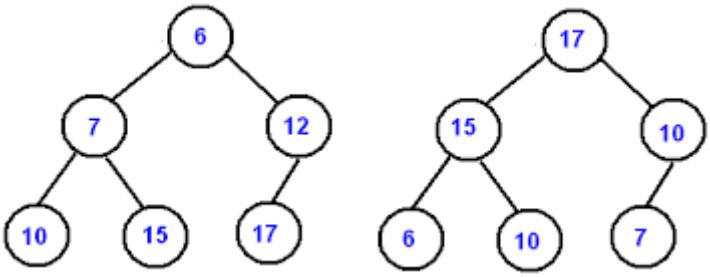
HEAP

A heap is a binary tree-based data structure.

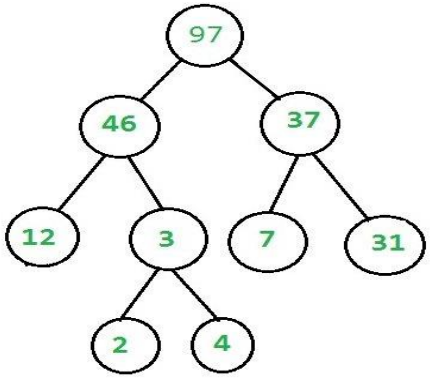
A heap data structure is a complete binary tree that satisfies the heap property,

Heap Property:

1. **Max-Heap**: The value of each node is always greater than or equal to the values of its children, ensuring that the root node contains the maximum value. As you move down the tree, the values decrease.
2. **Min-Heap**: The value of each node is always less than or equal to the values of its children, ensuring that the root node contains the minimum value. As you move down the tree, the values increase.



In a heap, all the tree levels are completely filled except possibly for the lowest level which is filled from left to right.



The above tree is not a heap because it is not a complete binary tree.

Example:

What are the minimum and maximum numbers of elements in heap of height h?

Solution:

Minimum number of nodes happens in a heap in which the last level contains only one node.

Thus, minimum no. of nodes $= 2^0 + 2^1 + \cdots + 2^{h-1} + 1 = 2^{h-1+1} - 1 + 1 = 2^h$

Maximum number of nodes happens in a heap in which the last level is full

Thus, Maximum no. of nodes $= 2^0 + 2^1 + \cdots + 2^{h-1} + 2^h = 2^{h+1} - 1$

Example:

Show that an n-element heap has height $\lfloor \log n \rfloor$

Solution:

Since, the minimum and maximum numbers of elements in heap of height h are 2^h and $2^{h+1} - 1$ respectively, we have:

$$2^h \leq n \leq 2^{h+1} - 1$$

Taking logarithm we get:

$$\log 2^h \leq \log n \leq \log 2^{h+1} - 1$$

$$\Rightarrow\ h \leq \log n \leq \log 2^{h+1}$$

$$\Rightarrow\ h \leq \log n \leq h + 1$$

Hence, $h = \lfloor \log n \rfloor$

Array reprsentation o f a binafry heap:

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as array. The representation is done as:

If the idex of the root element is 1 an i is the index of an element in the array then:

1. Index of the lift child is $2i$
2. Index of the right child is $2i + 1$ and
3. Inex of the parent is $\lfloor i/2 \rfloor$

If the idex of the root element is 0 an i is the index of an element in the array then:

1. Index of the lift child is $2i + 1$
2. Index of the right child is $2i + 2$ and
3. Inex of the parent is $\lfloor i - 1/2 \rfloor$

Heapify:

The process in which the binary tree is reshaped into a Heap data structure is known as heapify.

Maintaining the Heap property:

The heap property can be maintained by using the procedure Max-Heapify or, Min-Heapify.

Max-Heapify Procedure:

Max-heapify is a process of arranging the nodes in correct order so that they follow max-heap property. It has two inputs: Array A and Index i into the array. MAX-HEAPIFY lets the value at A[i] float down in the max-heap so that the subtree rooted at index i obeys the max-heap property.

Working Procedure:

1. Set current element i as largest.
2. If left child is greater than current element (i.e. element at i^{th} index), then set left child index as largest.
3. If right child is greater than element in largest, then set right child index as largest.
4. Swap largest with current element.
5. Repeat the same procedure until the subtree is heapified.

Algorithm:

MAX-HEAPIFY (A, i)

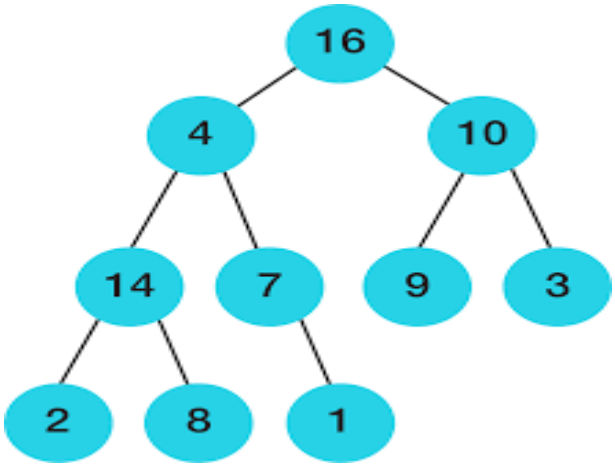
l← left[i]
r←right[i]
if $l \leq heap - size[A]$ *and* $A[l] > A[i]$ *then*
    $largest \leftarrow l$
else $largest \leftarrow i$
if $r \leq heap - size[A]$ *and* $A[r] > A[largest]$ *then*
    $largest \leftarrow r$
if $largest \neq i$ *then*
    exchange $A[i] \leftrightarrow A[largest]$
    MAX-HEAPIFY $(A, largest)$

Example:
Suppose we have an array of elements: $A = \{16, 4, 10, 14, 7, 9, 3, 2, 8, 1\}$ which form the following complete binary tree.

$$heap - size[A] = \text{size of the array} = 10.$$

In this tree the root node 16 satisfies max-heap property. But the parent node 4 violates max- heap property.

Since index of 4 is 2, $i = 2$.
So we call *MAX-HEAPIFY (A, 2)*



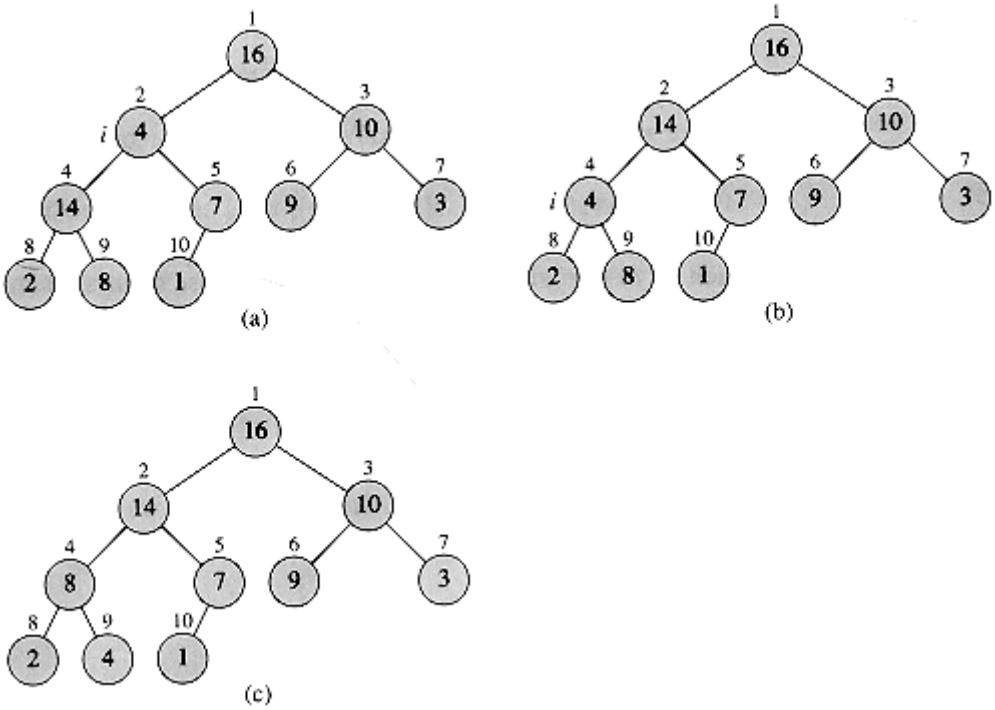
$i = 2,\ \ l = \ left[i] = left[2] = 4, \qquad r = right[2] = 5$

$l \leq \ heap - size[A]$ and $A[l] > A[2] \quad \Longrightarrow largest = l = 4$

Then $r \ \leq \ heap - size[A]\ and\ A[r] \not> A[largest], largest \ = 4$

Since, $largest \neq i$, $exchange\ A[i] \leftrightarrow A[largest]$

exchange $4 \leftrightarrow 14$



Now *MAX-HEAPIFY* $(A, largest\ )$i.e. *MAX-HEAPIFY* $(A, 4\ )$

Time Complexity:

The time complexity of Heapify algorithm is equal to $O(height\ of\ the\ complete\ binary\ tree)$ i.e $O(log\ n)$.

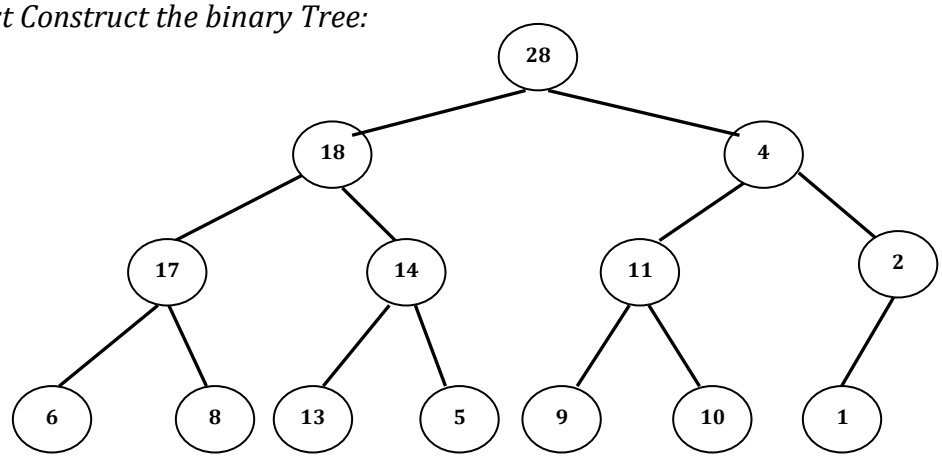
Example:

Illustrate the operation of *MAX-HEAPIFY (A, 3)* on the array:

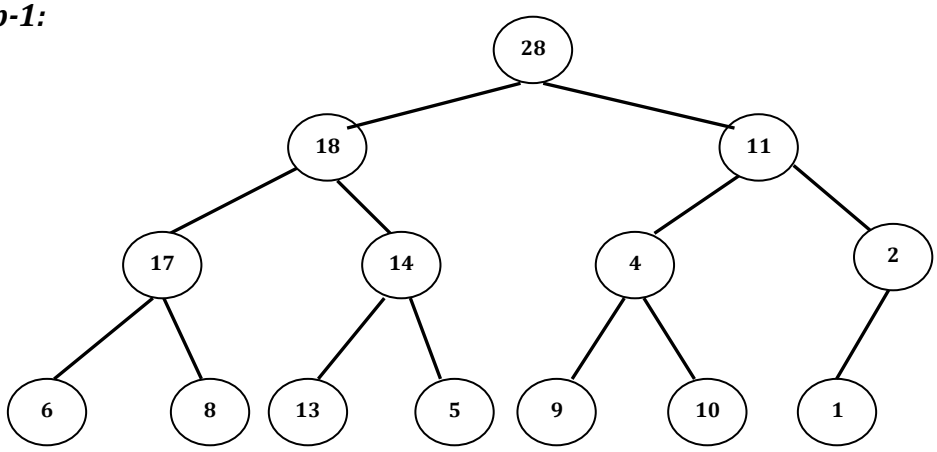
$$A = \{28, 18, 4, 17, 14, 11, 2, 6, 8, 13, 5, 9, 10, 1\}$$

Solution:

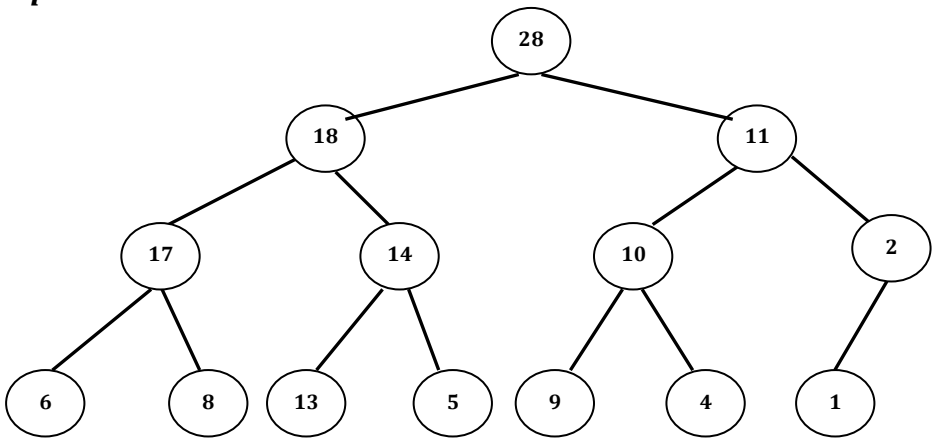
First Construct the binary Tree:



Step-1:



Step-2:



Building a Heap:

Given an array of n elements, the task is to build a Binary Heap (Max-heap/Min-heap) from the given array.

Working Procedure:

1. Create a complete binary tree from the array.
2. Starting from the first index of a non-leaf node to the index of the root apply Max heapify procedure recursively.

Algorithm:

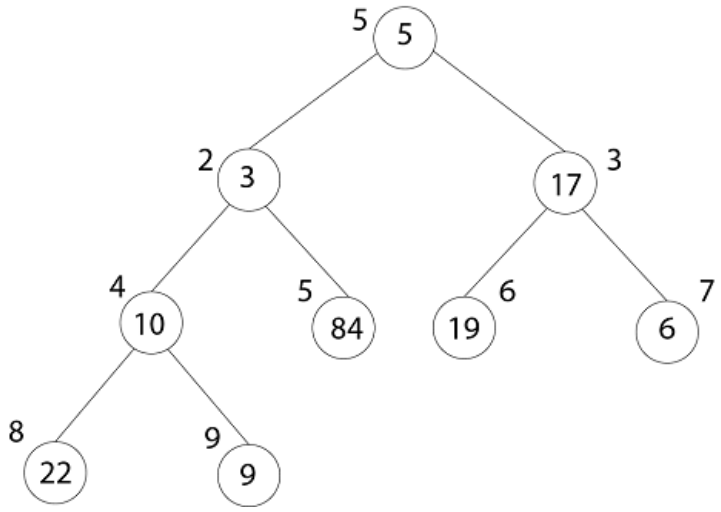
BUILD-MAX-HEAP(A)
heap_size(A) ← *length(A)*
for $(i = \lfloor length(A)/2 \rfloor$ *down to 1 do*
MAX-HEAPIFY(A, i)

Example:
Illustrate the operation of $BUILD - MAX - HEAP$ on the array:

$$A = \{5, 3, 17, 22, 84, 19, 6, 10, 9\}$$

Solution:

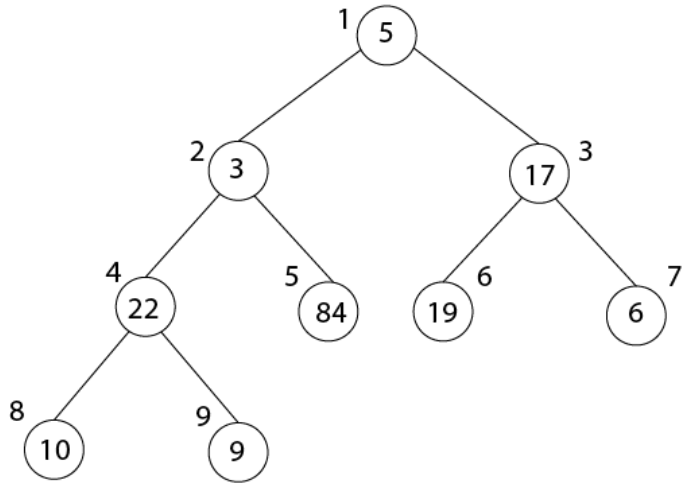
Construct a complete binary tree as follows:



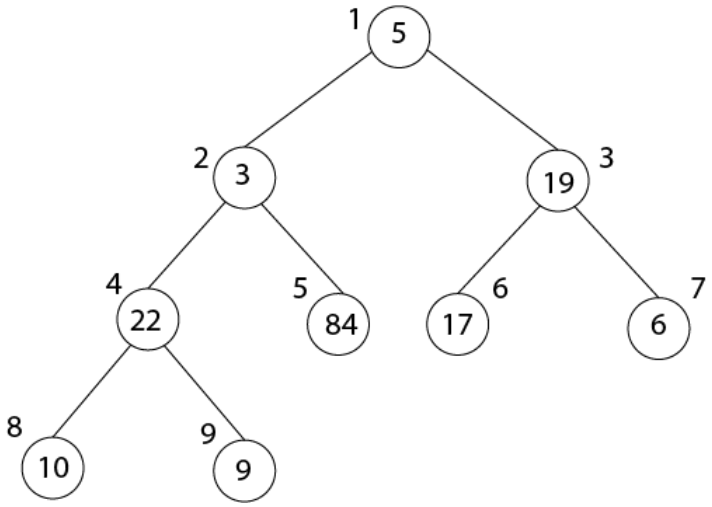
$Heap - size\ (A) = length[A] = 9, i = \left\lfloor \frac{9}{2} \right\rfloor = 4$

So first we call the procedure *MAX-HEAPIFY (A, 4)*

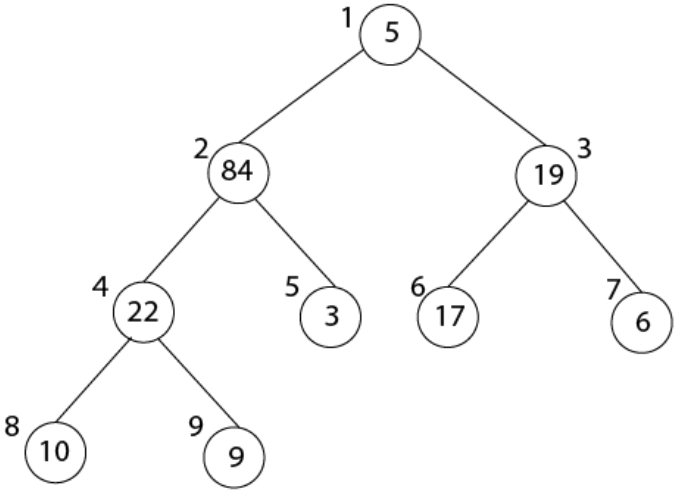
Pass 1: *MAX-HEAPIFY (A, 4)*



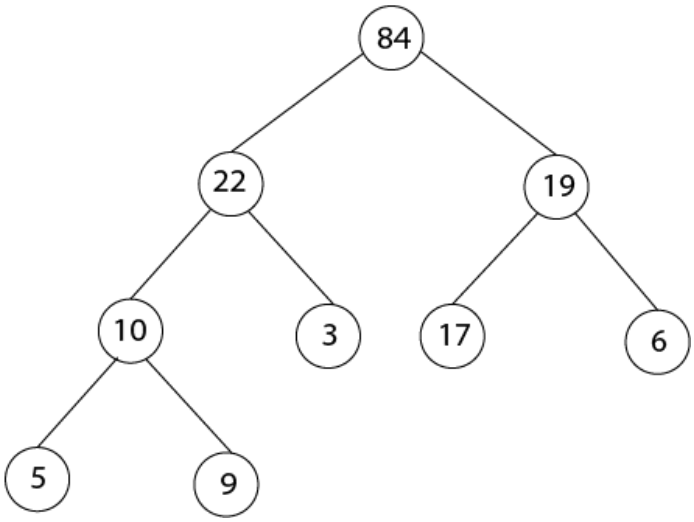
Pass 2: *MAX-HEAPIFY (A, 3)*



Pass 3: *MAX-HEAPIFY (A, 2)*



Pass 4: *MAX-HEAPIFY (A, 1)*



Time Complexity of *BUILD-MAX-HEAP*:

Each call to MAX-HEAPIFY takes $O(log\ n)$ time

There are $O(n)$ such calls specifically, $\left\lfloor \frac{n}{2} \right\rfloor$

Thus the running time is $O(n\ log\ n)$

Heap Sort:

This produces a sorted array by repeatedly removing the largest element from the heap (which is the root of the heap), and then inserting it into the array. The heap is updated after each removal. Once all elements have been removed from the heap, the result is a sorted array.

Working Procedure:

1. Build a max-heap from an unordered array.
2. Find the maximum element, which is located at $A[0]$ because the heap is a max-heap.
3. Swap elements $A[n]$ and $A[0]$ so that the maximum element is at the end of the array where it belongs.
4. Decrement the heap size by one (this discards the node we just moved to the bottom of the heap, which was the largest element).
5. Now run max_heapify on the heap in case the new root causes a violation of the max-heap property.
6. Return to step 2.

Algorithm

HEAP-SORT (A)

BUILD-MAX-HEAP(A)

for $i \leftarrow length\,[A]$ *down to 2 do*

exchange $A[1] \leftrightarrow A[i]$

heap-size$[A] \leftarrow$*heap-size*$[A] - 1$

MAX-HEAPFY$(A, 1)$

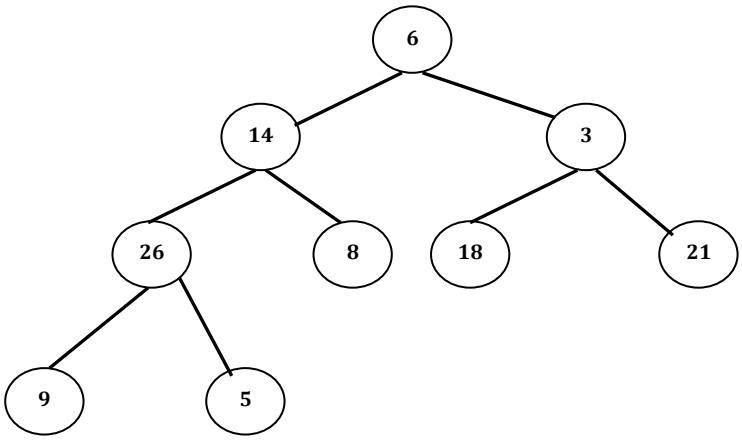
Example:

Illustate the operation of HEAP-SORT on the array $A = \{6, 14, 3, 26, 8, 18, 21, 9, 5\}$

Solution:

Initial Array:

6	14	3	26	8	18	21	9	5



Step 1:

First call *BUILD-MAX-HEAP(A)* to build a max heap. $Heap - size\ (A) = 9$
So, $i = 4$ to 1, call *MAX-HEAPFY*(A, i).

Pass 1: *MAX-HEAPFY*$(A, 4)$

$$A[i] = 26, A[l] = 9, A[r] = 5$$

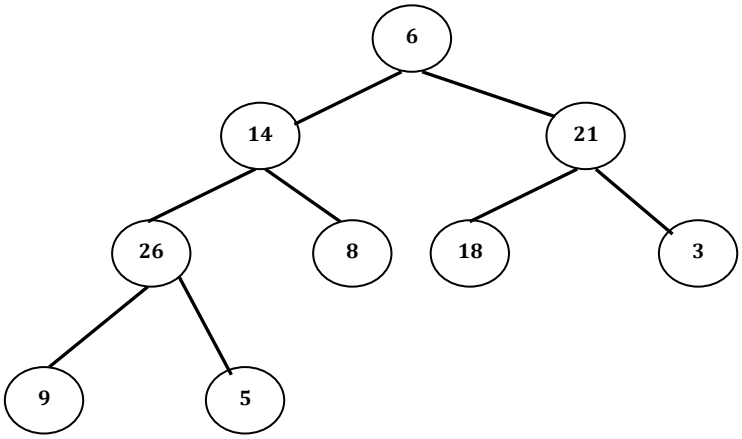
$A[i] > A[l]$ and $A[i] > A[r]$

Pass2: *MAX-HEAPFY*$(A, 3)$

$$A[i] = 3, A[l] = 18, A[r] = 21$$

$A[l] > A[i]$, Thus $largest = 6$

$A[r] > A[largest]$ Thus $largest\ = 7$ and $exchange\ A[i] \leftrightarrow A[largest]$

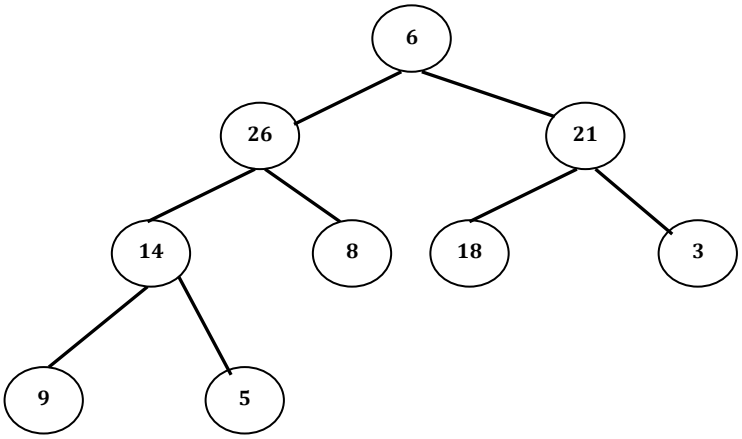


Pass3: *MAX-HEAPFY*$(A, 2)$

$$A[i] = 14, A[l] = 26, A[r] = 8$$

$A[l] > A[i]$, Thus $largest = 4$

$A[r] < A[largest]$ and $exchange\ A[i] \leftrightarrow A[largest]$



Now $largest\ = 4$, call *MAX-HEAPFY*$(A, 4)$

$$A[i] = 14, A[l] = 9, A[r] = 5$$

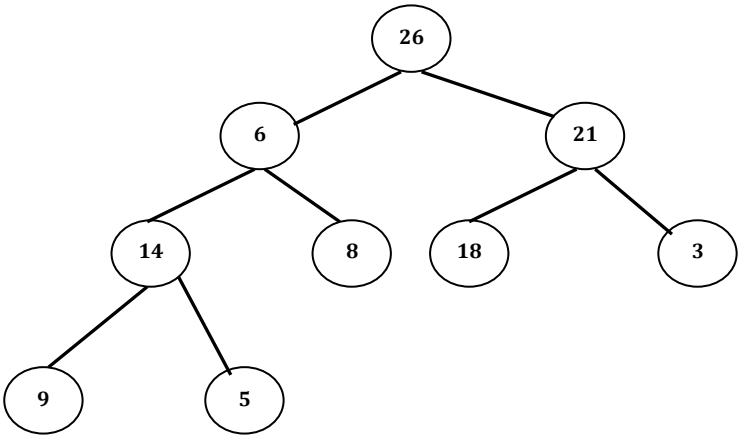
$A[i] > A[l]$ and $A[i] > A[r]$

Pass 4:

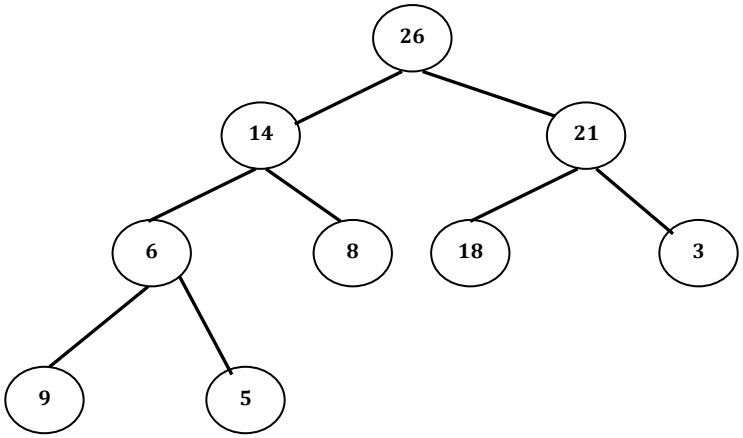
MAX-HEAPFY$(A, 1)$

$$A[i] > A[l], largest = 2$$

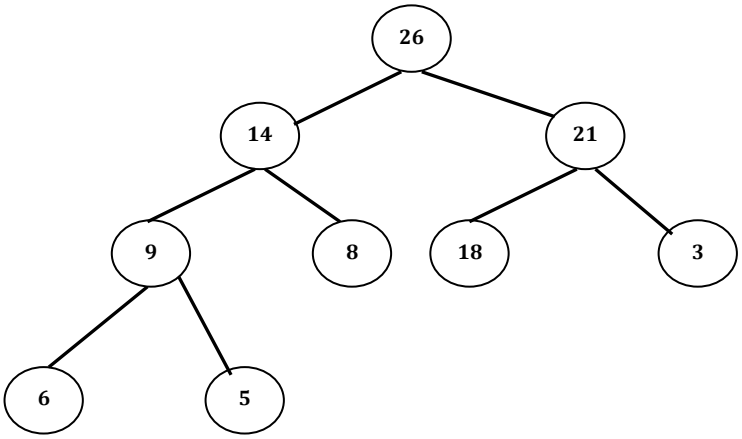
$A[largest] > A[r]$ and $exchange\ A[i] \leftrightarrow A[largest]$



Now $largest = 2$, call $MAX\text{-}HEAPFY(A, 2)$
$A[i] < A[l]$. Thus $largest = 4$
$A[largest] > A[r]$ and $exchange\ A[i] \leftrightarrow A[largest]$



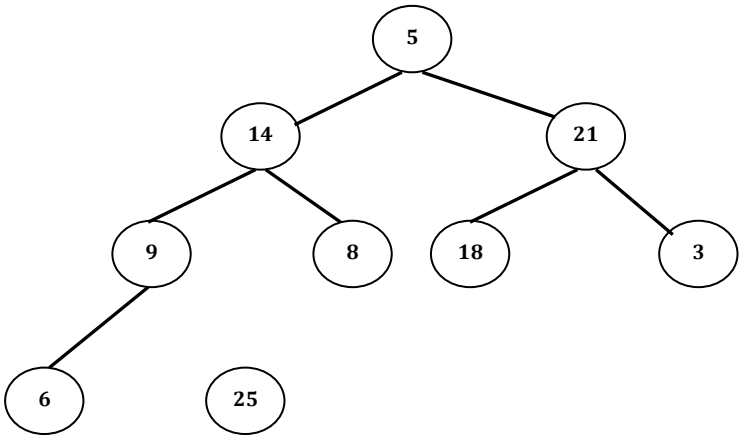
Now $largest = 4$, call $MAX\text{-}HEAPFY(A, 4)$
$A[i] < A[l]$. Thus $largest = 8$
$A[largest] > A[r]$ and $exchange\ A[i] \leftrightarrow A[largest]$



Step 2:
For $i =$ 9 down to 2
$exchange\ A[1] \leftrightarrow A[i]$ and heap-size = heap-size-1 and call $MAX - HEAPIFY\ (A, 1)$

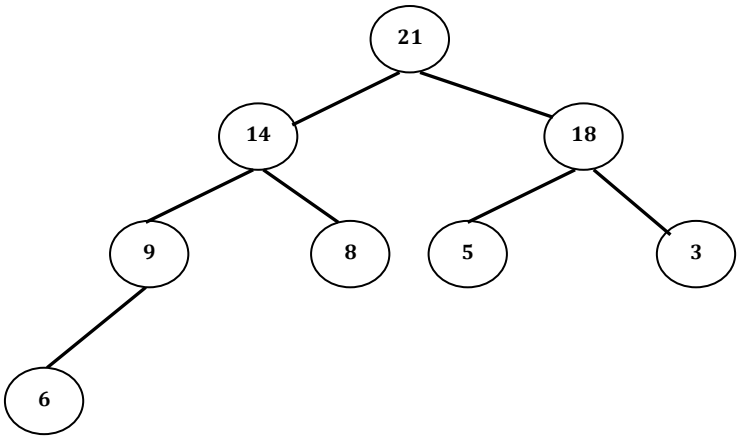
Pass 1:

By exchanging A[1] with A[9] and reducing heap-size by 1 we get



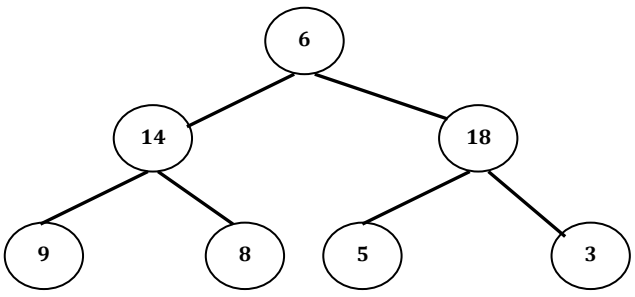
5	14	21	9	8	18	3	6	26

Now applying $MAX - HEAPIFY\ (A, 1)$ we get



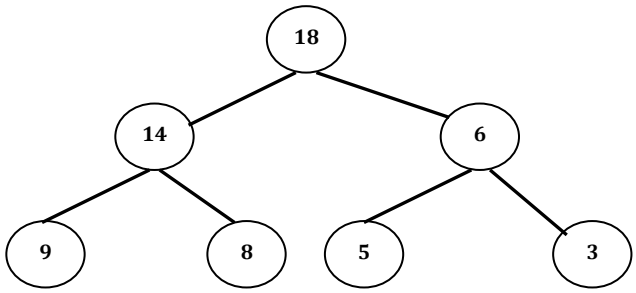
Pass 2:

By exchanging A[1] with A[8] and reducing heap-size by 1 we get



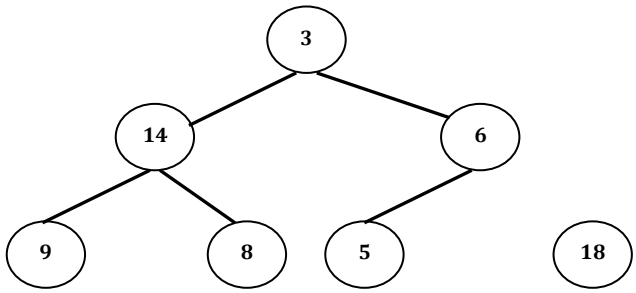
6	14	18	9	8	5	3	21	26

Now applying $MAX - HEAPIFY\ (A, 1)$ we get



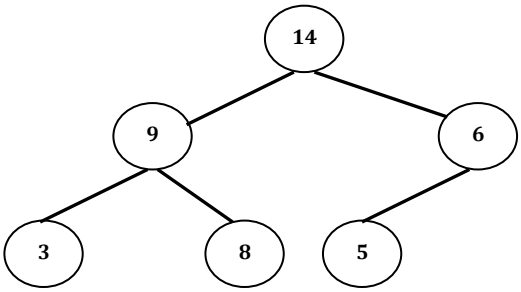
Pass 3:

By exchanging A[1] with A[7] and reducing heap-size by 1 we get



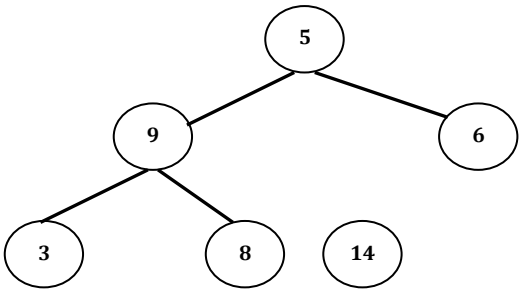
3	14	6	9	8	5	18	21	26

Now applying $MAX - HEAPIFY\ (A, 1)$ we get



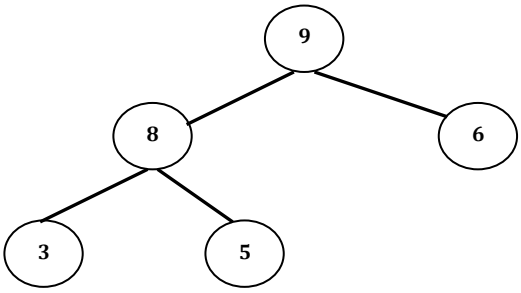
Pass 4:

By exchanging A[1] with A[6] and reducing heap-size by 1 we get



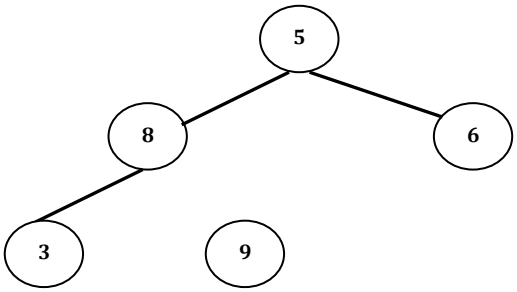
5	9	6	3	8	14	18	21	26

Now applying $MAX - HEAPIFY\ (A, 1)$ we get



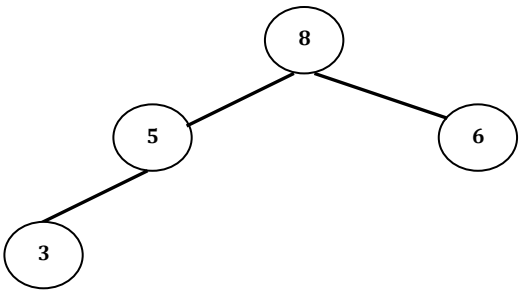
Pass 5:

By exchanging A[1] with A[5] and reducing heap-size by 1 we get



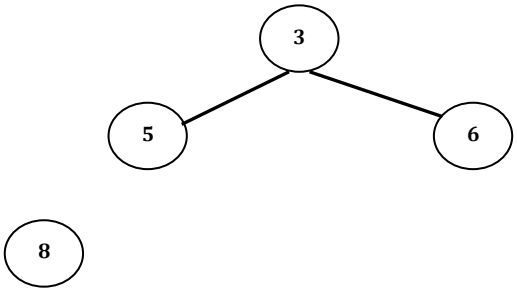
5	8	6	2	9	14	18	21	26

Now applying $MAX - HEAPIFY\ (A, 1)$ we get



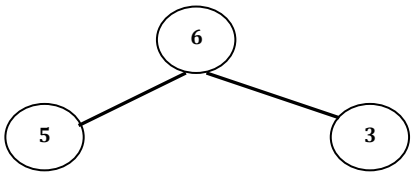
Pass 6:

By exchanging A[1] with A[4] and reducing heap-size by 1 we get



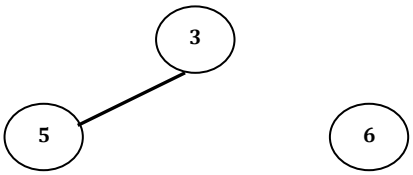
3	5	6	8	9	14	18	21	26

Now applying $MAX - HEAPIFY\ (A, 1)$ we get



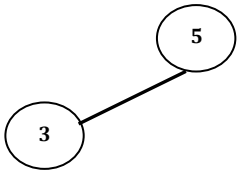
Pass 7:

By exchanging A[1] with A[3] and reducing heap-size by 1 we get



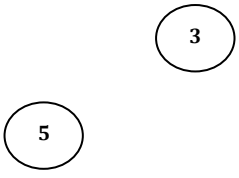
3	5	6	8	9	14	18	21	26

Now applying $MAX - HEAPIFY\ (A, 1)$ we get



Pass 8:

By exchanging A[1] with A[2] and reducing heap-size by 1 we get



3	5	6	8	9	14	18	21	26

Thus the sorted array is:

3	5	6	8	9	14	18	21	26

Time Complexity:

Worst Case: If the elements of the array is alreay in ascending order, line 1 takes $O(n)$ time while line 5 takes $O(\log n)$ time since MAX-HEAPIFY must do $\log n$ exchange. Hence $T(n) = O(n) + n\,O(\log n) = O(n \log n)$.

Best Case: If the elements of the array is in descending order, line 1 calls to MAX-HEAPIFY $O(n/2)$ times without any work and line 5 takes $O(\log n)$ times. Hence $T(n) = O(n/2) + n\,O(\log n) = O(n \log n)$.

Space complexity:

The space complexity of heap-sort is $O(1)$, because it does not require any extra memory.

PRIORITY QUEUE

A priority queue is a type of queue in which each element has a priority value associated with it and they are arranged in a queue based on their priority. Elements with higher priority values are typically retrieved or removed before elements with lower priority values.

Difference between Priority Queue and Normal Queue:

In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority**.**

Properties:

1. Every element in a priority queue has some priority associated with it.
2. An element with the higher priority will be dequeued before the elements with lesser priority.
3. If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Types of Priority queues:

1. **Min priority Queue:** If the element with the smallest value has the highest priority, then that priority queue is called the min-priority queue or ascending order priority queue.
2. **Max Priority Queue:** If the element with the highest value has the highest priority, then that priority queue is called the max-priority queue or descending order priority queue.

Implementation of Priority Queue

Priority queue can be implemented by using:

- an array
- a linked list
- a heap data structure, or
- a binary search tree.

Implementation of Priority Queue by using binary heap:

We can implement the min-priority queue using a min heap, whereas we can implement the max priority queue using a max heap.

MAX-Priority Queue Operations:

A Max-priority queue supports the following operations:

1. $MAXIMUM(S)$: returns the element of S with the largest key.
2. $EXTRACT - MAX(S)$: removes and returns the element of S with the largest key.
3. $INCREASE - KEY\ (S, x, k)$: increases value of element x's key to the new value k. Assume that $k\ \geq$x's current key value.
4. $INSERT(S\ , x\ )$: inserts the element x with key k into the set S, i.e. $S = S \cup \{x\}$

1. Finding the maximum element from the Priority Queue:

The maximum element in a priority queue is the root of the tree.

Algorithm:

MAX-HEAP-MAXIMUM (A)

if $A.heap - size < 1$
    error "heap underflow";
return A[1];

Time Complexity: $O(1)$

2. Extracting the maximum element from the priority queue:

1. Remove the element from the root
2. Remove the last element from the last level of the heap
3. Replace the root with the last element
4. Re-heapify the heap with one fewer node.

Algorithm:

MAX-HEAP-EXTRACT-MAX(A)

1. $max = MAX\text{-}HEAP\text{-}MAXIMUM\ (A)$
2. $A[1] = A[A.heap\text{-}size]$
3. $A.heap\text{-}size = A.heap\text{-}size - 1$
4. $MAX\text{-}HEAPIFY(A, 1)$
5. $return\ max$

Time Complexity: $MAX\text{-}HEAPIFY$ takes $O(log\ n)$ time and since only constant work is added to it, time complexity of MAX-HEAP-EXTRACT-MAX(A) is $O(log\ n)$.

3. Increasing Key Value:

Suppose element whose key to be increased is identified by index i

1. Make sure that key is more than the element at index i.
2. Update $A[i]$ to key
3. Traverse the tree upward comparing $A[i]$ to its parent and swapping keys if necessary, until $A[i]$'s key is smaller than its parent's key.

Algorithm:

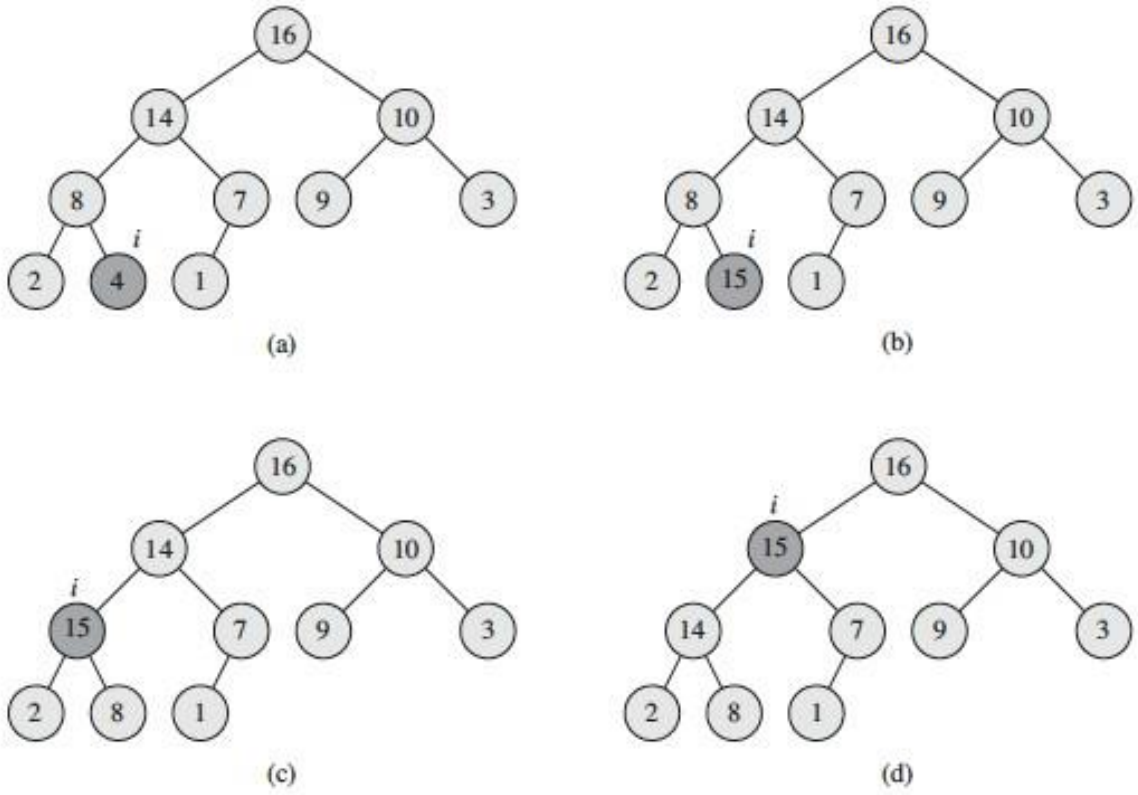
MAX-HEAP-INCREASE-KEY$(\boldsymbol{A}, \boldsymbol{i}, \boldsymbol{key})$

if key< A[i] then
    error "new key is smaller than current key"
end if
A[i] ← *key*
while i > 1 and A[Parent(i)] < A[i] do
    exchange $A[i] \leftrightarrow A[Parent(i)]$
    i ←*Parent(i)*
end while

Time Complexity: It is height of tree $O(log\ n)$

Example:

MAX-HEAP-INCREASE-KEY$(A, 9, 15)$: Update node 9 from 4 to 15.



4. Inserting into the heap

Given a key to insert into the heap:

1. Increment the heap size.
2. Insert a new node in the last position in the heap, with key $-\infty$.
3. Increase the $-\infty$ key to key using the ***MAX-HEAP-INCREASE-KEY*** procedure defined above.

Algorithm:

MAX-HEAP-INSERT $(\boldsymbol{A}, \boldsymbol{key})$

1. $heap\text{-}size(A) \leftarrow heap\text{-}size(A) + 1$
2. $A[heap\text{-}size(A)] \leftarrow -\infty$
3. $MAX\text{-}HEAP\text{-}INCREASE\text{-}KEY(A, heap\text{-}size(A), key)$

Time Complexity: $O(log\ n)$

LOWER BOUNDS OF SORTING

All of the sorting algorithms we've seen so far are comparison-based, because they all do their work primarily by comparing pairs of elements and making decisions on the basis of those comparisons.

- Insertion sort compares elements when doing its insertions.
- Selection sort compares elements when looking for maximums.
- Heapsort compares elements to build and maintain the heap structure.
- Quicksort compares elements when partitioning them.
- Mergesort compares elements when merging them.

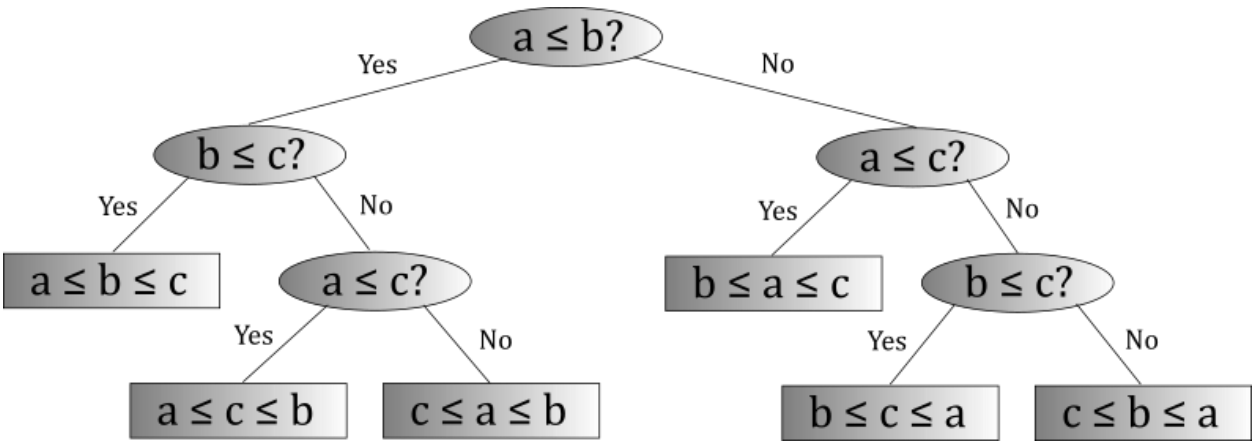
All sorting algorithms have worst case running time $\Omega(n \log n)$.

Can we say that it is impossible to sort faster than $\Omega(n \log n)$ in the worst case? If we could, then this would be what it's called a lower bound.

Modeling a comparison-based sorting algorithm as a decision tree

A comparison-based sorting algorithm does its work by comparing pairs of elements, then taking action based on the results of those comparisons. We could represent such an algorithm as a decision tree, in which the non-leaf nodes represent comparisons the algorithm makes, and the leaf nodes represent the final conclusions that the algorithm can reach regarding the correct sorted order of the elements.

For example, this decision tree represents a comparison-based sorting algorithm capable of sorting three elements.



Theorem:

Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Proof:

Assume that elements are distinct numbers 1 through n.

Number of comparisons is equal to height of the tree.

If there are n elements, there are n! permutations of those elements.

This implies that there must be $n!$ leaves.

We know that a tree of height h has at most 2^h leaves.

$$2^h \geq n!$$

$$\Rightarrow h \geq log(n!)$$

$$\Rightarrow h \geq n \log n$$

$\therefore h = \Omega(n \log n)$

DYNAMIC PROGRAMMING

It is a programming technique in which solution is obtained from a sequence of decisions.

A dynamic programming problem can be divided into a number of stages where an optimal decision must be made at each stage. The decision made at each stage must take into account its effects not only on the next stage, but also on the entire subsequent stages. Dynamic programming provides a systematic procedure whereby starting with the last stage of the problem and working backwards one makes an optimal decision for each stage of problem. The information for the last stage is the information derived from the previous stage.

Dynamic programming design involves 4 major steps.

1. Characterize the structure of optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimum solution in a bottom up fashion.
4. Construct an optimum solution from computed information

General Characteristics of Dynamic Programming:

The general characteristics of Dynamic programming are

1. The problem can be divided into stages with a policy decision required at each stage.
2. Each stage has number of states associated with it.
3. Given the current stage an optimal policy for the remaining stages is independent of the policy adopted.
4. The solution procedure begins be finding the optimal policy for each state of the last stage.
5. A recursive relation is available which identifies the optimal policy for each stage with n stages remaining given the optimal policy for each stage with (n-1) stages remaining.

Difference between Divide-and Conquer & Dynamic Programming

Sl. No.	**Divide-and Conquer**	**Dynamic Programming**
1.	It Breaks a problem into smaller sub-problems and solves them independently.	Breaks a problem into overlapping sub-problems and solves them recursively or iteratively, storing their solutions.
2.	It can be thought of as top-down algorithms.	It can be thought of as bottom-up algorithm
3.	Sub-problems are independent of each other.	Sub-problems can overlap or share common sub-problems.

4.	Generally simpler to understand and implement.	Can often be quite complex and tricky
5.	It can be used for any kind of problems.	It is generally used for optimization problem.

Applications of Dynamic Programming

1. Matrix Chain Multiplication
2. Longest common sequence
3. Assembly line scheduling
4. Optimal Binary search Trees
5. 0/1 Knapsack Problem
6. Shortest path problem
7. Travelling sales person problem

Matrix Chain Multiplication:

Given a sequence of n matrices $A_1, A_2, \dots A_n$ of order $p_0 \times p_1, p_1 \times p_2, \dots, p_{n-1} \times p_n$ respectively, the objective is to compute their product $A_1 \times A_2 \times \dots \times A_n$ using minimum number of scalar multiplications.

Suppose there are two matrices A_1& A_2 of order (3×3) and (3×2) respectively:

$$A_1 = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \& A_2 = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} \text{ then}$$

$$A_1 \times A_2 = \begin{bmatrix} a_{11}.b_{11} + a_{12}.b_{21} + a_{13}.b_{31} & a_{11}.b_{12} + a_{12}.b_{22} + a_{13}.b_{32} \\ a_{21}.b_{11} + a_{22}.b_{21} + a_{23}.b_{31} & a_{21}.b_{12} + a_{22}.b_{22} + a_{23}.b_{32} \\ a_{31}.b_{11} + a_{32}.b_{21} + a_{33}.b_{31} & a_{31}.b_{12} + a_{32}.b_{22} + a_{33}.b_{32} \end{bmatrix}$$

Total Number of scalar multiplications

$$= 3 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + 3 = 18 = 3 \times 3 \times 2$$

If A_1 is of order $p_0 \times p_1$ and A_2 is of order $p_1 \times p_2$ then number of scalar multiplications in computation of $A_1 \times A_2$ is $p_0 p_1 p_2$.

Algorithm to multiply two matrices:

```
MATRIX-MULTIPLY (A, B, C, m, n, p)
{
        for i = 1 to m do
        {
                for j = 1 to n do
                {
                        for k = 1 to p do

                                C[i, j] = C[i, j] + A[i, k] * B[k, j];
                }
        }
}
```

The running time of the above algorithm is mnp.

Since matrix multiplication is non-commutative and associative, there are different ways to compute $A_1 \times A_2 \times \ldots \times A_n$ and our objective is to find the best way (minimum number of multiplications).

For example, let us consider three matrices A_1, A_2, A_3 of order $(10 \times 100), (100 \times 5)$ and (5×50) respectively.

Here $p_0 = 10, p_1 = 100, p_2 = 5$ and $p_3 = 50$

There are two ways to compute $A_1 \times A_2 \times A_3$:

$A_1 \times A_2 \times A_3 = (A_1 \times A_2) \times A_3 = A_1 \times (A_2 \times A_3)$

Computation of $(A_1 \times A_2) \times A_3$:

$A_1 \times A_2 = A_{12}$ requires $10 \times 100 \times 5 = 5000$ multiplications

$A_{12} \times A_3$ requires $10 \times 5 \times 50 = 2500$ multiplications

Total: 7500 multiplications

Computation of $A_1 \times (A_2 \times A_3)$:

$A_2 \times A_3 = A_{23}$ requires $100 \times 5 \times 50 = 25000$ multiplications

$A_1 \times A_{23}$ requires $10 \times 100 \times 50 = 50000$ multiplications

Total: 75000 multiplications

No. of ways for parenthesizing the matrices:

Let $P(n)$ = Possible number of parenthesization to multiply n matrices $A_1 \times A_2 \times \ldots \times A_n$

$P(1) = 1$ (A_1)

$P(2) = 1$ $(A_1 \times A_2)$

$P(3) = 2$ $(A_1 \times A_2) \times A_3$

$A_1 \times (A_2 \times A_3)$

$P(4) = 5$ $A_1 \times (A_2 \times (A_3 \times A_3))$

$A_1 \times ((A_2 \times A_3) \times A_3)$

$(A_1 \times A_2) \times (A_3 \times A_3)$

$((A_1 \times A_2) \times A_3) \times A_3$

$(A_1 \times (A_2 \times A_3)) \times A_3$

In general if there are n matrices then there are $n-1$ places where we can split the sequence into two parts: one consists of k matrices and the other consists of $n-k$ matrices.

If there are p ways for parenthesizing the left sequence and q ways for parenthesizing the right sequence then total number of ways will be $p \times q$.

Since k has $n-1$ choices then

$$P(n) = \begin{cases} 1 & if\ n = 1 \\ \sum_{k=1}^{n-1} P(k) \times P(n-K) & if\ n \geq 2 \end{cases}$$

$$P(n) = \begin{cases} 1 & if\ n = 1 \\ \frac{2(n-1)C_{n-1}}{n} & if\ n \geq 2 \end{cases}$$

Dynamic Programming Approach:

Input: $(p_0, p_1, \ldots p_n)$ i.e. A sequence of n matrices $A_1, A_2, \ldots A_n$ of order $(p_0 \times p_1), (p_1 \times p_2), \ldots (p_{n-1} \times p_n)$

Output: A parenthesization of $A_1 \times A_2 \times \ldots \times A_n$ that minimizes the total number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.

1. Characterize the structure of optimal solution:

Our first step in the dynamic-programming paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to sub-problems.

Let $A_{i..j} = A_i \times A_{i+1} \times \ldots \times A_j$

The order of $A_{i..j}$ is $(p_{i-1} \times p_j)$

Our objective is to break the problem into several simpler and smaller sub-problems of similar structure.

$$A_{i..j} = A_{i..k} \times A_{k+1..j}$$

That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$and then multiply them together to produce the final product $A_{i..j}$.

$$The\ cost\ of\ this\ parenthesization\ =$$

$$The\ cost\ of\ computing\ the\ matrix\ A_{i..k} + The\ cost\ of\ computing\ A_{k+1..j} + The\ cost\ of\ multiplying\ them\ together.$$

2. Recursively define the value of an optimal solution:

Let $m[i,j]$ be the minimal number of multiplications needed to compute $A_{i..j}$

$$= A_i \times A_{i+1} \times \ldots \times A_j$$

If the final multiplication for $A_{i..j}$is $A_{i..j} = A_{i..k} \times A_{k+1..j}$ then we have to compute recursively the best way to multiply the chain from i to k, and from $k\ +\ 1$ to j, and add the cost of the final product. This means that

$$m[i,j]\ =\ m[i,k]\ +\ m[k+1,j]\ +\ p_{i-1} \times p_k \times p_j.$$

We have to find value of k so that $m[i,j]$ is minimum.

We do not know the optimal value of k, hence,

$$m[i,j]\ = \begin{cases} 0 & if\ i = j \\ \min\limits_{i \le k < j} \{m[i,k]\ +\ m[k+1,j]\ +\ p_{i-1} \times p_k \times p_j\} & if\ i < j \end{cases}$$

To keep track of optimal sub-solutions, we store the value of k in a table $s[i,j]$. Note that, k is the place at which we split the product $A_{i..j}$ to get an optimal parenthesization.

3.Compute the value of optimal solution in a bottom-up fashion by constructing cost tables.

That is we calculate in the order:

$m[1,2]$, $m[2,3]$, $m[3,4]$ .. .. $m[n-2,n-1]$ $m[n-1,n]$
$m[1,3]$ $m[2,4]$ $m[3,5]$ .. .. $m[n-2,n]$
$m[1,4]$ $m[2,5]$ $m[3,6]$ .. .. $m[n-3,n]$
.
.
$m[1,n]$

4.Construct an optimal solution from computed information.

The array $s[i,j]$ gives the optimal split points.

Example:

Consider a sequence of matrices A, B, C and D with order $(3\times 7), (7\times 6), (6\times 2)$ and (2×9) respectively. Find the lowest cost parenthesization to multiply the given matrices using matrix chain multiplication.

Solution:

Given, $n=4, p_0=3, p_1=7, p_2=6, p_3=2$ and $p_4=9$

$$P(1)=1$$

$$P(2)=P(1).P(1)=1\times 1=1$$

$$P(3)=P(1).P(2)+P(2).P(1)+P(1).P(1)=1\times 1+1\times 1=2$$

$$P(4)=\sum_{k=1}^{4}P(k)P(4-k)=P(1).P(4-1)+P(2).P(4-2)+P(3).P(4-3)$$
$$+P(4).P(4-4)=P(1).P(3)+P(2).P(2)+P(3).P(1)=2+1+2=5.$$

Hence number of possible parenthesization for $n=4$ is 5.

1. $((A\times B)\times C)\times D$
2. $(A\times(B\times C))\times D$
3. $(A\times B)\times(C\times D)$
4. $A\times((B\times C)\times D)$
5. $A\times\big(B\times(C\times D)\big)$

Initialization:

for $i = 1$ to n do $m[i, i] = 0$

$m[1,1] = 0, m[2, 2] = 0, m[3, 3] = 0, m[4, 4] = 0$

$\boldsymbol{m[i,j]}$

	1	2	3	4
1	0			
2		0		
3			0	
4				0

Iteration 1:

$$m[1,2] = \min_{1 \le k < 2} \{m[1,1] + m[2,2] + p_0 \times p_1 \times p_2\} = 0 + 0 + 3 \times 7 \times 6 = 126$$

$$m[2,3] = \min_{1 \le k < 2} \{m[2,2] + m[3,3] + p_1 \times p_2 \times p_3\} = 0 + 0 + 7 \times 6 \times 2 = 84$$

$$m[3,4] = \min_{1 \le k < 2} \{m[3,3] + m[4,4] + p_2 \times p_3 \times p_4\} = 0 + 0 + 6 \times 2 \times 9 = 108$$

$\boldsymbol{m[i,j]}$

	1	2	3	4
1	0	126		
2		0	84	
3			0	108
4				0

Iteration 2:

$$m[1,3] = \min_{1 \le k < 3} \{m[1,1] + m[2,3] + p_0 \times p_1 \times p_3, \quad m[1,2] + m[3,3] + p_0 \times p_2 \times p_3\}$$

$$= \min\{(0 + 84 + 3 \times 7 \times 2), \quad 126 + 0 + 3 \times 6 \times 2\}$$

$$= \min\{106, 162\} = 106 (k = 1)$$

$$m[2,4] = \min_{2 \le k < 4} \{m[2,2] + m[3,4] + p_1 \times p_2 \times p_4, \quad m[2,3] + m[4,4] + p_1 \times p_3 \times p_4\}$$

$$= \min\{(0 + 108 + 7 \times 6 \times 9), \quad 84 + 0 + 7 \times 2 \times 9\}$$

$= \min\{486,210\} = 210(k = 3)$

$\boldsymbol{m[i,j]}$

	1	2	3	4
1	0	126	106	
2		0	84	210
3			0	108
4				0

Iteration 3:

$$m[1,4] = \min_{1\le k<4}\{m[1,1] + m[2,4] + p_0 \times p_1 \times p_4,\quad m[1,2] + m[3,4] + p_0 \times p_2 \times p_4,$$
$$m[1,3] + m[4,4] + p_0 \times p_3 \times p_4\}$$

$$= \min\{(0 + 210 + 3 \times 7 \times 9),\quad 126 + 108 + 3 \times 6 \times 9,\ 106 + 0 + 3 \times 2 \times 9\}$$

$$= \min\{399, 396, 160\} = 160(k = 3)$$

$\boldsymbol{m[i,j]}$

	1	2	3	4
1	0	126	106	160
2		0	84	210
3			0	108
4				0

Therefore, the optimal solution $m[1,4] = 160$

$\boldsymbol{s[i,j]}$

	1	2	3	4
1	0	126/1	106/1	160/3
2		0	84/2	210/3
3			0	108/3
4				0

The lowest cost value at [1, 4] is achieved when k = 3, therefore, the first parenthesization must be done at 3.

$$(A \times B \times C) \times D$$

The lowest cost value at [1, 3] is achieved when k = 1, therefore the next parenthesization is done at 1.

$$\big(A \times (B \times C)\big) \times D$$

and optimal parenthesization is

$$\big(A \times (B \times C)\big) \times D$$

Algorithm Matrix-Chain-Order:

This algorithm determines the optimal number of scalar multiplications needed to compute product of a sequence of matrices.

$s[i, j]$: The value of k such that an optimal parenthesization of $A_i, A_2, \dots A_j$ splits into $(A_i, \dots, A_k)$ and $A_{k+1}, \dots, A_j$.

$m[i, j]$: Minimal number of multiplications needed to compute $A_{i..j}$

$\boldsymbol{Matrix - Chain - Order(p)}$

```
{
    n = length[p] – 1;
    for i = 1 to n do
        m[i, i] = 0;
    for l = 2 to n do // l is the chain length.
    {
        for i = 1 to n − l + 1 do
        {
            j = i + l − 1;
            m[i, j] = ∞;
            for k = i to j – 1 do
            {
                q = m[i, k] + m[k + 1, j] + p[i − 1] * p[k] * p[j];
                if q < m[i, j] then
                {
                    m[i, j] = q;
                    s[i, j] = k;
                }
            }
        }
    }
    return m and s;
}
```

Time Complexity:
There are three nested loops and each loop index takes on $\leq n$ values. Hence the time complexity is $O(n^3)$.

Algorithm PRINT − OPTIMAL − PARENTHESIS
This algorithm prints an optimal parenthesization of product of the matrices $A_1, A_2, \ldots A_n$.

```
PRINT − OPTIMAL − PARENTHESIS (s, i, j)
{
    if (i == j) then print "A"_i;
    else
    {
        print "(";
        PRINT − OPTIMAL − PARENTHESIS (s, i, s[i, j]);
        PRINT − OPTIMAL − PARENTHESIS (s, s[i, j] + 1, j);
        print ")";
    }
}
```

Example:

Consider a sequence of matrices $A_1, A_2, \ldots, A_5$ with order $(4 \times 10), (10 \times 3), (3 \times 12), (12 \times 20)$, and (20×7) respectively. Find the lowest cost parenthesization to multiply the given matrices using matrix chain multiplication.

Solution:

Initialization:

Given, $n = 5, p_0 = 4, p_1 = 10, p_2 = 3, p_3 = 12,\ p_4 = 20, p_5 = 7$

$m[1,1\ ] = 0, m[2\ ,2\ ] = 0, m[3\ ,3\ ] = 0, m[4\ ,4\ ] = 0, m[5\ ,5\ ] = 0$

$\boldsymbol{m[i,j]}$	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

Iteration 1:

$$m[1,2] = \min_{1 \leq k < 2} \{m[1,1] + m[2,2] + p_0 \times p_1 \times p_2\} = 0 + 0 + 4 \times 10 \times 3 = 120$$

$$m[2,3] = \min_{1\le k<2}\{m[2,2] + m[3,3] + p_1 \times p_2 \times p_3\} = 0 + 0 + 10 \times 3 \times 12 = 360$$

$$m[3,4] = \min_{1\le k<2}\{m[3,3] + m[4,4] + p_2 \times p_3 \times p_4\} = 0 + 0 + 3 \times 12 \times 20 = 720$$

$$m[4,5] = \min_{1\le k<2}\{m[4,4] + m[5,5] + p_3 \times p_4 \times p_5\} = 0 + 0 + 12 \times 20 \times 7 = 1680$$

$m[i,j]$	1	2	3	4	5
1	0	120			
2		0	360		
3			0	720	
4				0	1680
5					0

Iteration 2:

$$m[1,3] = \min_{1\le k<3}\begin{cases} m[1,1] + m[2,3] + p_0 \times p_1 \times p_{3,} & = 0 + 360 + 4 \times 10 \times 12 = 840 \\ m[1,2] + m[3,3] + p_0 \times p_2 \times p_3 & = 120 + 0 + 4 \times 3 \times 12 = 264 \end{cases}$$

$$= 264\ (k = 2)$$

$$m[2,4] = \min_{2\le k<4}\begin{cases} m[2,2] + m[3,4] + p_1 \times p_2 \times p_{4,} & = 0 + 720 + 10 \times 3 \times 20 = 1320 \\ m[2,3] + m[4,4] + p_1 \times p_3 \times p_4 & = 360 + 0 + 10 \times 12 \times 20 = 2760 \end{cases}$$

$$= 1320(k = 2)$$

$$m[3,5] = \min_{3\le k<5}\begin{cases} m[3,3] + m[4,5] + p_2 \times p_3 \times p_{5,} & = 0 + 1680 + 3 \times 12 \times 7 = 1932 \\ m[3,4] + m[5,5] + p_2 \times p_4 \times p_5 & = 720 + 0 + 3 \times 20 \times 7 = 1140 \end{cases}$$

$$= 1140(k = 4)$$

$m[i,j]$	1	2	3	4	5
1	0	120	264		
2		0	360	1320	
3			0	720	1140
4				0	1680
5					0

Iteration 3:

$$m[1,4]$$

$$= \min_{1\le k<4} \begin{cases} m[1,1] + m[2,4] + p_0 \times p_1 \times p_4 & = 0 + 1320 + 4 \times 10 \times 20 & = 2120 \\ m[1,2] + m[3,4] + p_0 \times p_2 \times p_4 & = 120 + 720 + 4 \times 3 \times 20 & = 1080 \\ m[1,3] + m[4,4] + p_0 \times p_3 \times p_4 & = 264 + 0 + 4 \times 12 \times 20 & = 1224 \end{cases}$$

$$= 1080(k = 2)$$

$$m[2,5]$$

$$= \min_{2\le k<5} \begin{cases} m[2,2] + m[3,5] + p_1 \times p_2 \times p_5 & = 0 + 1140 + 10 \times 3 \times 7 & = 1350 \\ m[2,3] + m[4,5] + p_1 \times p_3 \times p_5 & = 360 + 1680 + 10 \times 12 \times 7 & = 2880 \\ m[2,4] + m[5,5] + p_1 \times p_4 \times p_5 & = 1320 + 0 + 10 \times 20 \times 7 & = 2720 \end{cases}$$

$$= 1350(k = 2)$$

$\boldsymbol{m[i,j]}$	1	2	3	4	5
1	0	120	264	1080	
2		0	360	1320	1350
3			0	720	1140
4				0	1680
5					0

Iteration 4:

$$m[1,5]$$

$$= \min_{1\le k<4} \begin{cases} m[1,1] + m[2,5] + p_0 \times p_1 \times p_5 & = 0 + 1350 + 4 \times 10 \times 7 & = 1630 \\ m[1,2] + m[3,5] + p_0 \times p_2 \times p_5 & = 120 + 1140 + 4 \times 3 \times 7 & = 1344 \\ m[1,3] + m[4,5] + p_0 \times p_3 \times p_5 & = 264 + 1680 + 4 \times 12 \times 7 & = 2016 \\ m[1,4] + m[5,5] + p_0 \times p_4 \times p_5 & 1380 + 0 + 4 \times 20 \times 7 & = 1544 \end{cases}$$

$$= 1344(k = 2)$$

$\boldsymbol{m[i,j]}$	1	2	3	4	5
1	0	120	264	1080	1344
2		0	360	1320	1350
3			0	720	1140
4				0	1680
5					0

Therefore the optimal solution $m[1,5] = 1344$

Now for optimal parenthesization, each time we find the optimal value of $m[i,j]$, we have to store the value of k that is used.

$s[i,j]$	1	2	3	4	5
1	0	120/1	264/2	1080/2	1344/2
2		0	360/2	1320/2	1350/2
3			0	720/3	1140/4
4				0	1680/4
5					0

The k value for the solution $m[1,5]$ is 2, so we have $(A_1 \times A_2)(\mathrm{A}_3 \times \mathrm{A}_4 \times \mathrm{A}_5)$

The k value for $m[3,5]$ is 4, so we have $(A_1 \times A_2)((\mathrm{A}_3 \times \mathrm{A}_4) \times \mathrm{A}_5)$

Therefore the optimal parenthesis is $\big((A_1 \times A_2)((\mathrm{A}_3 \times \mathrm{A}_4) \times \mathrm{A}_5)\big)$.

LONGEST COMMON SUBSEQUENCE

A **subsequence** of a string S, is a set of characters that appear in left-to-right order, but not necessarily consecutively.

For Example: Consider a string S: $A\ C\ T\ T\ G\ C\ G$

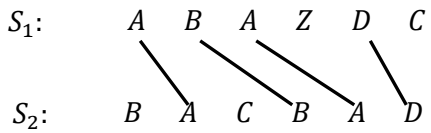
$A\ C\ T$, $A\ T\ T\ C$, T , $A\ C\ T\ T\ G\ C$ are all subsequences of S but, $T\ T\ A$ is not a subsequence of S.

A **common subsequence** of two strings is a subsequence that appears in both strings. Given two sequences S_1 and S_2, a common subsequence is a subsequence that occurs in both S_1 and S_2.

A **longest common subsequence** is a common subsequence of maximal length.

Example:

Consider two strings:



The longest common subsequence is $A\ B\ A\ D$ of length 4.

Example:

Consider two strings:

S_1: $A\ A\ A\ C\ C\ G\ T\ G\ A\ G\ T\ T\ A\ T\ T\ C\ G\ T\ T\ C\ T\ A\ G\ A\ A$

S_2: $C\ A\ C\ C\ C\ C\ T\ A\ A\ G\ G\ T\ A\ C\ C\ T\ T\ \ T\ G\ G\ T\ T\ C$

The longest common subsequence is $A\ C\ C\ T\ A\ G\ T\ A\ C\ T\ T\ T\ G$ of length 13.

Dynamic programming Approach:

Let $X = \{x_1, x_2, \dots, x_m\}$ and $Y = \{y_1, y_2, \dots, y_n\}$ be two strings of length m and n respectively.

Suppose that X_i and Y_j be be the prefixes of X and Y of length i and j respectively.

Let $c[i,j] =$ length of LCS of X_i and Y_j

$c[m,n] =$ length of LCS of X and Y which is the final solution.

Step 1: *Characterize the structure of the optimal solution:*

Case 1: If Last characters of both X and Y is same then the problem gets reduced to finding the LCS of the remaining substrings of size m-1 and n-1 respectively and add 1 to get the output. $LCS\ (X, Y) = LCS\ (X_{m-1}, Y_{n-1}) + 1$.

X = | A | B | C | B | D | A | A |

Y = | B | D | C | A | B | A |

Case 2: If the last characters of two strings are not equal, there are two possibilities for smaller sub-problems and we need to find maximum of them.

1. Find the length of the LCS by excluding the last character of string X and including the last character of String Y.

X = | A | B | C | B | D | A | B |

Y = | B | D | C | A | B | A |

$$LCS\ (X, Y) = LCS\ (X_{m-1}, Y)$$

2. Find the length of the LCS by including the last character of string X and excluding the last character of String Y.

X = | A | B | C | B | D | A | B |

Y = | B | D | C | A | B | A |

$$LCS\ (X, Y) = LCS\ (X, Y_{n-1})$$

$$\therefore LCS\ (X, Y) = \begin{cases} 1 + LCS\ (X_{m-1}, Y_{n-1}) & if\ x_m = y_n \\ \max(LCS\ (X_{m-1}, Y), LCS\ (X, Y_{n-1}) & otherwise \end{cases}$$

Step 2: ***Recursively define the value of an optimal solution.***

$$c[i, j] = \begin{cases} 1 + c\ [i-1, j-1] & if\ x_i = y_j \\ \max(c\ [i-1, j], c[i, j-1]) & otherwise \end{cases}$$

Where $c[i, j]$ = length of LCS of X_i and Y_j

Step 3: ***Compute the value of optimal solution in a bottom-up fashion***

Start with $i = j = 0$, Since X_0 and Y_0 are empty strings, their LCS is always empty. Hence $c[0,0] = 0$.

- LCS of empty string and any other string is empty. Hence for every i and j: $c[0,j] = c[i,0] = 0$.
- Calculate other values of $c[i,j]$ by following procedure:

  Case 1: If $x_i = y_j$, then length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{i-1} plus 1.

  Case 2: If $x_i \neq y_j$, then our solution is not improved, and the length of $LCS(X_i, Y_j)$ is the same as before (i.e. maximum of $LCS(X_i, Y_{j-1})$ and $LCS(X_{i-1}, Y_j)$.

Step 4: ***Construct an optimum solution from computed information.***

Working Procedure to find LCS of *X* and *Y*:

1. Create a blank table of dimension $(n+1, m+1)$ where n and m are the lengths of X and Y respectively. Put 0's in the first row and first column of the table.
2. If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element, encircle it and point an arrow to the diagonal cell. Otherwise, take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.
3. Repeat step 2 until the table is filled.
4. The value in the last row and the last column is the length of the longest common subsequence.
5. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to encircled cell form the longest common subsequence.

Example:

Find the LCS of the following two strings:

X:	A	C	A	D	B
Y:	C	B	D	A	

Solution:

Initialization:

	Y_i	C	B	D	A
X_i	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

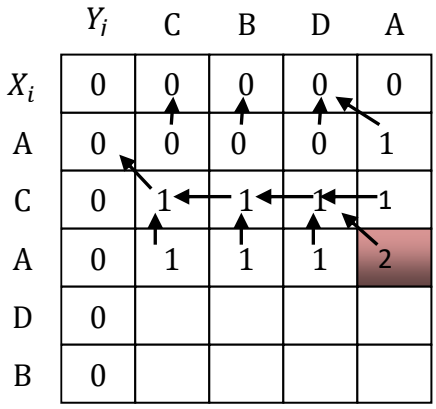
Iteration 1:

	Y_i	C	B	D	A
X_i	0	0	0	0	0
A	0	0	0	0	1
C	0				
A	0				
D	0				
B	0				

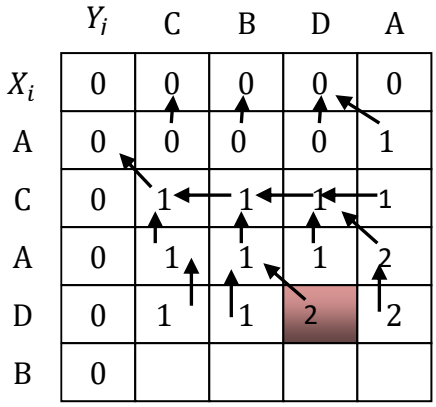
Iteration 2:

	Y_i	C	B	D	A
X_i	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0				
D	0				
B	0				

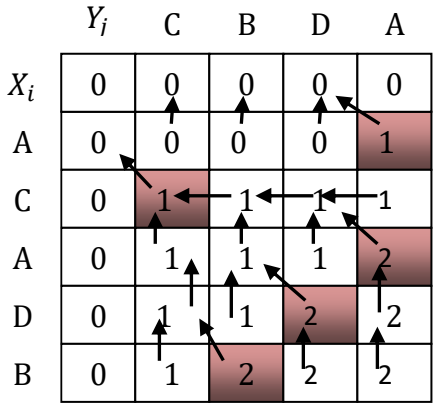
Iteration 3:



Iteration 4:



Iteration 5:



Hence length of Longest common subsequence is 2 and the LCS can be obtained using backtracking i. e. CA

Example:

Find the LCS of the following two strings:

X: A G G T A B

Y: G X T X A Y B

Solution:

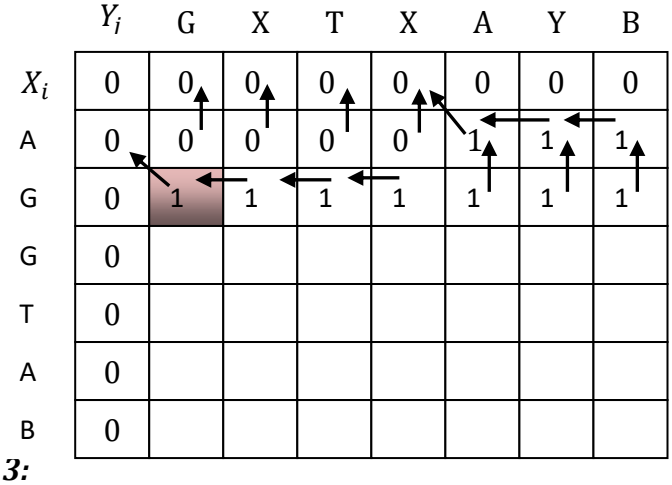
Initialization:

	Y_i	G	X	T	X	A	Y	B
X_i	0	0	0	0	0	0	0	0
A	0							
G	0							
G	0							
T	0							
A	0							
B	0							

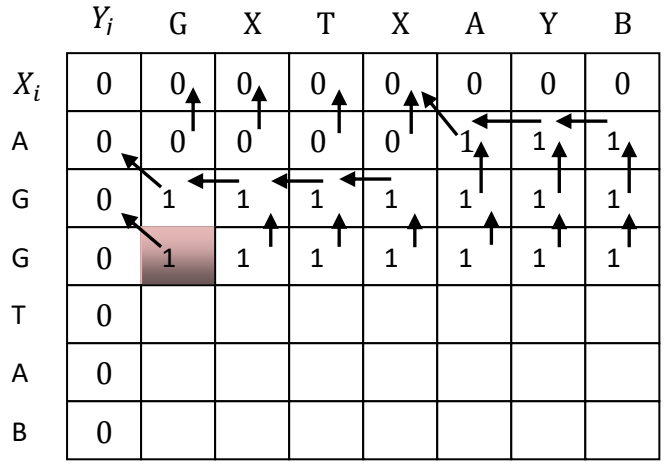
Iteration 1:

	Y_i	G	X	T	X	A	Y	B
X_i	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0							
G	0							
T	0							
A	0							
B	0							

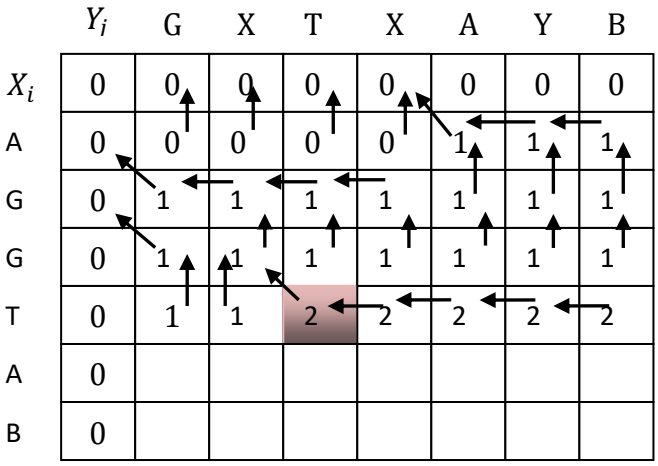
Iteration 2:



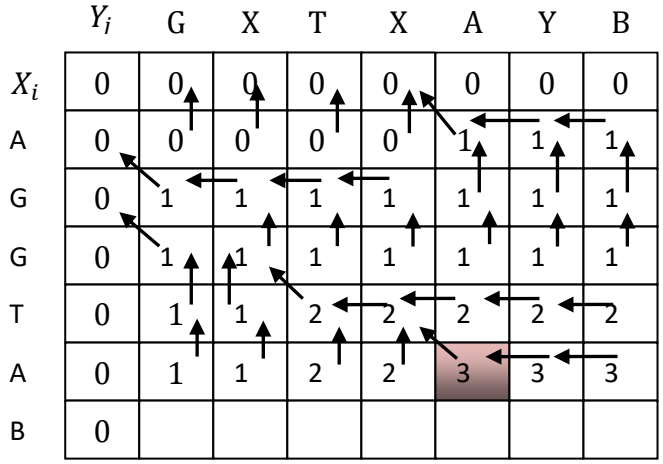
Iteration 3:



Iteration 4:

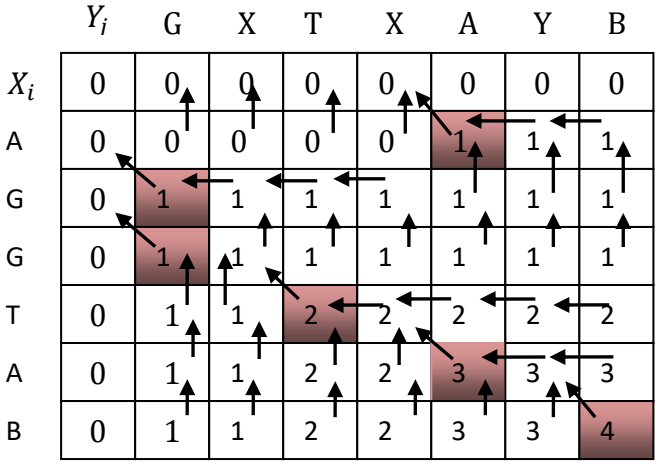


Iteration 5:



X_i \ Y_i		G	X	T	X	A	Y	B
	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1
T	0	1	1	2	2	2	2	2
A	0	1	1	2	2	3	3	3
B	0							

Iteration 6:



X_i \ Y_i		G	X	T	X	A	Y	B
	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1
T	0	1	1	2	2	2	2	2
A	0	1	1	2	2	3	3	3
B	0	1	1	2	2	3	3	4

Hence length of Longest common subsequence is 4 and the LCS can be obtained using backtracking i. e. **GTAB**

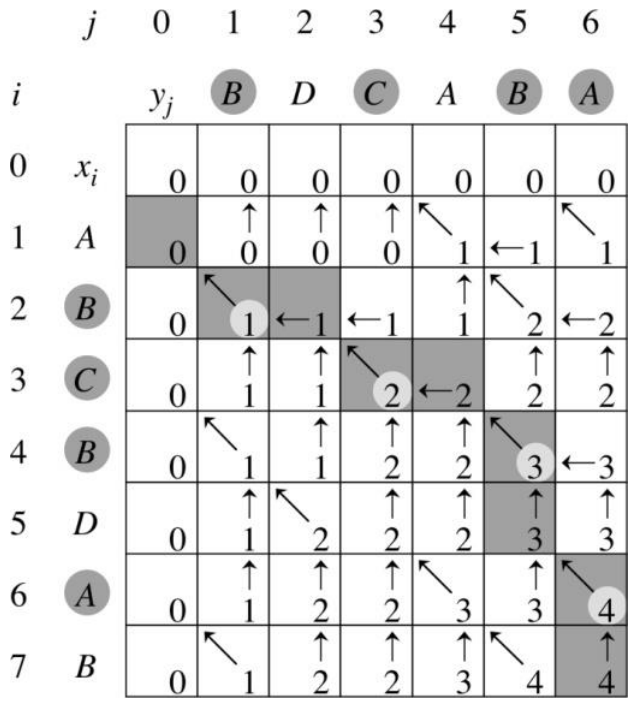
Example:

Find the LCS of the following two strings:

X: B D C A B A

Y: A B C B D A B

Solution:



i		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Hence length of Longest common subsequence is 4 and the LCS can be obtained by backtracking i. e. **BCBA**

Algorithm LCS:

By using this algorithm we can find length of the longest common subsequence of two strings X and Y.

```
LCS-Length(X, Y)
{
    m = length(X)                    // get the No. of symbols in X
    n = length(Y)                    // get the No. of symbols in Y
    for i= 1 to m do
        c[i, 0] = 0;                 // special case: Y0
    for j = 1 to n do
        c[0, j] = 0;                 // special case: X0
    for i= 1 to m do                 // for all Xi
    {
        for j = 1 to n do            // for all Yj
        {
            if (Xi== Yj) then
                c[i,j] = c[i-1, j-1] + 1;
            else
                c[i,j] = max( c[i-1, j], c[i, j-1] )
        }
```

```
        }
        return c
}
```

Time Complexity:

Time Complexity of the algorithm = Time Complexity for initializing the table +Time complexity for filling the table in buttom-up manner = $O(m+n)+O(mn)=O(mn)$.

Algorithm to print LCS

```
PRINT-LCS (b, X, i, j)
{
        if (i ==0 or, j==0) then
                return;
        if (b[i, j] ==”↖”) then
        {
                PRINT-LCS (b, X, i − 1, j − 1);
                Print x_i;
        }
        else if (b[i, j] ==”↑”) then
                PRINT-LCS (b, X, i − 1, j);
        else
                PRINT-LCS (b, X, i, j − 1);
}
```

Time Complexity:

Time Complexity of the algorithm = $O(m+n)$

STRING MATCHING ALGORITHMS

String matching or, String searching algorithms are computational methods to locate specific patterns within texts or, string of characters. These patterns could be simple sequences of characters, substrings, regular expressions, or more complex structures.

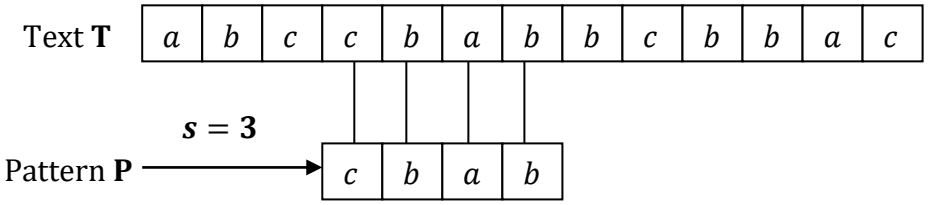
The primary goal of string matching algorithms is to efficiently identify occurrences of a given pattern within a larger text or string.

Given a text array, $T\ [1\ldots.n]$, of n character and a pattern array, $P\ [1\ldots\ldots m]$, of m characters, where P and T are characters drawn from a finite alphabet $\sum$.

$\sum = \{0,1\}$ or, $\sum = \{a, b, \ldots, z\}$

The problems are to find an integer s, called **valid shift** where

1. $0 \leq s < n - m$ and
2. $T\ [s + 1\ldots\ldots s + m] = P\ [1\ldots\ldots m]$.



If **P** occurs with shift **s** in **T**, then we call **s** a valid shift; otherwise, we call **s** an invalid shift.

The string matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T .

There are five major string matching Algorithms:

1. Naive String Matching Algorithm
2. Rabin-Karp-Algorithm
3. Finite Automata
4. Knuth-Morris-Pratt Algorithm
5. Boyer-Moore Algorithm

The Naive String Matching Algorithm

This is one of the easy pattern-matching algorithms. All the characters of the pattern are matched to the characters of the corresponding/main string or text.

In this approach we test all the possible placements of input pattern $[1 \ldots \ldots m]$ relative to input text $[1 \ldots \ldots n]$. We then try to shift $s = 0, 1 \ldots \ldots n - m$, successively and for each shift s, we will compare both the input and pattern string.

Algorithm:

NAIVE-STRING-MATCHER (T, P)

```
{
        n = length [T];
        m =length [P];
        for s = 0 to n –m do
        {
                if (P [1.....m] == T [s + 1....s + m])then
                        print "Pattern occurs with shift" s;
        }
}
```

Time Complexity:

The for-loop is executed $n - m + 1$ times and to match m characters of the pattern P, it takes m time. Therefore the time complexity of the algorithm is $O((n - m + 1)m)$.

The 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.

The basic idea is to convert the pattern and each possible substring of the text into numeric values (hashes) and then compare these values rather than the strings themselves. This allows for faster comparisons, especially when dealing with large texts.

Working Procedure:

1. *Hashing the Pattern:* First, the algorithm turns the pattern into a numeric value using a hash function. This number is like a unique ID for the pattern.
2. *Hashing the Text:* Next, the algorithm takes the first part of the text that is the same length as the pattern and turns it into a number using the same hash function.
3. *Comparing Hashes:* The algorithm compares the number (hash) of the pattern with the number (hash) of the part of the text. If the numbers are the same, it means the pattern might be there. If the numbers are different, the pattern is definitely not there.
4. *Sliding the Window:* If the numbers don't match, the algorithm slides over to the next part of the text, creates a new hash for this part, and compares again. This sliding continues until the whole text is checked.
5. *Collision Check:* Sometimes, different parts of the text can have the same hash number even if they are not the same as the pattern. When the hash numbers

match, the algorithm checks the actual text to make sure it really found the pattern.

Key concepts of Rabin-Karp-Algorithm:

Hash Function:

A hash function is a mathematical function that converts a string into a fixed-size numeric value, which is called a hash.

$$H = c_1 a^{k-1} + c_2 a^{k-2} + \cdots + c_k a^0$$

where a is a constant, $c_1, c_2, \ldots c_k$ are input characters and k is the number of characters that are in the string we are comparing.

Sliding Window Technique:

The sliding window technique is a method used by the Rabin-Karp algorithm to move through the text efficiently while updating the hash value.

Imagine you have a window that covers a substring of the text that is the same length as the pattern. You start by calculating the hash for the first substring within this window. Then, you slide the window one character to the right to cover the next substring.

Instead of recalculating the hash from scratch for each new substring, the algorithm uses the hash of the previous substring and updates it by removing the effect of the first character and adding the effect of the new character.

For example, if our current window covers "abc", and we move the window to cover "bcd", you can quickly update the hash by subtracting the effect of 'a' and adding the effect of 'd'.

Suppose in the hash function $a = 26, k = 3$ and c is place in the alphabet where character appears. So for “a” c will be 1, for “b” c will be 2 and so on.

$$H(\text{“}abc\text{”}) = 1 \times 26^2 + 2 \times 26^1 \times 3 \times 26^0$$

To get the hash of “bcd”, we need to remove “a” and add “d”:

$$H(\text{“}bcd\text{”}) = H(\text{“}abc\text{”}) - H(\text{“}a\text{”}) + H(\text{“}d\text{”})$$

$$H(\text{“}bcd\text{”}) = ((1 \times 26^2 + 2 \times 26^1 \times 3 \times 26^0) - 1 \times 26^2) \times 26 + 4 \times 26^0$$

Spurious hit

When the hash value of the pattern matches with the hash value of a window of the text but the window is not the actual pattern then it is called a spurious hit.

Spurious hit increases the time complexity of the algorithm. In order to minimize spurious hit, we use modulus. It greatly reduces the spurious hit.

Algorithm:

***RABIN−KARP−MATCHER*(*T*, *P*, *d*, *q*)**

```
{
    n = Length[T];
    m = Length[P];
    h = d^(m−1) mod q;
    p = 0;
    t0 = 0;
    for i = 1 to m do
    {
        p = (d * p + P[i]) mod q;
        t0 = (d * t0 + T[i]) mod q;
    }
    for s = 0 to n − m do
    {
        if p = ts then
        {
            if P[1..m] == T[s + 1..s + m] then
                "Pattern founds at shift " s;
        }
        if s < n − m then
            ts+1 = (d(ts − T[s + 1]h) + T[s + m + 1]) mod q;
    }
}
```

Time Complexity:

In this algorithm, the hash value of the pattern is calculated in O(m) time and the traversal of the given string for the calculation of the hash value and comparing the corresponding hash value with that of the pattern is done in O(n) time.

Therefore, the time complexity of the Rabin-Karp Algorithm is $O(m + n)$, where 'm' and 'n' are the lengths of the given pattern and string respectively.

Example:

Text **T**

A	*S*	*H*	*O*	*K*	*P*	*A*	*N*	*D*	*A*

Pattern **P**

P	*A*	*N*

Solution:

Alphabets:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
Code:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Length of the text $n = 10$, Length of the pattern $m = 3$.
Total number of windows to be matched $= n - m + 1 = 10 - 3 + 1 = 8$
Here we have to use each window of size 3.
Since $d = 19 \neq 10$, computation of modulus is not required.
Hash value of the pattern $P = 16 \times 19^2 + 1 \times 19^1 + 14 \times 19^0 = 5809$

Now we have to calculate hash value of each window of size $m = 3$

1. Calculation of hash value of first window ASH:
$$1 \times 19^2 + 19 \times 19^1 + 8 \times 19^0 = 730 \neq 5809$$
2. Calculation of hash value of second window SHO:
$$19 \times 19^2 + 8 \times 19^1 + 15 \times 19^0 = 7026 \neq 5809$$
3. Calculation of hash value of third window HOK:
$$8 \times 19^2 + 15 \times 19^1 + 11 \times 19^0 = 3184 \neq 5809$$
4. Calculation of hash value of fourth window OKP:
$$15 \times 19^2 + 11 \times 19^1 + 16 \times 19^0 = 5640 \neq 5809$$
5. Calculation of hash value of fifth window KPA:
$$11 \times 19^2 + 16 \times 19^1 + 1 \times 19^0 = 4276 \neq 5809$$
6. Calculation of hash value of sixth window PAN:
$$16 \times 19^2 + 1 \times 19^1 + 14 \times 19^0 = 5809 = 5809$$
Now we will start matching the characters of the pattern with the characters of the window. **Exact Match**
7. Calculation of hash value of seventh window AND:
$$1 \times 19^2 + 14 \times 19^1 + 4 \times 19^0 = 631 \neq 5809$$

8. Calculation of hash value of tenth window NDA:
$$14 \times 19^2 + 4 \times 19^1 + 1 \times 19^0 = 5131 \neq 5809$$

Example:
For string matching, working module $q = 11$, how many spurious hits does the Rabin-Karp matcher encounters in Text $T = 3141592653589793$, when looking for the pattern $P = 26$?

Solution:

Text **T**	3	1	4	1	5	9	2	6	5	3	5

Pattern **P**	2	6

Length of the text $n = 11$, Length of the pattern $m = 2$.
Total number of windows to be matched $= n - m + 1 = 11 - 2 + 1 = 10$
Here we have to use each window of size 2.
Hash value of the pattern $= P \ mod \ q = 26 \ mod \ 11 = 4$.
Now we have to calculate hash value of each window of size $m = 2$

1. Calculation of hash value of first window 31:
$$31 \ mod \ 11 \ = \ 9 \ \neq \ 4$$
2. Calculation of hash value of 2nd window 14:
$$14 \ mod \ 11 \ = \ 3 \ \neq \ 4$$
3. Calculation of hash value of 3rd window 41:
$$41 \ mod \ 11 \ = \ 8 \ \neq \ 4$$
4. Calculation of hash value of 4th window 15:
$$15 \ mod \ 11 \ = \ 4$$
Now we will start matching the characters of the pattern with the characters of the window. No matching, hence spurious hit.
5. Calculation of hash value of 5th window 59:
$59 \ mod \ 11 \ = \ 4 = \ 4$ → Spurious hit
6. Calculation of hash value of 6th window 92:
$92 \ mod \ 11 \ = \ 4 \ = \ 4$ → Spurious hit
7. Calculation of hash value of 7th window 26:
$26 \ mod \ 11 \ = \ 4 \ = \ 4$ → Exact match
8. Calculation of hash value of 8th window 65:
$65 \ mod \ 11 \ = \ 10 \ \neq \ 4$.
9. Calculation of hash value of 9th window 53:
$53 \ mod \ 11 \ = \ 9 \neq \ 4$.
10. Calculation of hash value of 10th window 35:
$35 \ mod \ 11 \ = \ 2 \neq \ 4$.

Thus total number of spurious hits $= 3$.

Example:

Working modulo q = 997, how many spurious hits does the Rabin Karp matcher encounter in the text T = 3141592653589793 when looking for the pattern P = 26535.

Solution:

$n = 16, m = 5, q = 997$

Hash value of the pattern $= 26335 \; mod \; 997 = 613$

Total number of windows to be matched $= n - m + 1 = 16 - 5 + 1 = 12$

1. $31415 \; mod \; 997 = 508$
2. $14159 \; mod \; 997 = 201$
3. $41592 \; mod \; 997 = 715$
4. $15926 \; mod \; 997 = 971$
5. $59265 \; mod \; 997 = 442$
6. $92653 \; mod \; 997 = 929$
7. $26535 \; mod \; 997 = 613$ **Exact Match**.
8. $65358 \; mod \; 997 = 553$
9. $53589 \; mod \; 997 = 748$
10. $35897 \; mod \; 997 = 5$
11. $58979 \; mod \; 997 = 156$
12. $89793 \; mod \; 997 = 63$

No spurious hits.

Problem:

Explain spurious hits in Rabin-Karp string matching algorithm. Working modulo q = 13, how many spurious hits does the Rabin Karp matcher encounter in the text T = 2359023141526739921 when looking for the pattern P = 31415?