UNIT I - INTRODUCTION

Algorithm analysis: Time and space complexity - Asymptotic Notations and its properties Best case, Worst case and average case analysis – Recurrence relation: substitution method - Lower bounds – searching: linear search, binary search and Interpolation Search, Pattern search: The naïve string-matching algorithm - Rabin-Karp algorithm - Knuth-Morris-Pratt algorithm. Sorting: Insertion sort – heap sort.

Important Questions:

Part A Questions

- Define time complexity and space complexity. Write an algorithm for adding n natural numbers and find the space required by that algorithm
- List the steps to write an Algorithm
- Define Big 'Oh' notation.
- Differentiate between Best, average and worst case efficiency.
- Define recurrence relation.
- How do you measure efficiency of the algorithm?
- Write an algorithm to find the area and circumference of a circle.
- How to measure algorithms running time?
- List the desirable properties of algorithms
- Write the recursive Fibonacci algorithm and its recurrence relation.
- Write an algorithm to compute the GCD of two numbers.

Part B Questions

- Discuss the concepts of asymptotic notations and its properties
- What is divide and conquer strategy? Explain binary search problem in detail.
- Solve the following using Brute-Force algorithm:
- Find whether the given string follows the specified pattern and return 0 or 1 accordingly

  Examples

  Pattern: "abba", input "redblueredblue" should return 1
  Pattern: "aaaa", input "asdasdasdasd" should return 1
  Pattern: "aabb", input "xyzabcxyzabc" should return 0

APEC

1. Algorithm

1.1 Definition

The sequence of steps to be performed in order to solve a problem by the computer is known as an algorithm.

$$Programs = Algorithms + Data$$

Another way to describe algorithm is the sequence of unambiguous instructions. It starts from an initial input of instructions that describe a computation and proceeds through a finite number of well-defined successive steps, producing an output and a final ending state.

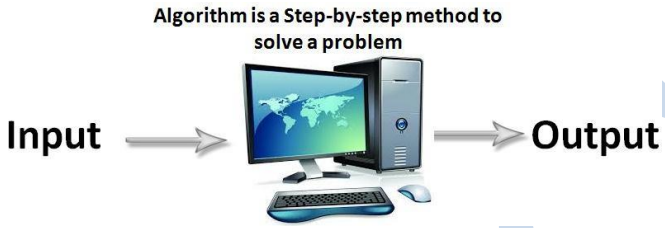


Figure 1.1 Definition of the Algorithm

Algorithms was first developed by Persian scientist, astronomer and mathematician Abdullah Muhammad bin Musa al-Khwarizmi in 9th century. He was often cited as "The father of Algebra", and was responsible for the creation of the term "Algorithm".

1.1.1 Examples on Algorithms

Example 1:

Problem statement: *Calling a friend on the telephone*

Input: The telephone number of your friend.
Output: Talk to your friend

Algorithm Steps:

1. Pick up the phone and listen for a dial tone
2. Press each digit of the phone number on the phone
3. If busy, hang up phone, wait 2 minutes, jump to step 2
4. If no one answers, leave a message then hang up
5. If no answering machine, hang up and wait 2 hours, then jump to step 2
6. Talk to friend
7. Hang up phone

Example 2:

Problem statement: *Find the largest number in the given list of numbers*

Input: A list of positive integer numbers.
Output: Largest number.

Algorithm Steps:

1. Define a variable 'max' and initialize with '0'.
2. Compare first number (say 'x') in the list 'L' with 'max'.
3. If 'x' is larger than 'max', set 'max' to 'x'.

4. Repeat step 2 and step 3 for all numbers in the list 'L'.
5. Display the value of 'max' as a result.

1.1.2 Properties of an Algorithm

- *Finiteness:* The algorithm must always terminate after a finite number of steps.
- *Definiteness:* Each instruction must be clear, well-defined and precise. There should not be any ambiguity.
- *Effectiveness:* Each Instruction must be simple and be carried out in a finite amount of time.
- *Input:* An algorithm has zero or more inputs, taken from a specified set of objects.
- *Output:* An algorithm has one or more outputs, which have a specified relation to the inputs.
- *Feasibility:* It must be possible to perform each instruction.
- *Generality* - the algorithm must be able to work for a set of inputs rather than a single input.
- *Efficiency:* The term efficiency is measured in terms of time and space required by an algorithm to implement. Thus, an algorithm must ensure that it takes little time and less memory space.
- *Independent*: An algorithm must be language independent. It means that it should mainly focus on the input and the procedure required to get the output instead of depending upon the language.

1.1.3 Necessity to analyse the algorithm

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

Performance of an algorithm depends on the following parameters like -

- Whether that algorithm provides the exact solution for the problem statement
- Whether it is easy to understand
- Whether it is easy to implement
- How much space (memory) required to solve the problem
- How much time required to solve the problem

1.2 Algorithm analysis

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Algorithm analysis is the process of calculating space and time required by that algorithm. The term "analysis of algorithms" was coined by Donald Knuth.

Algorithm analysis is performed by using the following measures -

- *Space Complexity:* Space required to complete the task. It includes program space and data space
- *Time Complexity:* Time required to complete the task.

1.2.1 Space complexity

Space complexity is an amount of memory used by the algorithm (including the input values of the algorithm), to execute it completely and produce the result. We know that to execute an algorithm it must be loaded in the main memory. The memory can be used in different forms:

- Variables (This includes the constant values and temporary values)
- Program Instruction
- Execution

Space complexity includes both Auxiliary space and space used by input. Auxiliary Space is the extra space or temporary space used by an algorithm.

Memory Usage during program execution

- Instruction Space → used to save compiled instruction in the memory.
- Environmental Stack → used for storing the addresses while a module calls another module or functions during execution.
- Data space →used to store data, variables, and constants which are stored by the program and it is updated during execution.

Space complexity is a parallel concept to time complexity. If we need to create an array of size n, this will require $O(n)$ space. If we create a two-dimensional array of size $n*n$, this will require $O(n^2)$ space.

1.2.2 Time complexity

Time complexity of an algorithm measures the amount of time taken by an algorithm ie. the time taken to execute each statement of code in an algorithm.

Example:

- Time taken to execute 1 statement = `x` milliseconds.
- Time taken to execute `n` statement = `x * n` milliseconds
- To execute `n` statement inside a `FOR` loop = `x * n` milliseconds + `y` milliseconds
  where `y` milliseconds is the time taken to execute `FOR` loop.

1.2.3 Asymptotic Notation

To perform analysis of an algorithm, it is necessary to calculate the complexity of that algorithm. To calculate the complexity of an algorithm, exact amount of resource required. But it will not be provided. So instead of taking the exact amount of resource, we represent that complexity in a general form (Notation) for analysis process.

In asymptotic notation, the complexities of an algorithm are represented only by the most significant terms and ignore least significant terms (Here complexity is, Space Complexity or Time Complexity).

Example,

- Algorithm 1 : $25n^3 + 2n + 1$
- Algorithm 2 : $1223n^2 + 8n + 3$
  The term '2n + 1' have least significance than the term '$25n^3$', and the term '8n + 3' in algorithm has least significance than the term '$1223n^2$'.

Definition:

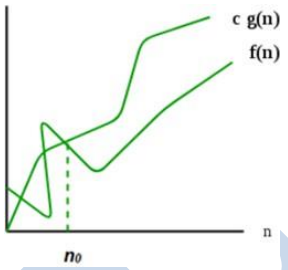
Asymptotic notations are mathematical tools to represent the time and space complexity of algorithms for asymptotic analysis.

There are mainly three asymptotic notations:

- Big-O Notation (O-notation)
- Omega Notation (Ω-notation)
- Theta Notation (Θ-notation)

1.2.3.1 Big - Oh Notation (O)

- Big - Oh notation is used to define the upper bound of an algorithm ie. it indicates the maximum time required by an algorithm for all input values. Therefore, it gives the worst-case complexity of an algorithm.
- Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_o, C > 0 and n_o >= 1. Then we can represent f(n) as O(g(n)).



f(n) = O(g(n))

Fig. Big - Oh Notation

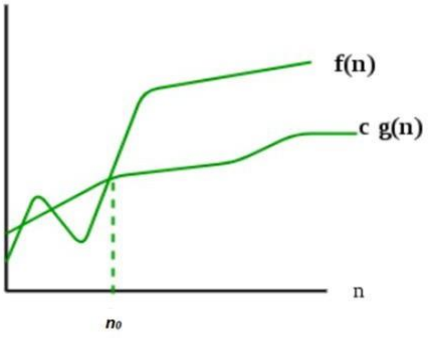
Examples:

- 100 , log (2000) , 10^4 --------> have O(1)
- (n/4) , (2n+3) , (n/100 + log(n)) ---> have O(n)
- (n^2+n) , (2 n^2) , (n^2+log(n)) ----> have O(n^2)
  O provides upper bounds.

1.2.3.2 Omega Notation (Ω-Notation)

- Omega notation represents the lower bound of an algorithm ie. it indicates the minimum time required by an algorithm for all input values. Thus, it provides the best case complexity of an algorithm.

- Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n_o, C > 0 and n_o >= 1. Then we can represent f(n) as Ω(g(n)).



f(n) = Ω(g(n))

APEC

Fig. Omega Notation

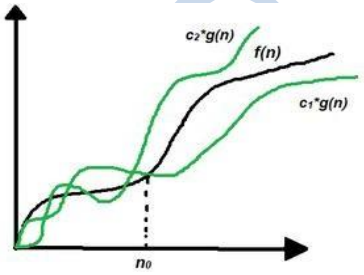
Examples:

- 100 , log (2000) , 10^4 --------> have Ω (1)
- (n/4) , (2n+3) , (n/100 + log(n)) ---> have Ω (n)
- (n^2+n) , (2 n^2) , (n^2+log(n)) ----> have Ω (n^2)
  Ω provides lower bounds.

1.2.3.3 Theta Notation (Θ-Notation)

- Theta notation always indicates the average time required by an algorithm. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

- Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).



f(n) = Θ(g(n))

Fig. Theta Notation

Examples:

- 100 , log (2000) , 10^4 --------> have Θ (1)
- (n/4) , (2n+3) , (n/100 + log(n)) ---> have Θ (n)
- (n^2+n) , (2 n^2) , (n^2+log(n)) ----> have Θ (n^2)
  Θ provides exact bounds.

Asymptotic Notation
f(n) = O(g(n)) Big - Oh Notation ---- Upper Bound ====> Worst case
f(n) = Ω(g(n)))) Omega Notation---- Lower Bound ====> Best case
f(n) = Θ(g(n)) Theta Notation ----Upper & Lower Bound ====> Average case

1.2.4 Worst Case, Average Case, and Best Case in Algorithm Analysis

- Best case - Function which performs the minimum number of steps on input data of size n.
- Worst case - Function which performs the maximum number of steps on input data of size n.
- Average case - Function which performs an average number of steps on input data of size n.

Best Case Analysis (Very Rarely used):

- In the best case analysis, calculate the lower bound of the execution time of an algorithm. It is necessary to know the case which causes the execution of the minimum number of operations.
- Example – Linear Search
  In linear search, Best case occurs when x is present at the first location. The best case time complexity would be Ω(1)

Worst Case Analysis (Mostly used):

- In the worst case analysis, calculate the upper limit of the execution time of an algorithm. It is necessary to know which causes the execution of the maximum number of operations. The worst-case time complexity of the linear search would be O(n).
- Example – Linear Search
  In linear search, Worst case occurs when x is NOT present in the array. The worst case time complexity of the linear search would be O(n).

Average Case Analysis (Rarely used):

- In average case analysis, take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.
- Example:
  In linear search, assume all cases are uniformly distributed (including the case of x not being present in the array). After summing all the cases, divide the sum by (n+1).

Types of time complexities:

Big O Notation	Name	Example(s)
O(1)	Constant	1. Odd or Even number checking 2. Look-up table (on average)
O(n)	Linear	1. Find max element in unsorted array 2. Duplicate elements in array with Hash Map
$O(n^2)$	Quadratic	1. Duplicate elements in array 2. Bubble sort
O(log n)	Logarithmic	Binary Searching
O(n log n)	Linearithmic	Merge Sort
$O(2^n)$	Exponential	1. Travelling salesman problem using dynamic programming 2. Fibonacci series generation

<u>O(1) - Constant time</u>

O(1) describes algorithms that take the same amount of time to compute regardless of the input size. For example, if a function takes the same time to process ten elements and 1 million items, then it is O(1).

Examples:

- Find if a number is even or odd.

- Check if an item on an array is null.
- Print the first element from a list.
- Find a value on a map.

O(n) - Linear time

Linear time complexity *O(n)* means that the algorithms take proportionally longer to complete as the input grows. These algorithms imply that the program visits every element from the input.

Examples

- Get the max/min value in an array.
- Find a given element in a collection.
- Print all the values in a list.

$O(n^2)$ - Quadratic time

A function with a quadratic time complexity has a growth rate of n^2. If the input is size 2, it will do four operations. If the input is size 8, it will take 64, and so on.

Examples

- Check if a collection has duplicated values.
- Sorting using bubble sort, insertion sort, or selection sort.
- Find all possible ordered pairs in an array.

O(log n) - Logarithmic time

Logarithmic time complexities usually apply to algorithms that divide problems in half every time. For example, to find a word in a book which is sorted alphabetically, there are two ways to do it.

Method 1:

- Start on the first page of the book and go word by word until you find matching word.

Method 2:

- Open the book in the middle and check the first word on it.
- If the word you are looking for is alphabetically more significant, then look to the right. Otherwise, look in the left half.
- Divide the remainder in half again, and repeat above step until you find matching.

Method 1 - go word by word - *O(n)*

Method 2 - split the problem in half for each iteration - *O(log n)*

Example

- Binary search.

O(n log n) - Linearithmic

Linearithmic time complexity it's slightly slower than a linear algorithm. However, it's still much better than a quadratic algorithm.

Examples

- Sorting algorithms like merge sort, quicksort, and others.

$O(2^n)$ - Exponential time

Exponential (base 2) running time means the calculations performed by an algorithm double every time as the input grows.

Examples:

- Fibonacci series generation
- Travelling salesman problem using dynamic programming

2. Recurrence Relation

A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s). It helps in finding the subsequent term (next term) with the previous term. If we know the previous term in a given series, then we can easily determine the next term.

Example 1:

- Recursive definition for the factorial function

```
n!=(n-1)! * n
```

Example 2:

- Recursive definition for Fibonacci sequence

```
Fib(n)=Fib(n-1)+Fib(n-2)
```

Recurrence relations are often used to model the cost of recursive functions. For example, the number of multiplications required by a recursive version of the factorial function for an input of size n will be zero when n=0 or n=1 (the base cases), and it will be one plus the cost of calling `fact` on a value of n–1.

2.1 Expansion of the recurrence equations

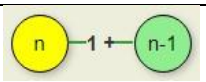
Example 1:

Let us see the expansion of the following recurrence equation.

```
T(n)= T(n-1)+1 for n>1
T(0)= T(1)=0.
```

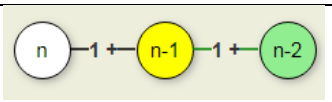
Step 1:

```
T(n) = 1 + T(n - 1)
```



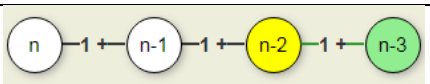
Step 2:

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



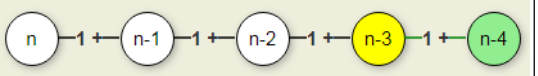
Step 3:

```
T(n) = 1 + (1 + (1 + T(n - 3)))
```



Step 4:

```
T(n) = 1 + (1 + (1 + (1 + T(n - 4))))
```



Step 5:

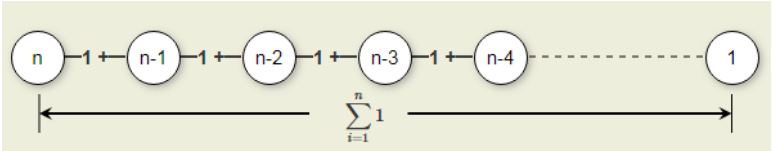
This pattern will continue till we reach a sub-problem of size 1.

```
T(n) = 1 + (1 + (1 + (1 + (1 + (1 + .....))))
```



Step 6:

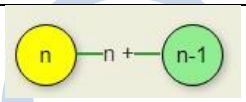
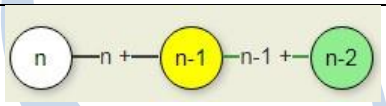
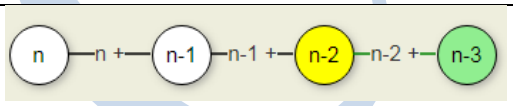
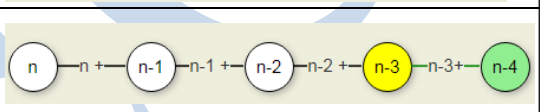
Thus the closed form of `T(n) = 1 + T(n - 1)` can be modeled as $\sum_{i=1}^{n} 1$



Example 2:

Let us see the expansion of the following recurrence equation.

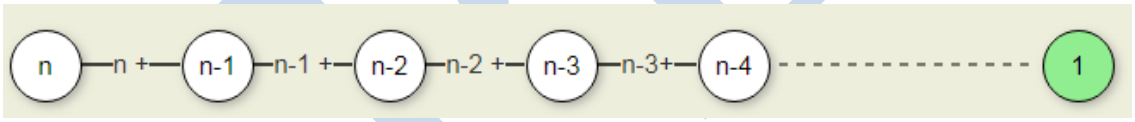
```
T(n)= T(n-1)+ n
T(1)= 1
```

Step 1: `T(n) = n + T(n - 1)`	
Step 2: `T(n) = n +(n - 1 + T(n - 2))`	
Step 3: `T(n) = n + (n - 1 + (n - 2 + T(n - 3))`	
Step 4: `T(n) = n +(n - 1+(n - 2 +(n -3 +T(n - 4))))`	

Step 5:

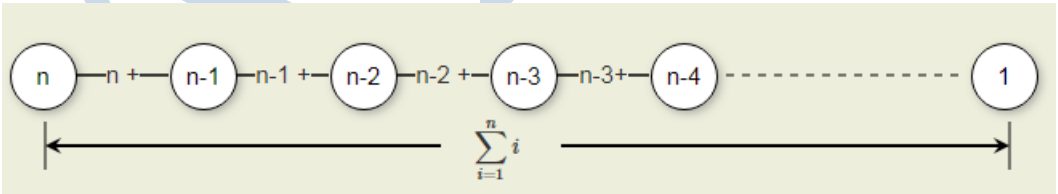
This pattern will continue till we reach a sub-problem of size 1.

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



Step 6:

Thus the closed form of `T(n) = n + T(n - 1)` can be modeled as $\sum_{i=1}^{n} 1$



2.2 Methods for solving Recurrence

- Substitution Method
- Iteration Method
- Recursion Tree Method
- Master Method

2.2.1 *Substitution Method*

In the substitution method, we have a known recurrence, and we use induction to prove that our guess is a good bound for the recurrence's solution.

Steps

- Guess a solution through your experience.
- Use induction to prove that the guess is an upper bound solution for the given recurrence relation.

Example:

$$T(n) = 1 \quad \text{if } n=1$$
$$= 2T(n-1) \quad \text{if } n>1$$

$$T(n) = 2T(n-1)$$
$$= 2[2T(n-2)] = 2^2T(n-2)$$
$$= 4[2T(n-3)] = 2^3T(n-3)$$
$$= 8[2T(n-4)] = 2^4T(n-4)$$

Repeat the procedure for i times

$$T(n) = 2^i\ T(n-i)$$

Put $n-i=1$ or $i= n-1$ in (Eq.1)

$$T(n) = 2^{n-1}\ T(1)$$
$$= 2^{n-1}\ .1 \quad \{T(1) = 1 \ldots\ \text{given}\}$$
$$= 2^{n-1}$$

3. Searching

Searching is a technique that helps to find whether the given element is present in the set of elements. 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 standards searching techniques are:

- Linear Search or Sequential Search
- Binary Search
- Interpolation Search

3.1 Linear Search:

It is one of the most simple and straightforward search algorithms. In this, you need to traverse the entire list and compare the current element with the target element. If a match is found, you can stop the search else continue.

Linear search is implemented using following steps...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the first element in the array.

Step 3: If both are matched, then display "Given element found!!!" and terminate the program

Step 4: If both are not matched, then compare search element with the next element in the array.

Step 5: Repeat steps 3 and 4 until the search element is compared with the last element in the array.

Step 6: If the last element in the array is also not matched, then display "Element not found!!!" and terminate the function.

APEC

/* 3.1.1 Python Program to search the given element in the list of items using Linear Search */

Example

Given a array, Search a given element in array.

Case 1:

Input: Search 20.

12	5	10	15	31	20	25	2	40
0	1	2	3	4	5	6	7	8

Output: True (20 is present in array)

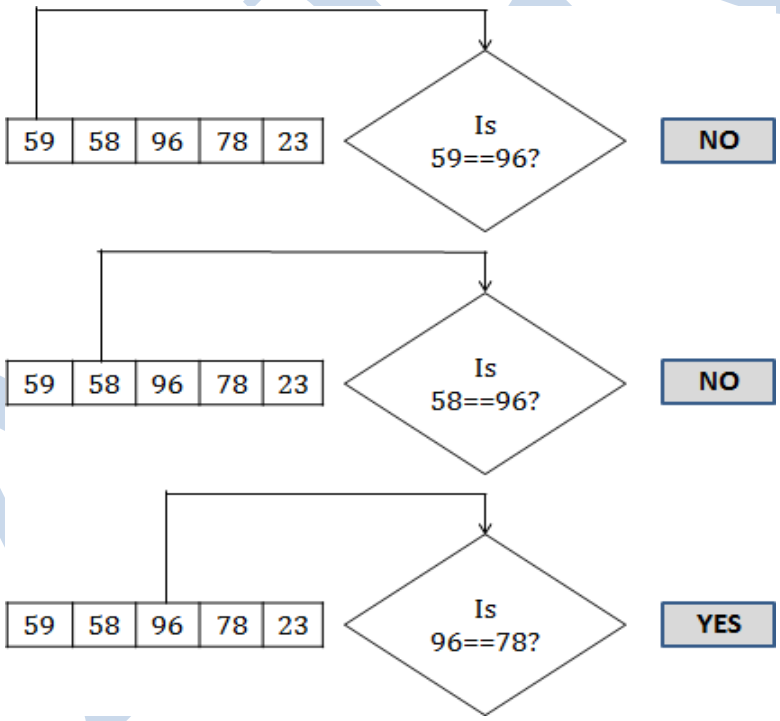
Case 2:

Input: Search 26.

12	5	10	15	31	20	25	2	40
0	1	2	3	4	5	6	7	8

Output: False (26 is not present in array)

Given the array of elements: 59,58,96,78,23 and the element to be searched is 96, the working of linear search is as follows:



The Element is FOUND. Hence stop the searching process.

```
def LinearSearch(mylist, n, k):
    for j in range(0, n):
        if (mylist[j] == k):
            return j
    return -1
```

```
mylist = [1, 3, 5, 7, 9]
print("Given Elements : ", mylist)
k = int(input("Enter the element to be searched : "))
n = len(mylist)
result = LinearSearch(mylist, n, k)
if(result == -1):
    print("Element not found")
else:
    print("Element found at index: ", result)

Execution:
Input
Given Elements :  [1, 3, 5, 7, 9]
Enter the element to be searched : 3
Output
Element found at index:  1
```

3.1.2 Complexity Analysis of Linear Search

Time Complexity

- *Best case - O(1)*
  The best case occurs when the target element is found at the beginning of the list/array. Since only one comparison is made, the time complexity is O(1).
  - Example :
    Array A[] = {3,4,0,9,8} & Target element = 3
    Here, the target is found at A[0].

- *Worst-case - O(n)*, where n is the size of the list/array.
  The worst-case occurs when the target element is found at the end of the list or is not present in the list/array. Since you need to traverse the entire list, the time complexity is O(n), as n comparisons are needed.

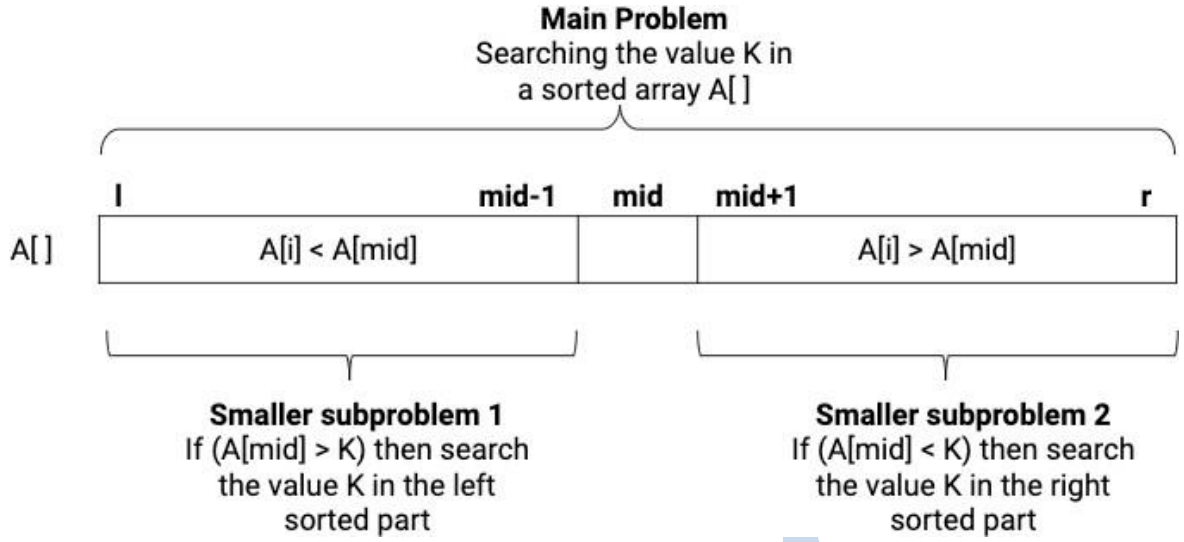
- *Average case - O(n)*
  The average case complexity of the linear search is also O(n).

Space Complexity

- The space complexity of the linear search is O(1), as we don't need any auxiliary space for the algorithm.

3.2 Binary search

Binary search is a searching algorithm which works efficiently on sorted elements. It uses divide and conquers method in which we compare the target element with the middle element of the list. If they are equal, then it implies that the target is found at the middle position; else, we reduce the search space by half, i.e. apply binary search on either of the left and right halves of the list depending upon whether target<middle element or target>middle element. We continue this until a match is found or the size of the array reaches 1.



Binary search is implemented using following steps:

Step 1: Read the search element from the user

Step 2: Find the middle element in the sorted array

Step 3: Compare, the search element with the middle element in the sorted array.

Step 4: If both are matched, then display "Given element found!!!" and terminate the function

Step 5: If both are not matched, then check whether the search element is smaller or larger than middle element.

Step 6: If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sub array of the middle element.

Step 7: If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sub array of the middle element.

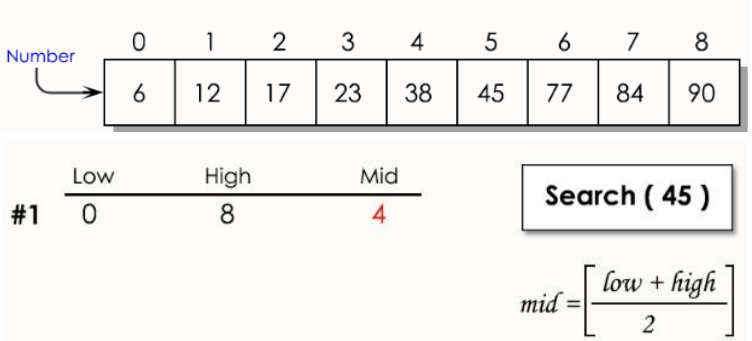
Step 8: Repeat the same process until we find the search element in the array or until the sub array contains only one element.

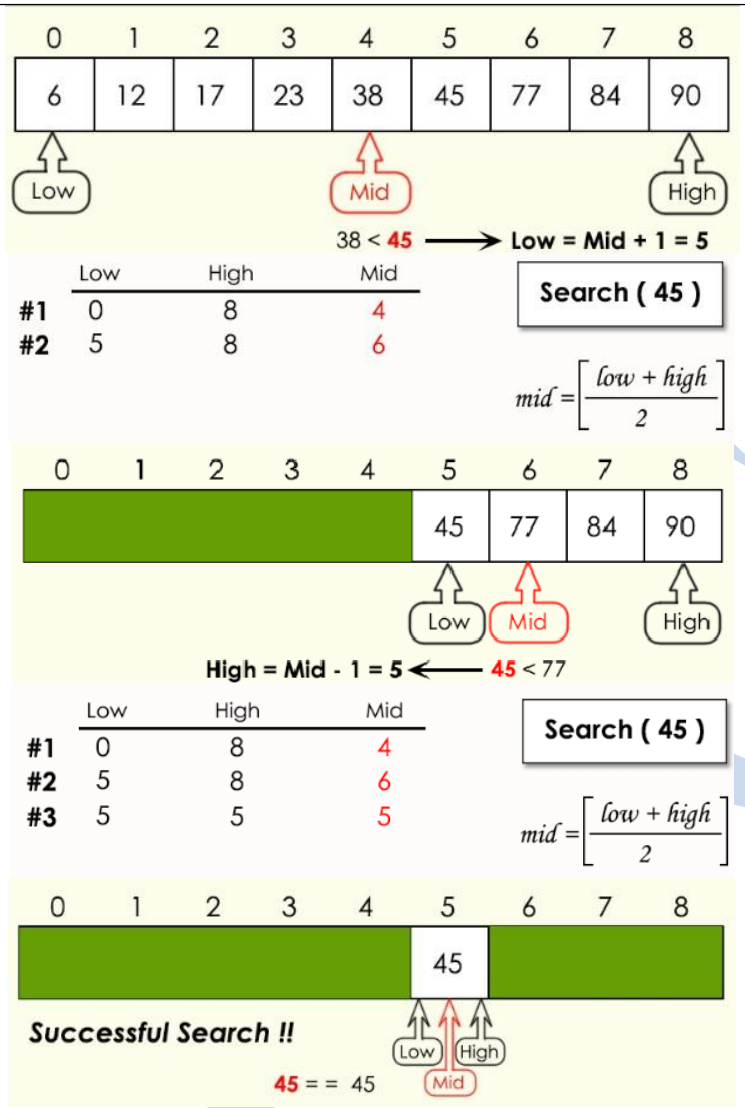
Step 9: If that element also doesn't match with the search element, then display "Element not found in the array!!!" and terminate the function.

/* 3.2.1 Python Program to search the given element in the list of items using Binary Search using Iterative approach */

Method 1 – Iterative approach:

```
Given an array of elements: 6, 12, 17, 323, 38, 45, 77, 84, 90
The element to be searched: 45
Formula for calculating middle is, Mid = (start +end)/2
```





The Element is FOUND. Hence stop the searching process.

```
def mybinarySearch(myarray, x, low, high):
    # Binary Search using Iterative approach
    while low <= high:
        mid = low + (high - low)//2
        if myarray[mid] == x:
            return mid
        elif myarray[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1


myarray = [3, 4, 5, 6, 7, 8, 9]
print("Elements in the array: " , myarray)
x = int(input("Enter the element to be searched : "))
```

```
result = mybinarySearch(myarray, x, 0, len(myarray)-1)

if result != -1:
    print("Element is present at index :" + str(result))
else:
    print("Element not found ")
```

Execution:

Input

```
Elements in the array:  [33, 44, 55, 66, 77, 88, 99]
Enter the element to be searched : 66
```

Output

```
Element is present at index :3
```

/* 3.2.2 Python Program to search the given element in the list of items using Binary Search using Recursive approach */

Method 2 – Recursive approach:

Method 2 is the recursive approach. In the recursive approach the function calls itself again and again. We declared a recursive function and its base condition. The condition is the lowest value is smaller or equal to the highest value. We calculate the middle number as in the last program.

We have used *if statement* to proceed with the binary search.

- If the middle value equal to the number that we are looking for, the middle value is returned.
- If the middle value is less than the value, we are looking then our recursive function binary_search() again and increase the mid value by one and assign to low.
- If the middle value is greater than the value we are looking then our recursive function binary_search() again and decrease the mid value by one and assign it to low.

Program:

```
def mybinary_search(myarr, low, high, x):
    if high >= low:
        mid = (high + low) // 2
        if myarr[mid] == x:
            return mid
        # If element is smaller than mid, then it can only
        # be present in left subarray
        elif myarr[mid] > x:
            return mybinary_search(myarr, low, mid - 1, x)
        # Else the element can only be present in right subarray
        else:
            return mybinary_search(myarr, mid + 1, high, x)
    else:
        # Element is not present in the array
        return -1
```

```
# Test data
myarr = [ 2, 3, 4, 10, 40 ]
print("Elements in the array :", myarr)
x = int(input("Enter the element to be searched : "))

# Function call
result = mybinary_search(myarr, 0, len(myarr)-1, x)

if result != -1:
    print("Element is present at index : ", str(result))
else:
    print("Element is not present in array")

Execution:
Input
Elements in the array : [2, 3, 4, 10, 40]
Enter the element to be searched : 10
Output
Element is present at index :  3
```

3.2.3 Complexity Analysis of Binary Search

Time Complexity

- *Best case - O(1)*
  The best case occurs when the target element is found in the middle of list/array. Since only one comparison is made, the time complexity is O(1).

- *Worst-case - O(logn)*
  The worst occurs when the algorithm keeps on searching for the target element until the size of the array reduces to 1. Since the number of comparisons required is logn, the time complexity is O(logn).

- *Average case - O(logn)*
  Binary search has an average-case complexity of O(logn).

Space Complexity

- Since no extra space is needed, the space complexity of the binary search is O(1).

3.3 Interpolation Search

The interpolation search is basically an improved version of the binary search. This searching algorithm resembles the method by which one might search a telephone book for a name. It performs very efficiently when there are uniformly distributed elements in the sorted list. In a binary search, we always start searching from the middle of the list, whereas in the interpolation search we determine the starting position depending on the item to be searched. In the interpolation search algorithm, the starting search position is most likely to be the closest to the

start or end of the list depending on the search item. If the search item is near to the first element in the list, then the starting search position is likely to be near the start of the list.

Important points on Interpolation Search

- Interpolation search is an improvement over binary search.
- Binary Search always checks the value at middle index. But, interpolation search may check at different locations based on the value of element being searched.
- For interpolation search to work efficiently the array elements/data should be sorted and uniformly distributed.

Interpolation search is implemented using following steps:

Step 1: Let `A` - Array of elements, `e` - element to be searched, `pos` - current position

Step 2: Assign `start = 0` & `end = n-1`

Step 3: Calculate position (pos) to start searching by using formula:

$$pos = start + \left[\frac{(end - start)}{(A[end] - A[start])} * (e - A[start]) \right]$$

Step 4: If `A[pos] == e` , element found at index pos.

Step 5: Otherwise `if e > A[pos]` we make `start = pos + 1`

Step 6: Else if `e < A[pos]` we make `end = pos -1`

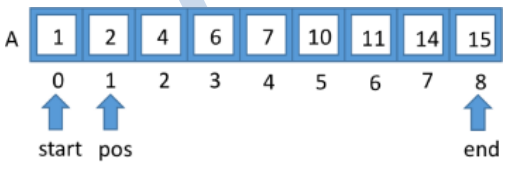
Step 7: Do steps 3, 4, 5, 6.

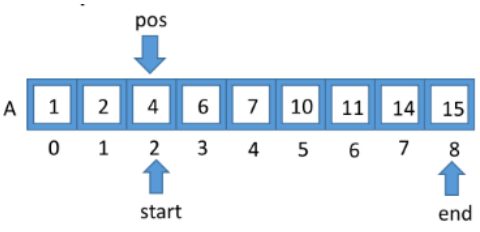
`While : start <= end && e >= A[start] && e =< A[end]`

- `start <= end` is checked until we have elements in the sub-array.
- `e >= A[start]` is done when the element we are looking for is greater than or equal to the starting element of sub-array we are looking in.
- `e =< A[end]` is done when the element we are looking for is less than or equal to the last element of sub-array we are looking in.

```
/* Python Program to search the given element in the list of
items using Interpolation Search */

Example: Element to be searched  = 4.
```





start	end	pos
0	8	0 + (8 – 0)/(15 – 1) * (4 – 1) 8/14 *3 = 0.57 * 3 = 1.71 = 1
2	8	2 + (8 – 2)/(15-4) * (4 -4) 2 + 6/11 * 0 = 2

Program

```
def interpolationSearch(arr, lo, hi, x):
    if (lo <= hi and x >= arr[lo] and x <= arr[hi]):
        pos = lo + ((hi-lo)//(arr[hi]-arr[lo])*(x - arr[lo]))
        if arr[pos] == x:
            return pos
        if arr[pos] < x:
            return interpolationSearch(arr, pos + 1, hi, x)
        if arr[pos] > x:
            return interpolationSearch(arr, lo, pos - 1, x)
    return -1

arr = [10, 12, 13, 16, 18, 19, 20,
       21, 22, 23, 24, 33, 35, 42, 47]
print("Elements in the array :", arr)
x = int(input("Enter the element to be searched : "))

n = len(arr)
index = interpolationSearch(arr, 0, n - 1, x)

if index != -1:
    print("Element found at index", index)
else:
    print("Element not found")
```

Execution:

Input

```
Elements in the array : [10, 12, 13, 16, 18, 19, 20, 21, 22,
23, 24, 33, 35, 42, 47]
Enter the element to be searched : 20
```

Output

```
Element found at index 6
```

Complexity Analysis of Interpolation Search

Time Complexity

- *Best case - O(1)*
  The best-case occurs when the target is found exactly as the first expected position computed using the formula. As we only perform one comparison, the time complexity is O(1).

- *Worst-case - O(n)*
  The worst case occurs when the given data set is exponentially distributed.

- *Average case - O(log(log(n)))*
  If the data set is sorted and uniformly distributed, then it takes O(log(log(n))) time as on an average (log(log(n))) comparisons are made.

Space Complexity

- Since no extra space is needed, the space complexity of the interpolation search is O(1).

3.4 Comparative analysis:

Algorithm	Time Complexity			Space Complexity
	Best case	Worst-case	Average case	
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(logn)	O(logn)	O(1)
Interpolation Search	O(1)	O(n)	O(log(log(n)))	O(1)

4. Pattern Search

The Pattern Searching algorithms are sometimes also referred to as String Searching Algorithms. These algorithms are useful in the case of searching a pattern in a string.

Algorithms used for String Matching:

Various string matching algorithms are:

- The Naive String Matching Algorithm
- The Rabin-Karp-Algorithm
- Finite Automata
- The Knuth-Morris-Pratt Algorithm
- The Boyer-Moore Algorithm

Algorithms based on character comparison

Naive Match Algorithm:

It slides the pattern over text one by one and checks for a match. If a match is found, then slides by 1 again to check for subsequent matches.

KMP (Knuth Morris Pratt) Algorithm:

KMP algorithm is used to find a "Pattern" in a "Text". This algorithm compares character by character from left to right. But whenever a mismatch occurs, it uses a pre-processed table called "Prefix Table" to skip characters comparison while matching.

Algorithms based on Hashing Technique

Rabin Karp Algorithm:

It matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters.

4. 1 Naive Match Algorithm:

This is simple and efficient brute force approach. It compares the first character of pattern with given stringt. If a match is found, pointers in both strings are advanced. If a match is not found,

the pointer to text is incremented and pointer of the pattern is reset. This process is repeated till the end of the text. The naïve approach does not require any pre-processing.

Given a text array, T [1.....n], of n character and a pattern array, P [1. m], of m characters. The algorithms are to find an integer s, called valid shift where 0 ≤ s < n-m. In other words, we need to find, if P is in T, i.e., where P is a substring of T. The item of P and T are character drawn from some finite alphabet such as {0, 1} or {A, B .....Z, a, b z}.

Steps:

```
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s
```

Input:

string = "This is my class room"
pattern = "class"

Output:

Pattern found at index 11

Input:

string = "AABAACAADAABAABA"
pattern = = "AABA"

Output:

Pattern found at index 0
Pattern found at index 9
Pattern found at index 12

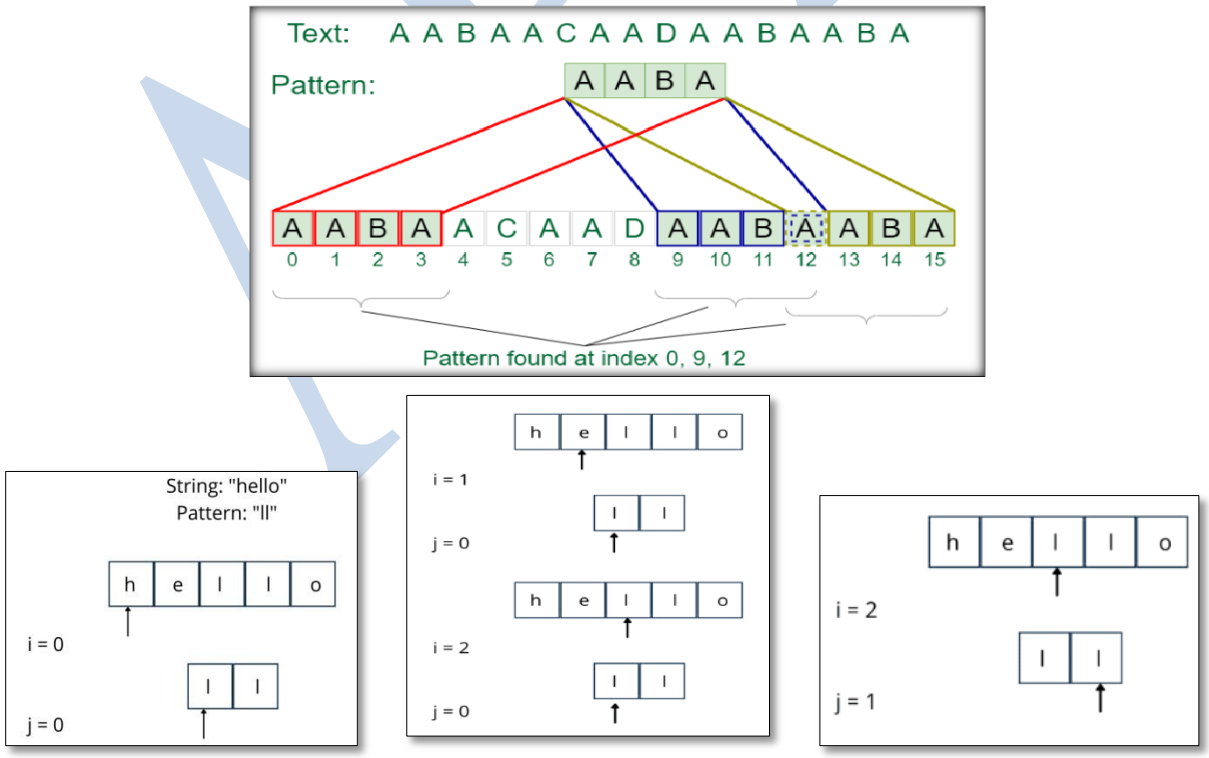


Fig Working of Naïve Pattern matching algorithm

```
/* 4.1.1 Python Program to search the pattern in the given string
using Naïve Match algorithm */

def naïve_algorithm(string, pattern):
    n = len(string)
    m = len(pattern)
    if m > n:
        print("Pattern not found")
        return
    for i in range(n - m + 1):
        j = 0
        while j < m:
            if string[i + j] != pattern[j]:
                break
            j += 1
        if j == m:
            print("Pattern found at index: ", i)

string = "hellohihello"
print("Given String : ", string)
pattern = input("Enter the pattern to be searched :")
naïve_algorithm(string, pattern)

Execution:
Input
Given String :  hellohihello
Enter the pattern to be searched :hi
Output
Pattern found at index:  5
```

4.1.2 Complexity Analysis of Naïve Match

Time Complexity

- *Best Case Complexity - O(n).*

  Best case complexity occurs when the first character of the pattern is not present in string.

  String = "HIHELLOHIHELLO"

  Pattern = " LI"

  The number of comparisons in best case is O(n).

- *Worst Case Complexity - O(m*(n-m+1)).*

  Worst case complexity of Naive Pattern Searching occurs in following cases.

  Case 1: When all the characters of the string and pattern are same.

  String = "HHHHHHHHHHHH"

  Pattern = " HHH"

  Case 2: When only the last character is different.

String = "HHHHHHHHHHHM"
Pattern = " HHM"

The number of comparisons in the worst case is $O(m*(n-m+1))$.

Space Complexity

- Since no extra space is needed, the space complexity of the naïve search is O(1).

4.1.3 Merits & Demerits:

Advantages:

- The comparison of the pattern with the given string can be done in any order
- No extra space required
- Since it doesn't require the pre-processing phase, as the running time is equal to matching time

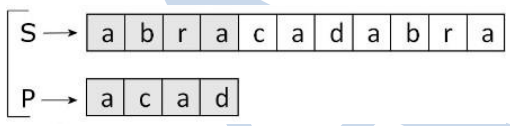
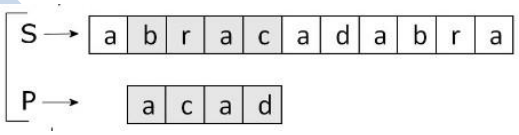
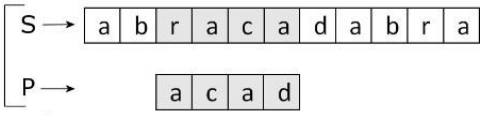
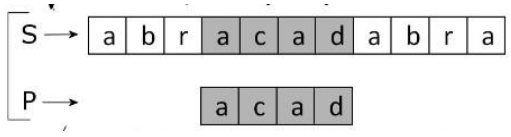
Disadvantage:

- Naive method is inefficient because information from a shift is not used again.

4. 2 Rabin Karp Algorithm:

Rabin-Karp algorithm is an algorithm used for searching/matching patterns in the text using a hash function. Unlike Naive string matching algorithm, it does not travel through every character in the initial phase rather it filters the characters that do not match and then performs the comparison.

- Initially calculate the hash value of the pattern.
- Start iterating from the starting of the string:
  - Calculate the hash value of the current substring having length m.
  - If the hash value of the current substring and the pattern are same, check if the substring is same as the pattern.
  - If they are same, store the starting index as a valid answer. Otherwise, continue for the next substrings.
- Return the starting indices as the required answer.

 Hash(acad) = 1466 Hash(abra) = 1493 Hash(acad) ≠ Hash(abra) *Hence, it is mismatch*	 Hash(acad) = 1466 Hash(brac) = 1533 Hash(acad) ≠ Hash(brac) *Hence, it is mismatch*
 Hash(acad) = 1466 Hash(raca) = 1595 Hash(acad) ≠ Hash(raca) *Hence, it is mismatch*	 Hash(acad) = 1466 Hash(acad) = 1466 Hash(acad) ≠ Hash(acad) *Match found at index 3*

Steps in Rabin-Karp Algorithm:

Step 1:

- Take the input string and the pattern, which we want to match.

Given string:										Pattern		
A	B	C	C	D	D	A	E	F	G	C	D	D

Step 2:

- Here, we have taken first ten alphabets only (i.e. A to J) and given the weights.

A	B	C	D	E	F	G	H	I	J
1	2	3	4	5	6	7	8	9	10

Step 3:

`n` → Length of the text

`m` → Length of the pattern

Here, `n` = 10 and `m` = 3.

`d` → Number of characters in the input set.

Here, we have taken input set {A, B, C, ..., J}. So, `d` = 10.

Note: we can assume any suitable value for d.

Step 4:

- Calculate the hash value of the pattern (CDD)

$$\text{hash value for pattern(p)} = \Sigma(v * d^{m-1}) \bmod 13$$
$$= ((3 * 10^2) + (4 * 10^1) + (4 * 10^0)) \bmod 13$$
$$= 344 \bmod 13$$
$$= 6$$

In the calculation above, choose a prime number (here, 13) in such a way that we can perform all the calculations with single-precision arithmetic.

- Now calculate the hash value for the first window (ABC)

$$\text{hash value for text(t)} = \Sigma(v * d^{m-1}) \bmod 13$$
$$= ((1 * 10^2) + (2 * 10^1) + (3 * 10^0)) \bmod 13$$
$$= 123 \bmod 13$$
$$= 6$$

- Compare the hash value of the pattern with the hash value of the text. If they match then, character-matching is performed. In the above examples, the hash value of the first window (i.e. text) matches with pattern, so go for character matching between ABC and CDD. Since they do not match so, go for the next window.

Step 5:

- We calculate the hash value of the next window by subtracting the first term and adding the next term as shown below.
- Simple Numerical example:
  - Pattern length is 3 and string is "23456"
  - Let us assume that we computed the value of the first window as 234.
  - How to compute the value of the next window "345"?
    It's just (234 – 2*100)*10 + 5 and we get 345.

hash value for text(t) = $((1 * 10^2) + ((2 * 10^1) + (3 * 10^0) - (1 * 10^2)) * 10 + (3 * 10^0)) \bmod 13$
$= 233 \bmod 13$
$= 12$

For BCC, t = 12 (≠6). Therefore, go for the next window.
After a few searches, we will get the match for the window CDA in the text.

```
/* 4.2.1 Python Program to search the pattern in the given string
using Rabin-Karp algorithm */

d = 10
def search(pattern, text, q):
    m = len(pattern)
    n = len(text)
    p = 0
    t = 0
    h = 1
    i = 0
    j = 0

    for i in range(m-1):
        h = (h*d) % q

    # Calculate hash value for pattern and text
    for i in range(m):
        p = (d*p + ord(pattern[i])) % q
        t = (d*t + ord(text[i])) % q

    # Find the match
    for i in range(n-m+1):
        if p == t:
            for j in range(m):
                if text[i+j] != pattern[j]:
                    break

            j += 1
            if j == m:
                print("Pattern is found at position: " + str(i+1))

        if i < n-m:
            t = (d*(t-ord(text[i])*h) + ord(text[i+m])) % q

            if t < 0:
                t = t+q

text = "hihellohi"
print("Given String : ", text)
pattern = input("Enter the pattern to be searched :")
q = int(input("Enter the prime number :"))

search(pattern, text, q)
```

```
Execution:
Input
Given String :  hihellohi
Enter the pattern to be searched :hello
Enter the prime number :3
Output
Pattern is found at position: 3
```

4.2.2 Complexity Analysis of Rabin-Karp algorithm

Time Complexity

- *Best Case Complexity - O(n+m).*
  The average and best-case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm).

- *Worst Case Complexity - O(nm).*
  The worst case of the Rabin-Karp algorithm occurs when all characters of pattern and text are the same as the hash values of all the substrings of text matches with the hash value of pattern.

Space Complexity

- Since no extra space is needed, the space complexity of the naïve search is O(1).

4.2.3 Merits & Demerits:

Advantages:

- Extends to 2D patterns.
- Extends to finding multiple patterns.

Disadvantage:

- Arithmetic operations is slower than character comparisons.

4.3 Knuth-Morris-Pratt Algorithm

KMP Algorithm is one of the most popular patterns matching algorithms. KMP stands for Knuth Morris Pratt algorithm. KMP algorithm was the first linear time complexity algorithm for string matching. KMP algorithm is used to find a "Pattern" in a "Text". This algorithm compares character by character from left to right. But whenever a mismatch occurs, it uses a pre-processed table called "Prefix Table" to skip characters comparison while matching. Sometimes prefix table is also known as LPS Table. Here LPS stands for "Longest proper Prefix which is also Suffix".

Steps for Creating LPS Table (Prefix Table)

Step 1: Define a one dimensional array with the size equal to the length of the Pattern. (LPS[size])
Step 2: Define variables i & j. Set i = 0, j = 1 and LPS[0] = 0.
Step 3: Compare the characters at Pattern[i] and Pattern[j].
Step 4: If both are matched then set LPS[j] = i+1 and increment both i & j values by one.
Goto Step 3.
Step 5: If both are not matched then check the value of variable 'i'. If it is '0' then set LPS[j] = 0 and increment 'j' value by one, if it is not '0' then set i = LPS[i-1]. Goto Step 3.
Step 6: Repeat above steps until all the values of LPS[] are filled.
Example:

Given Pattern	Initialize LPS[] table with size 7 which is equal to the length of the pattern
A B C D A B D	0 1 2 3 4 5 6 LPS

Step 1:

- Define variables i & j.
- Set i = 0, j= 1 and LPS[0] = 0.

	0	1	2	3	4	5	6
LPS	0						

Step 2:

- Compare Pattern[i] with Pattern[j] ====>A is compared with B. Since both were not matching, check the value of i.
- i = 0, so set LPS[j] = 0 and increment 'j' value by 1.

	0	1	2	3	4	5	6
LPS	0	0					

- Now, i = 0 & j = 2

Step 3:

- Compare Pattern[i] with Pattern[j] ====>A is compared with C. Since both were not matching, check the value of i.
- i = 0, so set LPS[j] = 0 and increment 'j' value by 1.

	0	1	2	3	4	5	6
LPS	0	0	0				

- Now, i = 0 & j = 3

Step 4:

- Compare Pattern[i] with Pattern[j] ====>A is compared with D. Since both were not matching, check the value of i.
- i = 0, so set LPS[j] = 0 and increment 'j' value by 1.

	0	1	2	3	4	5	6
LPS	0	0	0	0			

- Now, i = 0 & j = 4

Step 5:

- Compare Pattern[i] with Pattern[j] ====>A is compared with A. Since both are matching, set LPS[j] = i+1 and increment both 'i' & 'j' value by 1.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1		

- Now, i = 1 & j = 5

Step 6:

- Compare Pattern[i] with Pattern[j] ====>B is compared with B. Since both are matching, set LPS[j] = i+1 and increment both 'i' & 'j' value by 1.

0	1	2	3	4	5	6

APEC

LPS	0	0	0	0	1	2	

- Now, i = 2 & j = 6

Step 7:

- Compare Pattern[i] with Pattern[j] ====>C is compared with D. Since both were not matching, check the value of i.
- i !=0, so set i= LPS[i-1]====>i= LPS[2-1]
  - i= 0

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

- Now, i = 0 & j = 6

Step 7:

- Compare Pattern[i] with Pattern[j] ====>A is compared with D. Since both were not matching, check the value of i.
- i = 0, so set LPS[j] = 0 and increment 'j' value by 1.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

- Now, i = 0 & j = 7

Final LPS[] table is as follows:

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Working mechanismof KMP:

We use the LPS table to decide how many characters are to be skipped for comparison when a mismatch has occurred. When a mismatch occurs, check the LPS value of the previous character of the mismatched character in the pattern.

- If it is '0' then start comparing the first character of the pattern with the next character to the mismatched character in the text.
- If it is not '0' then start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in pattern with the mismatched character in the Text.

Example:

Consider the following Text and Pattern

Text : ABC ABCDAB ABCDABCDABDE

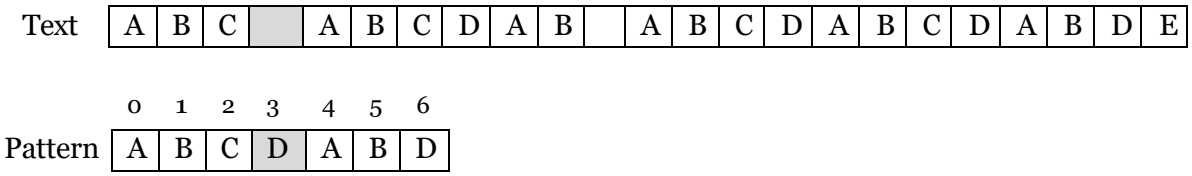
Pattern: ABCDABD

LPS[] table for the above pattern is as follows:

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Step 1:

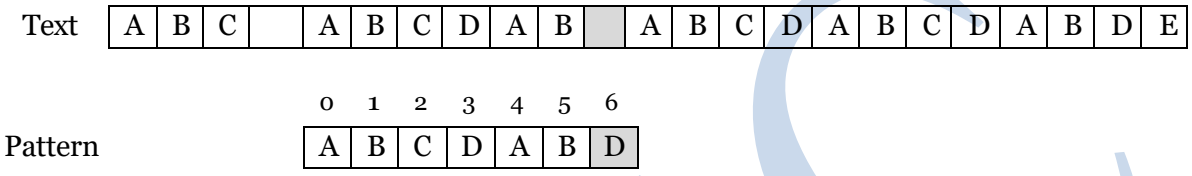
- Start comparing the first character of the pattern with the first character of Text from left to right.



- Here mismatch occurs at pattern[3], so we need to consider LPS[2] value is '0' we must compare first charater in pattern with next character in Text.

Step 2:

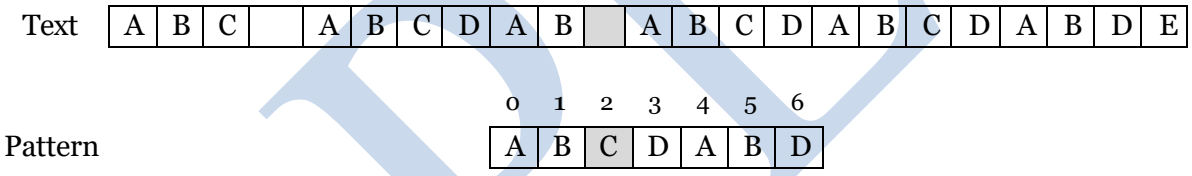
- Start comparing first charater in pattern with next character in Text.



- Here mismatch occurs at pattern[6], so we need to consider LPS[5] value. LPS[5]= 2, now we must compare first charater in pattern[2] with next character in Text.

Step 3:

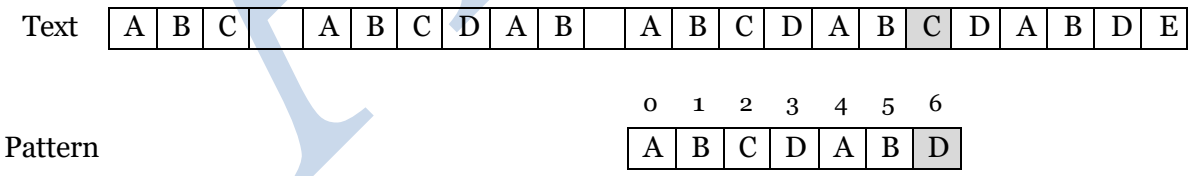
- Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values..



- Here mismatch occurs at pattern[2]. We need to consider LPS[2] value is '0'. Hence compare first charater in pattern with next character in Text.

Step 4:

- Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values..



- Here mismatch occurs at pattern[6]. We need to consider LPS[5] value. LPS[5]= 2, now we must compare first charater in pattern[2] with next character in Text.

Step 5:

- Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values. Compare pattern[2] with mismatched character in Text.

Text

A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E

Pattern

0	1	2	3	4	5	6
A	B	C	D	A	B	D

- Here all the characters of the pattern matched with the substring in the Text, which starts at index value 15. Hence, conclude that pattern found at index 15.

```
/* 4.3.1 Python Program to search the pattern in the given string
using Knuth-Morris-Pratt Algorithm */

def KMP_String(pattern, text):
    a = len(text)
    b = len(pattern)
    prefix_arr = get_prefix_arr(pattern, b)

    initial_point = []
    m = 0
    n = 0

    while m != a:

        if text[m] == pattern[n]:
            m += 1
            n += 1

        else:
            n = prefix_arr[n-1]

        if n == b:
            initial_point.append(m-n)
            n = prefix_arr[n-1]
        elif n == 0:
            m += 1

    return initial_point

def get_prefix_arr(pattern, b):
    prefix_arr = [0] * b
    n = 0
    m = 1
    while m != b:
        if pattern[m] == pattern[n]:
            n += 1
            prefix_arr[m] = n
            m += 1
        elif n != 0:
                n = prefix_arr[n-1]
        else:
            prefix_arr[m] = 0
            m += 1
    return prefix_arr
string = "hihellohihellohi"
```

APEC

```
print("Given String : ", string)
pat = input("Enter the pattern to be searched :")

initial_index = KMP_String(pat, string)
for i in initial_index:
    print('Pattern is found at index: ',i)


Execution:
Input
Given String : hihellohihellohi
Enter the pattern to be searched :hi
Output
Pattern is found at index: 0
Pattern is found at index: 7
Pattern is found at index: 14
```

4.3.2 Complexity Analysis of Knuth-Morris-Pratt Algorithm

Time Complexity

- Worst case complexity of KMP algorithm is $O(m+n)$.
  - $O(m)$ time is taken for LPS table creation.
  - Once this prefix suffix table is created, actual search complexity is $O(n)$.

Space Complexity

- Space complexity of KMP algorithm is $O(m)$ because some pre-processing work is involved.

4.3.3 Merits & Demerits:

Advantages:

- The running time of the KMP algorithm is O(m + n), which is very fast.
- The algorithm never needs to move backwards the input text T. It makes the algorithm good for processing very large files.

Disadvantage:

- Doesn't work so well as the size of the alphabets increases.

4.4 Comparative analysis:

Algorithm	Pre-processing the Pattern	Time Complexity	Space Complexity
Naive Match Algorithm	No pre-processing	$O(m*(n-m+1))$	$O(1)$
Rabin-Karp Algorithm	No pre-processing	$O(nm)$	$O(1)$
Knuth-Morris-Pratt Algorithm	Pre-process the pattern	$O(m + n)$	$O(m)$

5. Sorting

Sorting is the processing of arranging the data in ascending and descending order. There are several types of sorting in data structures namely,

- Bubble sort
- Insertion sort
- Selection sort
- Bucket sort
- Heap sort
- Quick sort
- Radix sort etc.

5.1 Insertion Sort

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



Fig. Insertion sort

Steps:

Step 1:

- The first element in the array is assumed to be sorted.

Step 2:

- Take the second element and store it separately in currentvalue. Compare currentvalue with the first element. If the first element is greater than currentvalue, then currentvalue is placed in front of the first element. Now, the first two elements are sorted.

Step 3:

- Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

Step 4:

- Similarly, place every unsorted element at its correct position. Repeat until list is sorted.

Working of Insertion Sort algorithm:

Example:

List = [12, 11, 13, 5, 6]

First Pass:

- Initially, the first two elements of the array are compared in insertion sort.

  12 11 13 5 6

- Here, 12 is greater than 11. They are not in the ascending order and 12 is not at its correct position. Hence, swap 11 and 12.
- So, for now 11 is stored in a sorted sub-array.

  11 12 13 5 6

Second Pass:

- Now, move to the next two elements and compare them

  11 12 13 5 6

- Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

Third Pass:

- Now, two elements are present in the sorted sub-array which are **11** and **12**
- Moving forward to the next two elements which are 13 and 5

  11 12 13 5 6

- Both 5 and 13 are not present at their correct place so swap them

  11 12 5 13 6

- After swapping, elements 12 and 5 are not sorted, thus swap again

  11 5 12 13 6

- Here, again 11 and 5 are not sorted, hence swap again

  5 11 12 13 6

Fourth Pass:

- Now, the elements which are present in the sorted sub-array are **5, 11** and **12**
- Moving to the next two elements 13 and 6

  5 11 12 13 6

- Clearly, they are not sorted, thus perform swap between both

  5 11 12 6 13

- Now, 6 is smaller than 12, hence, swap again

  5 11 6 12 13

- Here, also swapping makes 11 and 6 unsorted hence, swap again

  5 6 11 12 13

  Finally, the list is completely sorted.

/* 5.1.1 Python Program to sort the elements in the list using Insertion sort */

```
def insertionSort(arr):
   for index in range(1,len(arr)):

     currentvalue = arr[index]
     position = index

     while position>0 and arr[position-1]>currentvalue:
         arr[position]=arr[position-1]
```

```
            position = position-1

        arr[position]=currentvalue

arr = [54,26,93,17,77,91,31,44,55,20]
print("Given list : ", arr)
insertionSort(arr)
print("Sorted list : ",arr)

Execution:
Input
Given list :  [54, 26, 93, 17, 77, 91, 31, 44, 55, 20]
Output
Sorted list :  [17, 20, 26, 31, 44, 54, 55, 77, 91, 93]
```

5.1.2 Complexity Analysis of Insertion sort

Time Complexity

- *Best case complexity* - $O(n)$
  It occurs when there is no sorting required, i.e. the array is already sorted.
- *Worst case complexity* - $O(n^2)$
  It occurs when the array elements are required to be sorted in reverse order. It means suppose we need to sort the array elements in ascending order, but its elements are in descending order.
- *Average case complexity* - $O(n^2)$
  It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending.

Space Complexity

- Space complexity of insertion sort is $O(1)$

5.2 Heap sort

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements. Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

5.2.1 Heap

- A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

5.2.2 Relationship between Array Indexes and Tree Elements

- A complete binary tree has an interesting property that we can use to find the children and parents of any node.

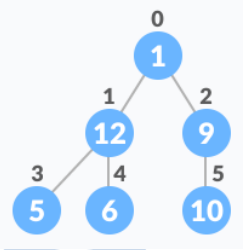
- If the index of any element in the array is i, the element in the index 2i+1 will become the left child and element in 2i+2 index will become the right child. Also, the parent of any element at index i is given by the lower bound of (i-1)/2.

Example:

Given array elements:

0	1	2	3	4	5
1	12	9	5	6	10

⇒ Array is converted to Heap



Steps to convert array elements to Heap	
Left child of 1 (index 0) = element in (2*0+1) index = element in 1 index = 12	Right child of 1 = element in (2*0+2) index = element in 2 index = 9
Left child of 12 (index 1) = element in (2*1+1) index = element in 3 index = 5	Right child of 12 = element in (2*1+2) index = element in 4 index = 6

Rules to find parent of any node	
Parent of 9 (position 2) = (2-1)/2 = ½ = 0.5 ~ 0 index = 1	Parent of 12 (position 1) = (1-1)/2 = 0 index = 1

5.2.3 Heap Data Structure

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

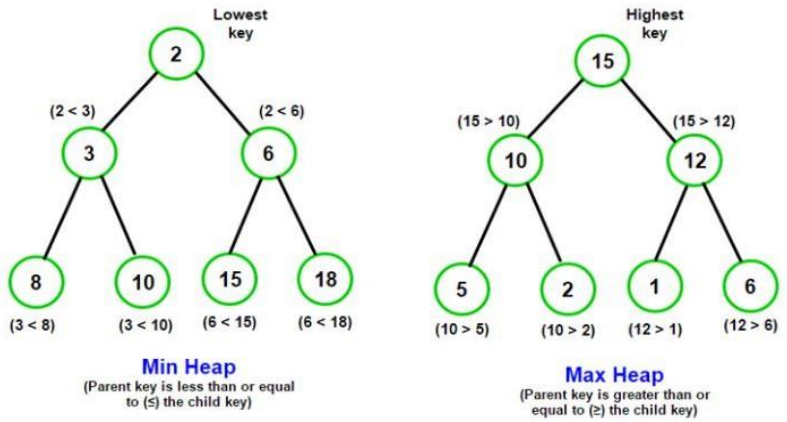
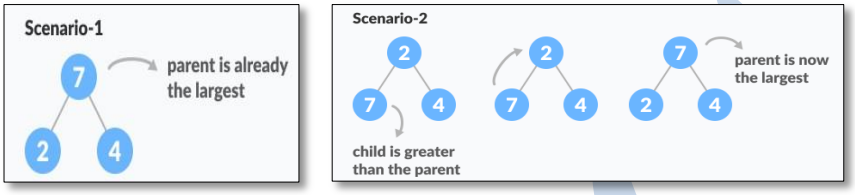


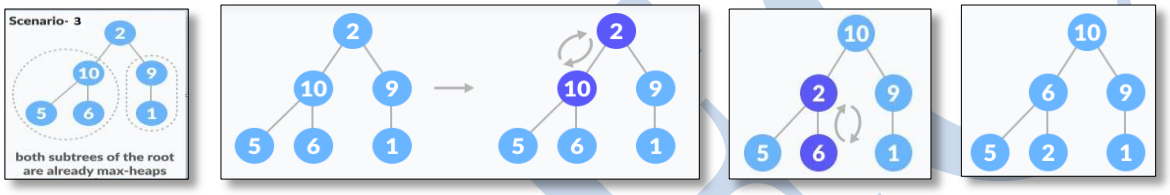
Fig Max Heap and Min Heap

5.2.4 "Heapify" process

- 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. Heapify process uses recursion.

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





- The top element isn't a max-heap but all the sub-trees are max-heaps. To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position.

Steps

Step 1: Construct a Binary Tree with given list of Elements.
Step 2: Transform the Binary Tree into Max Heap.
Step 3:
Since the tree satisfies Max-Heap property, then the largest item is stored at the root node. Three operations at each step are -

- *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.
- *Remove:* Reduce the size of the heap by 1.
- *Heapify:* Heapify the root element again so that we have the highest element at root.

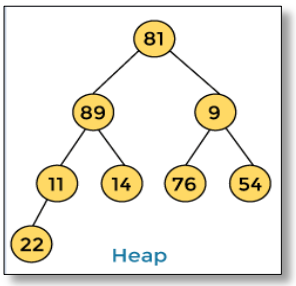
Step 4: Put the removed element into the Sorted list.
Step 5: Repeat the same until Max Heap becomes empty.
Step 6: Display the sorted list.

Working of Heap Sort Algorithm
Example:
Construct binary heap with the given list of elements

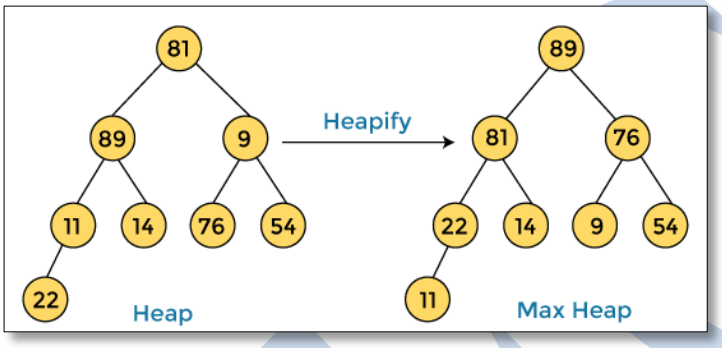
Given array elements:

0	1	2	3	4	5	6	7
81	89	9	11	14	76	54	22

⇒ Array is converted to Heap



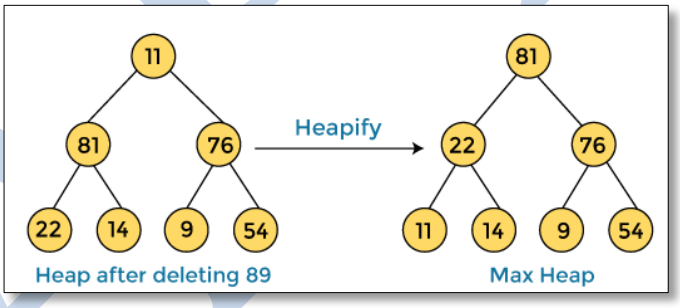
Convert the constructed heap to max heap using heapify algorithm



After converting the given heap into max heap, the array elements are -

0	1	2	3	4	5	6	7
89	81	76	22	14	9	54	11

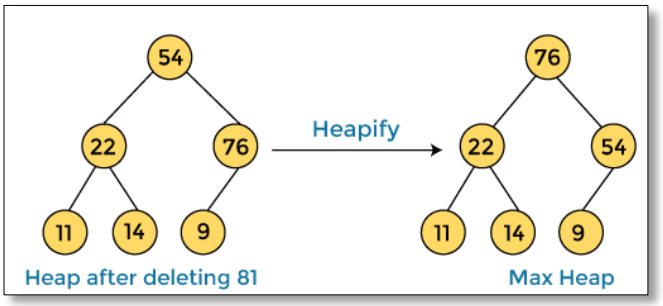
Next, we have to delete the root element (89) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 89 with 11, and converting the heap into max-heap, the elements of array are –

0	1	2	3	4	5	6	7
81	22	76	11	14	9	54	89

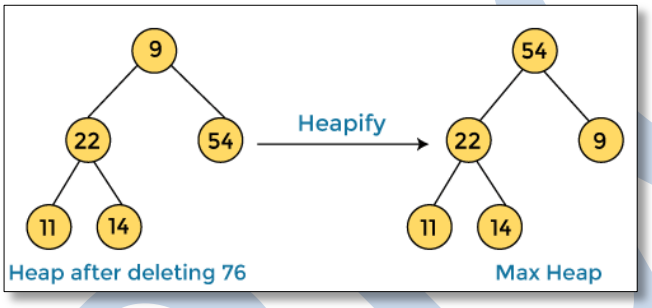
In the next step, again, we have to delete the root element (81) from the max heap. To delete this node, we have to swap it with the last node, i.e. (54). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 81 with 54 and converting the heap into max-heap, the elements of array are –

0	1	2	3	4	5	6	7
76	22	54	11	14	9	81	89

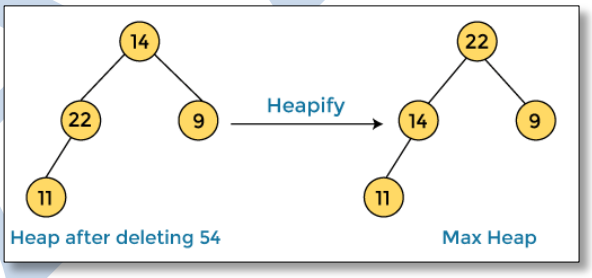
In the next step, we have to delete the root element (76) from the max heap again. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 76 with 9 and converting the heap into max-heap, the elements of array are –

0	1	2	3	4	5	6	7
54	22	9	11	14	76	81	89

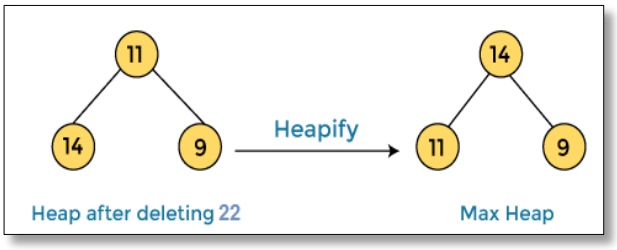
In the next step, again we have to delete the root element (54) from the max heap. To delete this node, we have to swap it with the last node, i.e. (14). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 54 with 14 and converting the heap into max-heap, the elements of array are –

0	1	2	3	4	5	6	7
22	14	9	11	54	76	81	89

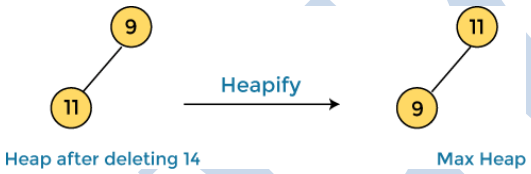
In the next step, again we have to delete the root element (22) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 22 with 11 and converting the heap into max-heap, the elements of array are –

0	1	2	3	4	5	6	7
14	11	9	22	54	76	81	89

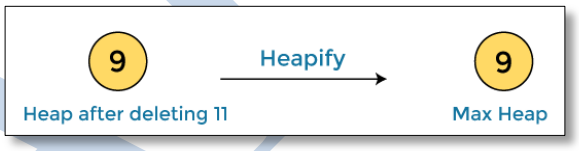
In the next step, again we have to delete the root element (14) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 14 with 9 and converting the heap into max-heap, the elements of array are –

0	1	2	3	4	5	6	7
11	9	14	22	54	76	81	89

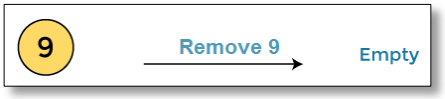
In the next step, again we have to delete the root element (11) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 11 with 9, the elements of array are –

0	1	2	3	4	5	6	7
9	11	14	22	54	76	81	89

Now, heap has only one element left. After deleting it, heap will be empty.



After completion of sorting, the array elements are –

0	1	2	3	4	5	6	7
9	11	14	22	54	76	81	89

Now, the array is completely sorted.

/* 5.2.5 Python Program to sort the elements in the list using Heap sort */

```
def heapify(array, a, b):
      largest = b
      l = 2 * b + 1
      root = 2 * b + 2

      if l < a and array[b] < array[l]:
            largest = l

      if root < a and array[largest] < array[root]:
            largest = root

      # Change root
      if largest != b:
          array[b], array[largest] = array[largest], array[b]
          heapify(array, a, largest)

# sort an array of given size
def Heap_Sort(array):
      a = len(array)

      # Building maxheap..
      for b in range(a // 2 - 1, -1, -1):
            heapify(array, a, b)

      # swap elements
      for b in range(a-1, 0, -1):
            array[b], array[0] = array[0], array[b]
            heapify(array, b, 0)


array = [81,89,9,11,14,76,54,22]
print("Original Array :", array)
Heap_Sort(array)
a = len(array)
print ("Sorted Array : ", array)
```

Execution:

Input

```
Original Array : [81, 89, 9, 11, 14, 76, 54, 22]
```

Output

```
Sorted Array :  [9, 11, 14, 22, 54, 76, 81, 89]
```

5.2.6 Complexity Analysis of Heap sort

Time Complexity

- *Best case complexity -* ***O(nlogn)***
  It occurs when there is no sorting required, i.e. the array is already sorted.
- *Worst case complexity -* ***O(nlogn)***
  It occurs when the array elements are required to be sorted in reverse order. It means suppose we need to sort the array elements in ascending order, but its elements are in descending order.
- *Average case complexity -* ***O(nlogn)***

It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending.

Space Complexity

- Space complexity of Heap sort is $O(1)$

5.4 Comparative analysis:

Algorithm	Time Complexity			Space Complexity
	Best case	Worst-case	Average case	
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Heap Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$

APEC