear Time

This algorithm determines the i^{th} smallest of an input array of n elements by executing the following steps:

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining n mod 5 elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by insertion sorting the elements of each group (of which there are 5 at most) and taking its middle element. (If the group has an even number of elements, take the larger of the two medians.)
3. Use Select recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2.
4. Partition the input array around the median-of-medians x using a modified version of Partition. Let k be the number of elements on the low side of the partition, so that n - k is the number of elements on the high side.
5. Use Select recursively to find the i^{th} smallest element on the low side if $i \leq k$, or the $(i - k)^{th}$ smallest element on the high side if $i > k$.



In the above figure 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.

Partition the n elements around pivot:

```
if (i==k) then
        Return (A[k])
else if (i < k) then
        Recursively to find i^th smallest element in first partition
else
        Recursively to find (i-k)^th smallest element in second partition
```

From above figure at least half the medians are ≤ x

Since there are $\lfloor n/5 \rfloor$ medians

Therefore $\frac{n/5}{2}$ medians are ≤ x

Or, n/10 medians are ≤ x

Since each medians contribute 3 elements which are ≤ x

⇨ 3 n/10 elements are ≤ x

Similarly, at least $3n/10$ elements are ≥ x

Since there are total n elements

If 3 n/10 elements are ≤ x

Then (n - 3n/10) =7n/10 elements are ≥ x

Now its recurrence relation is,

T(n) = T(n/5) + T(7n/10) + O(n)

Now solve this by using substitution method

Lets guess T(n)=O(n)

⇨ T(n) ≤ c n.....(1)

Now proof this by using mathematical induction as,

Basic step: for n=1,

T(1) ≤ c . 1

⇨ 1 ≤ c which is true for all c >0

Inductive step: let's assume that it is true for all k<n

Then T(k) ≤ c k.....(2)

It is also true for k=n/5 and 7n/10,

$\Rightarrow T(n/5) \leq c.\ n/5$

Also $T(7n/10) \leq c.\ 7n/10$

Now from given recurrence relation,

$T(n) = T(n/5) + T(7n/10) + O(n)$

$T(n) \leq c\,n/5 + c.\ 7n/10 + O(n)$

$\leq c\,n - 4c\,n\ /\ 5 + 7cn/10 + c\,n$

$\leq c\,n - c\,n[4/5 - 7/10 - 1]$

$\leq c\,n$

$\Rightarrow T(n) \leq c\,n$

Thus, $T(n) = O(n)$

Discussion Exercise

1. Write Divide and Conquer recursive Merge sort algorithm and derive the time complexity of this algorithm.
2. Write Divide and Conquer recursive Quick sort algorithm and analyze the algorithm for average time complexity.
3. Derive the time complexity of Quick sort algorithm for worst case.
4. Distinguish between Merge sort and quick sort.
5. What is stable sorting method? Is Merge sort a stable sorting method?
6. Trace the quick sort algorithm for any 10 arbitrary elements.
7. Write down the algorithm for quick sort then analyze it with their best case, worst cans and average case time complexity
8. Define randomized quick sort. How it is differ from quick sort? Explain
9. Write down the algorithm for randomized quick sort then analyze it.
10. What is heap? Define heapify operation of heap with suitable example.
11. What is selection? How it is differ from searching? Explain linear time selection algorithm with example.
12. Define build heap operation. Construct heap of any 10 elements by using build heap operation.

13. Define heap sort. Sort following data items by using heap sort algorithm
    A[ ]= {3, 5, 2, 66, 4, 11, 9, 34}
14. Write down the algorithm for build heap and heap sort algorithms then analyze them.
15. Define min-max finding algorithm. Writhe down the min-max algorithm and analyze it.
16. Define divide and conquer strategy. Compare it with recursion.
17. Sort following data items by using quick sort and merge sort.
    A[]={4, 6, 55, 2, 1, 5, 6, 44, 90, 3, 1}
18. Define binary search. Writhe down the recursive algorithm for binary search algorithm and analyze it.
19. Differentiate between binary search and sequential search algorithm.
20. Compare the complexity analysis of different recursive sorting algorithm.

❑❑❑