Syllabus

Binary Search- Introduction, Applications: Median of two sorted arrays, Find the fixed point in a given array, Find Smallest Common Element in All Rows, Longest Common Prefix, Koko Eating Bananas.

Greedy Method: General method – Applications –Minimum product subset of an array, Best Time to Buy and Sell Stock, Knapsack problem, Minimum cost spanning trees, Single source shortest path problem.

Backtracking: General method, Applications: N Queens Problem, Hamiltonian Cycle, Brace Expansion, Gray Code, Path with Maximum Gold, Generalized Abbreviation, Campus Bikes II.

I. Binary Search

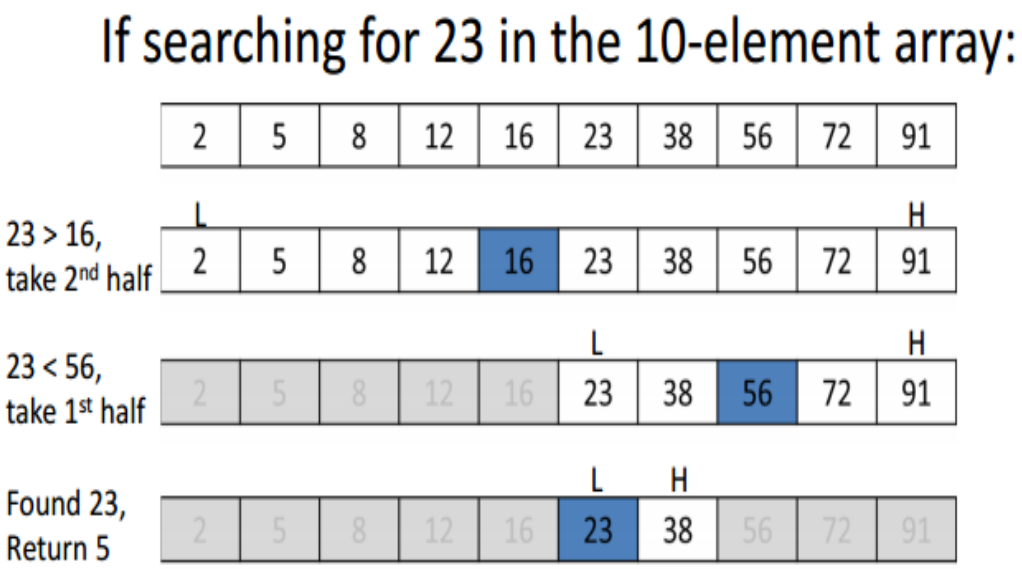
Introduction:

- Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].
- A simple approach is to do **Linear Search.**
- The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.
- A binary search or half-interval search algorithm finds the position of a specified value (the input "key") within a sorted array.
- In each step, the algorithm compares the input key value with the key value of the middle element of the array.
- If the keys match, then a matching element has been found so its index, or position, is returned.
- Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the input key is greater, on the sub-array to the right.
- If the remaining array to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.

- Every iteration eliminates half of the remaining possibilities. This makes binary searches very efficient - even for large collections.
- Binary search requires a sorted collection. Also, binary searching can only be applied to a collection that allows random access (indexing).

  Worst case performance: O(log n)

  Best case performance: O(1)



Working:

Binary Search Algorithm can be implemented in two ways which are discussed below.

1. Recursive Method
2. Iterative Method

1. Recursive Algorithm (Divide and Conquer Technique)

Algorithm BinSrch (a, low, high, x)

```
//Given an array a [ low : high ]of elements in increasing
//order,1≤low≤high,determine whether x is present, and
//if so, return j such that x=a[j]; else return 0.
{
    if( low = =high ) then   // If small(P)
    {
            if( x=a[low] ) then return low;
            else return 0;
    }
    else
    {
          //Reduce p into a smaller subproblem.
               mid:=  (low+high)/2
               if( x = a[mid] ) then return mid;
               else if ( x<a[mid] ) then
               return BinSrch(a, low, mid-1, x);
               else
            return BinSrch(a, mid+1, high, x);
     }
}
```

2. Iterative Algorithm (Divide and Conquer Technique)

```
Algorithm BinSearch(a, n, x)
// a is an array of size n, x is the key element to be searched.
{       low:=1;   high:=n;
        while( low ≤ high)
        {
                mid:=(low+high)/2;
               if (x==a[mid])
               {
                  return     mid;
               }
                if( x < a[mid]  )  then  high := mid-1;
              else  low := mid+1;
    }
   return 0;
}
```

Time complexity of Binary Search

- If the time for diving the list is a constant, then the computing time for binary search is described by the recurrence relation.

$$T(n) = \begin{cases} c_1 & n=1,\ c1 \text{ is a constant} \\ T(n/2) + c_2 & n>1,\ c2 \text{ is a constant} \end{cases}$$

Assume $n=2^k$, then

$T(n) = T(n/2) + c2$
$=T(n/4)+c2+c2$
$=T(n/8) +c2+c2+c2$
…..
…..
$= T(n / 2^k) + c2+c2+c2+$ ……..k times
$= T(1)+ kc2$
$= c1+kc2 =c1+ \log n*c2 = O(\log n)$

Successful searches:

best	average	worst
O(1)	O(log n)	O(log n)

Unsuccessful searches :

best	average	worst
O(log n)	O(log n)	O(log n)

Program for Iterative binary search : BinarySearch_iterative.java

```
import java.util.*;
class BinarySearch_iterative
{
  int binarySearch(int array[], int x, int low, int high)
  {

    // Repeat until the pointers low and high meet each other
    while (low <= high)
    {
      int mid = low + (high - low) / 2;

      if (array[mid] == x)
        return mid;

      if (array[mid] < x)
        low = mid + 1;

      else
        high = mid - 1;
    }

    return -1;
  }

  public static void main(String args[])
  {
BinarySearch_iterative ob = new BinarySearch_iterative ( );

      Scanner sc=new Scanner(System.in);

          System.out.println("enter array size");

            int n = sc.nextInt();

               int array[]=new int[n];

          System.out.println("enter the elements of array ");

      for(int i=0;i<n;i++)
          {
                array[i] =sc.nextInt();
       }
      // Applying sort() method over to above array by passing the array as an argument
      Arrays.sort(array);
```

```
        // Printing the array after sorting
        System.out.println("sorted array[]:"+ Arrays.toString(array));

            System.out.println("Enter the key");
    int key=sc.nextInt();

    int result = ob.binarySearch(array, key, 0, n - 1);
    if (result == -1)
      System.out.println("Not found");
    else
      System.out.println("Element found at index " + result);
  }
}
```

Output:

Case-1:

```
enter array size
5
enter the elements of array
33
65
32
68
95
sorted array[]:[32, 33, 65, 68, 95]
Enter the key
59
Not found
```

Case-2:

```
enter array size
5
enter the elements of array
33
65
32
68
95
sorted array[]:[32, 33, 65, 68, 95]
enter array the key
68
Element found at index 3
```

Program for Recursive binary search :

BinarySearch_recursive.java

```
import java.util.*;
class BinarySearch_recursive
{
  int binarySearch(int array[], int x, int low, int high) {

    if (high >= low) {
      int mid = low + (high - low) / 2;

      // If found at mid, then return it
      if (array[mid] == x)
        return mid;

      // Search the left half
      if (array[mid] > x)
        return binarySearch(array, x, low, mid - 1);

      // Search the right half
      return binarySearch(array, x, mid + 1, high);
    }

    return -1;
  }

  public static void main(String args[])
  {
    BinarySearch_recursive ob = new BinarySearch_recursive();

        Scanner sc=new Scanner(System.in);

        System.out.println("enter array size");

          int n = sc.nextInt();

       int array[]=new int[n];

  System.out.println("enter the elements of array ");

        for(int i=0;i<n;i++)
      {
```

```
        array[i] =sc.nextInt();
       }
      // Applying sort() method over to above array
      // by passing the array as an argument
      Arrays.sort(array);

      // Printing the array after sorting
      System.out.println("sorted array[]:"+ Arrays.toString(array));
 System.out.println("Enter the key");
  int key=sc.nextInt();


  int result = ob.binarySearch(array, key, 0, n - 1);
  if (result == -1)
    System.out.println("Element Not found");
  else
    System.out.println("Element found at index " + result);
 }
```

Output:

Case=1

```
enter array size5
enter the elements of array 15
35
25
95
65
sorted array[]:[15, 25, 35, 65, 95]
Enter the key
65
Element found at index 3
```

Case=2

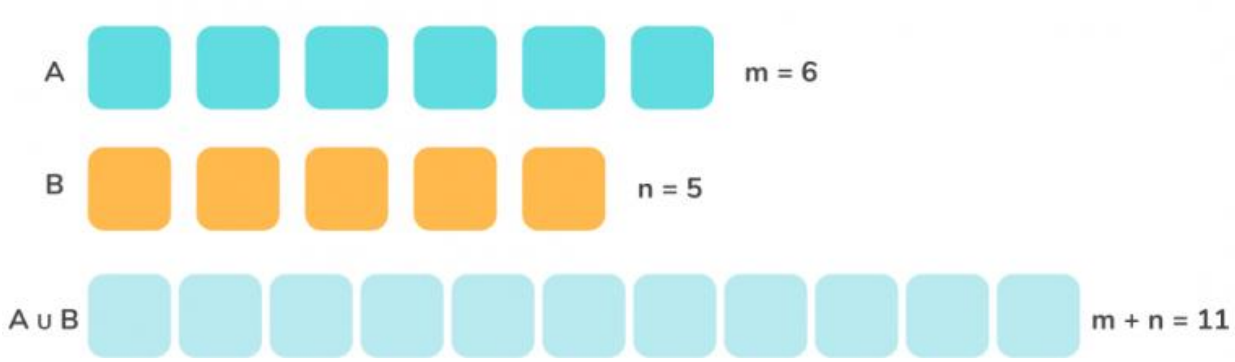
```
enter array size5
enter the elements of array 15
35
25
95
65
sorted array[]:[15, 25, 35, 65, 95]
Enter the key
30
Element Not found
```

Applications:

1. **Median of two sorted arrays.**
2. **Find the fixed point in a given array.**
3. **Find Smallest Common Element in All Rows.**
4. **Longest Common Prefix.**
5. **Koko Eating Bananas.**

1. Median of two sorted arrays.

- There are two sorted arrays **A** and **B** of sizes **m** and **n** respectively.
- Find the median of the two sorted arrays(The median of the array formed by merging both the arrays).
- **Median:** The middle element is found by ordering all elements in sorted order and picking out the one in the middle (or if there are two middle numbers, taking the mean of those two numbers).



Examples:

Input: A[] = {1, 4, 5}, B[] = {2, 3}
Output: 3
Explanation:
Merging both the arrays and arranging in ascending:
[1, 2, 3, 4, 5]
Hence, the median is 3

Input: A[] = {1, 2, 3, 4}, B[] = {5, 6}
Output: 3.5
Explanation:
Union of both arrays:
{1, 2, 3, 4, 5, 6}
Median = (3 + 4) / 2 = 3.5

Constraints:

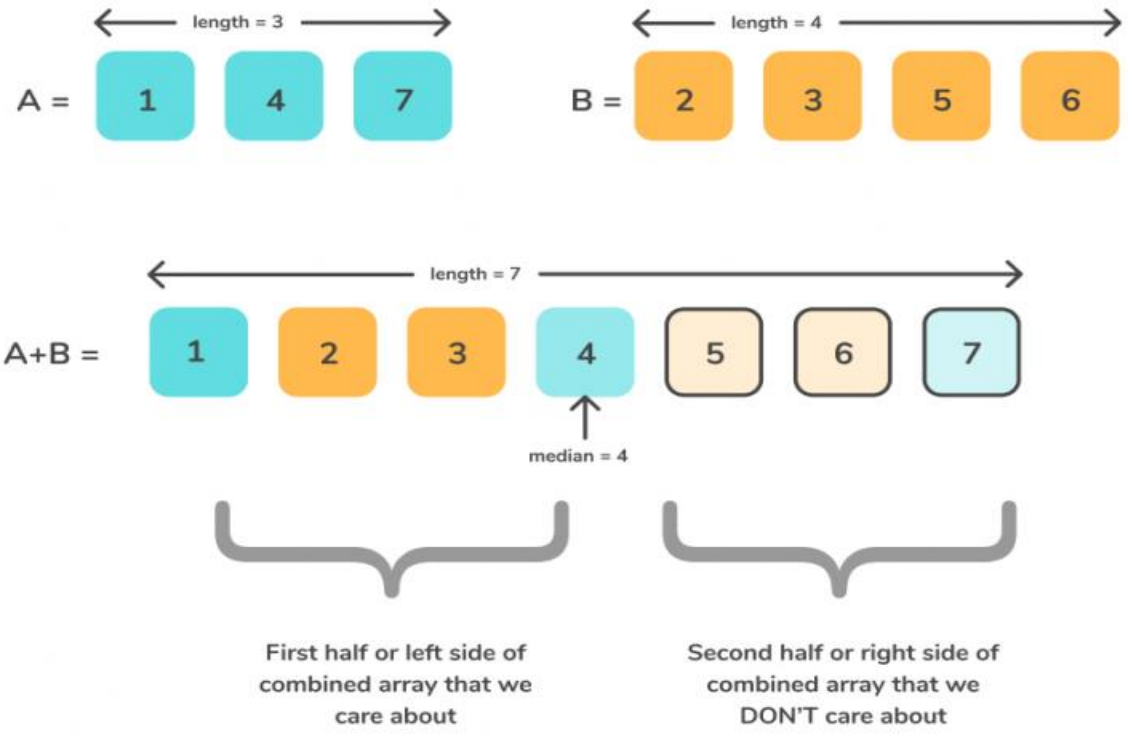
- **nums1.length == m**
- **nums2.length == n**
- **0 <= m <= 1000**
- **0 <= n <= 1000**
- **1 <= m + n <= 2000**
- **-10^6 <= nums1[i], nums2[i] <= 10^6**

The overall run time complexity should be O(log (m+n)).

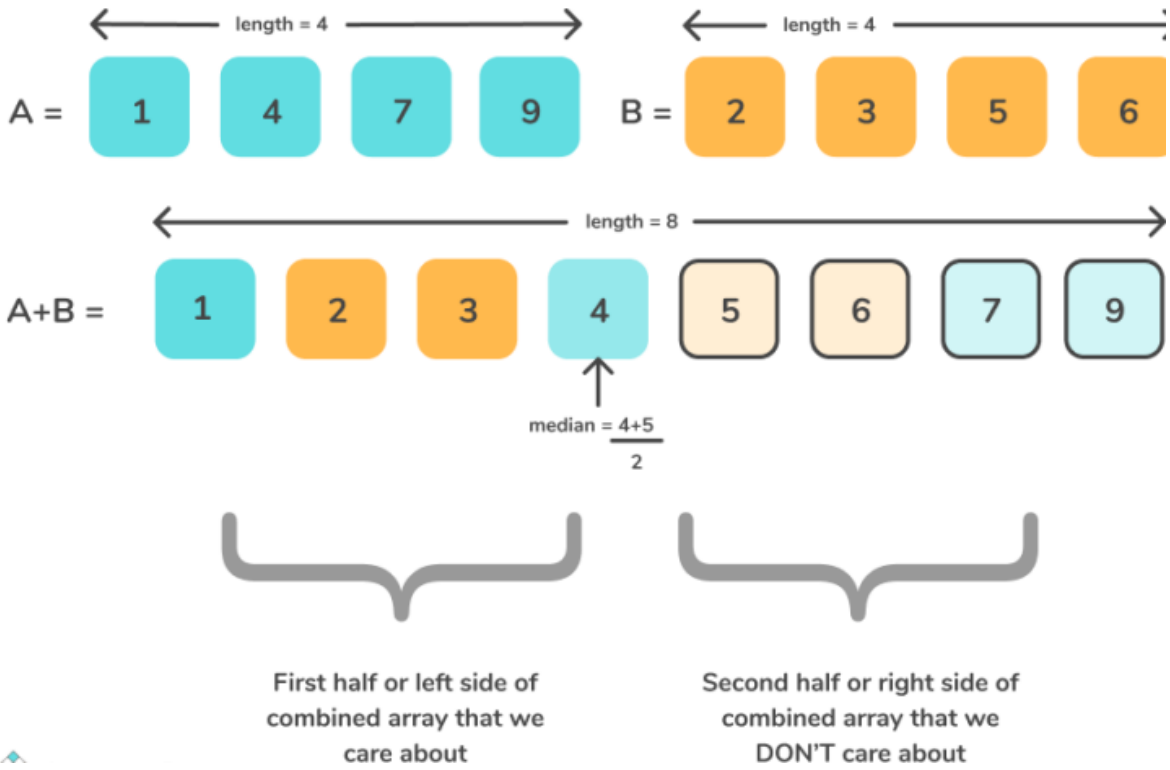
Using Binary search:

The key idea to note here is that both the arrays are **sorted**. Therefore, this leads us to think of **binary search**. Let us try to understand the algorithm using an example:

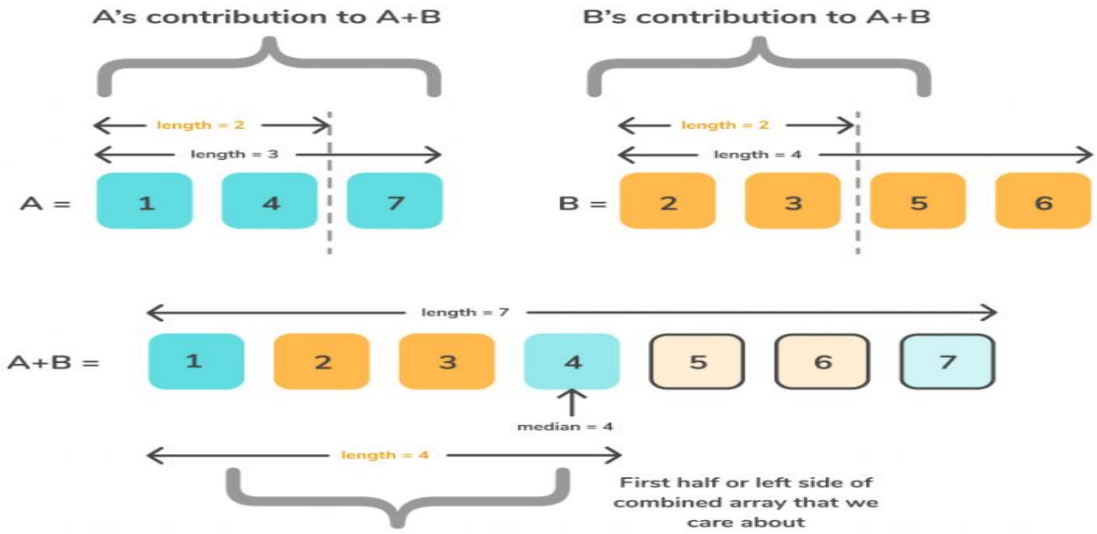
A[ ] = {1, 4, 7} **B[ ]** = {2, 3, 5, 6}



From the above diagram, it can be easily deduced that only the **first half** of the array is needed and the **right half** can be discarded.

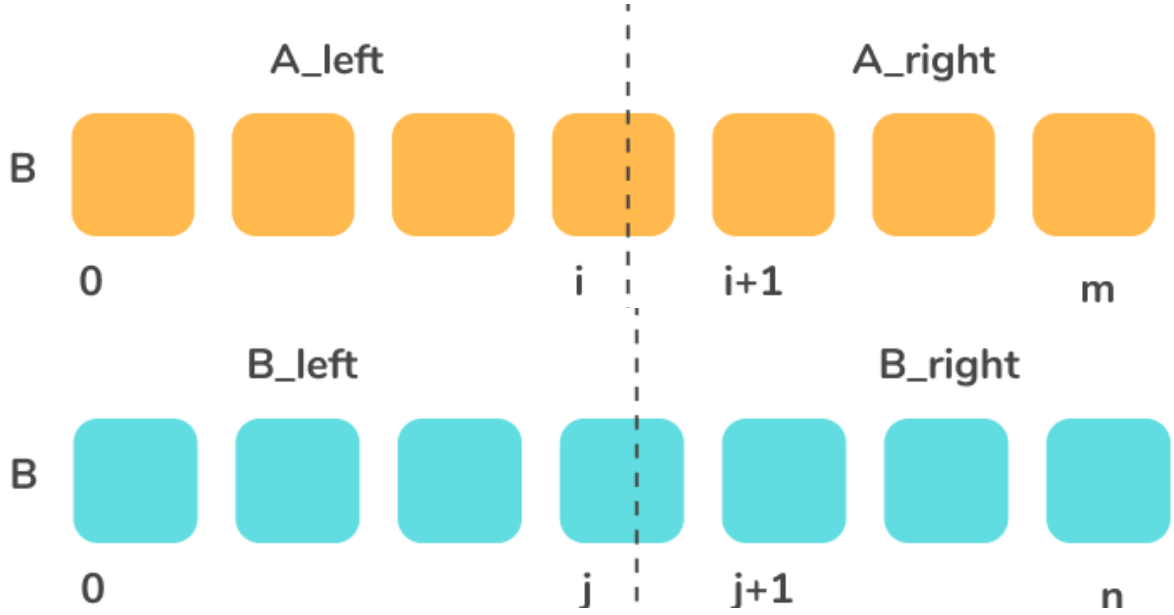


- Similarly, for an even length merged array, ignore the **right half** and only the **left half** contributes to our final answer.
- Therefore, the motive of our approach is to find which of the elements from both the array helps in contributing to the final answer. Therefore, the **binary search** comes to the rescue, as it can discard a part of the array every time, the elements don't contribute to the median.

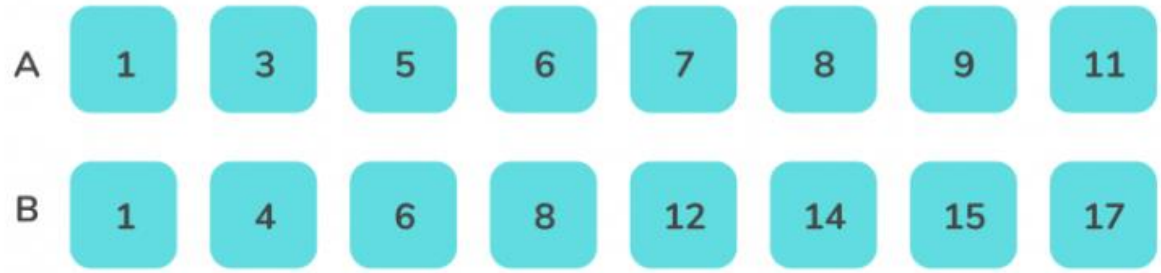


Algorithm

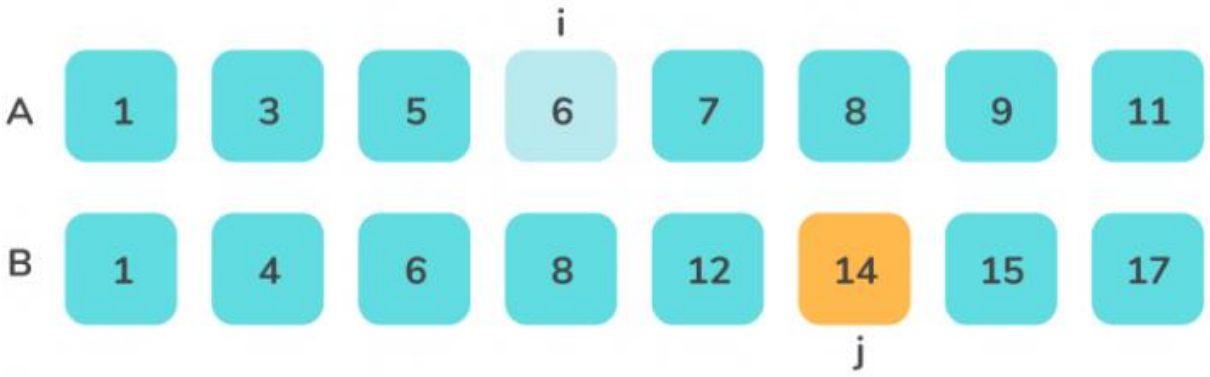
- The first array is of size **n**, hence it can be split into **n + 1** parts.
- The second array is of size **m**, hence it can be split into **m + 1** parts



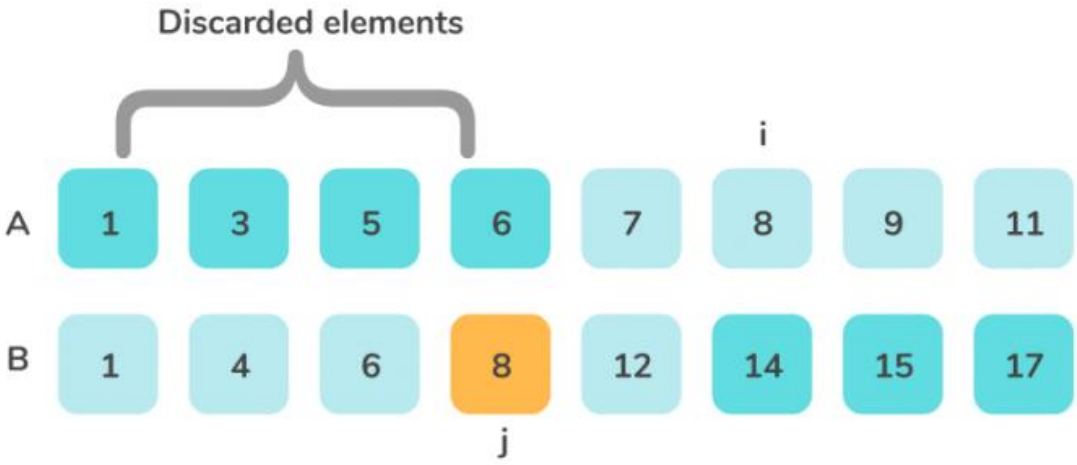
- As discussed earlier, we just need to find the elements contributing to the left half of the array.
  - Since, the arrays are already sorted, it can be deduced that
    A[i – 1] < A[i] and **B[i – 1] < B[i]**.
  - Therefore, we just need to find the index **i**, such that
    A[i – 1] <= B[j] and **B[j – 1] <= A[i].**
- Consider **mid = (n + m – 1) / 2** and check if this satisfies the above condition.
  - If **A[i-1] <= B[j]** and **B[j-1]<=A[i]** satisfies the condition, return the index **i**.
  - If **A[i] < B[j – 1]**, increase the range towards the right. Hence update **i = mid + 1**.
  - Similarly, if **A[i – 1] > B[j]**, decrease the range towards left. Hence update **i = mid – 1**.



i is the mid of array **A** as shown below:



A[i-1] = 5 b[j-1] = 12, since B[j-1] > A[i], needs to increase



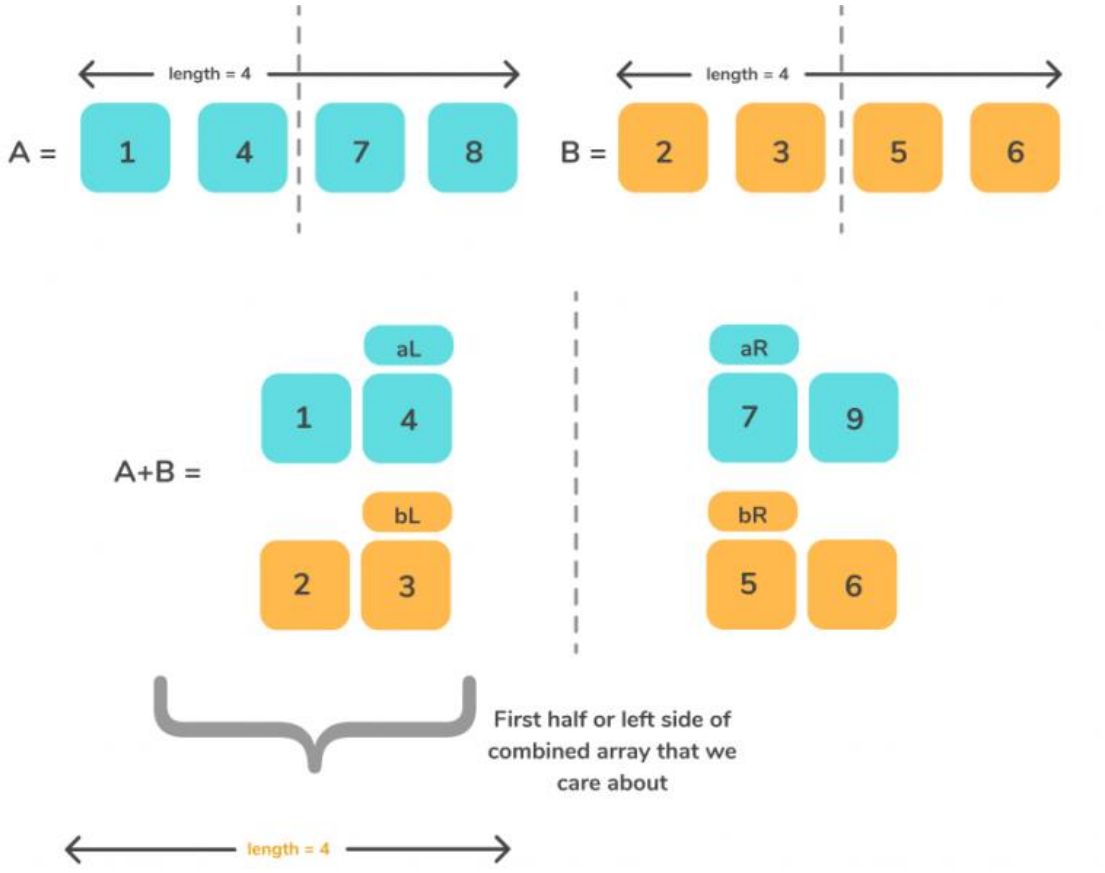
Max of left side will be either A[i-1] or B[j-1], in this case its A[i-1] = 7

Min of right will be either A[i] or B[j], in this case both are equal which is 8

Since, sum of length is even, we return average of these two which is 7.5

Few corner cases to take care of :

- If the size of any of the arrays is **0**, return the median of the non-zero sized array.
- If the size of smaller array is **1**,:
  - If the size of the larger array is also one, simply return the median as the mean of both the elements.
  - Else, if size of larger array is odd, adding the element from first array will result in size even, hence median will be affected if and only if, the element of the first array lies between **, M / 2th** and **M/2 + 1**th element of **B[].**
  - Similarly, if the size of the larger array is even, check for the element of the smaller array, **M/2th element** and **M / 2 + 1 th** element.
- If the size of smaller array is **2**,
  - If a larger array has an odd number of elements, the median can be either the **middle** element **or** the median of elements of smaller array and **M/2 - 1th** element or minimum of the second element of **A[]** and **M/2+ 1** th array.



Java program for Median Of Two Sorted Array(Binary search approach)

MedianOfTwoSortedArrays.java

```
import java.util.*;
public class MedianOfTwoSortedArrays {

   private static double findMedianSortedArrays(int[] nums1, int[] nums2)
   {
 // Check if num1 is smaller than num2 If not, then we will swap num1 with num2
      if (nums1.length > nums2.length)
      {
         return findMedianSortedArrays(nums2, nums1);
      }
      // Lengths of two arrays
      int m = nums1.length;
      int n = nums2.length;
      // Pointers for binary search
      int start = 0;
      int end = m;
      // Binary search starts from here
      while (start <= end)
        {
         // Partitions of both the array
         int partitionNums1 = (start + end) / 2;
         int partitionNums2 = (m + n + 1) / 2 - partitionNums1;

// Edge cases there are no elements left on the left side after partition

int maxLeftNums1 = partitionNums1 == 0 ? Integer.MIN_VALUE :nums1[partitionNum
s1 - 1];

// If there are no elements left on the right side after partition
 int minRightNums1 = partitionNums1 == m ? Integer.MAX_VALUE : nums1[partition
Nums1];

 // Similarly for nums2
int maxLeftNums2 = partitionNums2 == 0 ? Integer.MIN_VALUE :nums2[partitionNum
s2 - 1];

int minRightNums2 = partitionNums2 == n ? Integer.MAX_VALUE :nums2[partitionNu
ms2]
```

```
        // Check if we have found the match

    if (maxLeftNums1 <= minRightNums2 && maxLeftNums2 <= minRightNums1)
                {
            // Check if the combined array is of even/odd length
            if ((m + n) % 2 == 0)
                  {
                 return (Math.max(maxLeftNums1, maxLeftNums2)+Math.min(minRightN
       ums1, minRightNums2)) / 2.0;
            }
       else {
              return Math.max(maxLeftNums1, maxLeftNums2);
            }
        }
        // If we are too far on the right, we need to go to left side
        else if (maxLeftNums1 > minRightNums2) {
            end = partitionNums1 - 1;
        }
        // If we are too far on the left, we need to go to right side
        else {
            start = partitionNums1 + 1;
        }
    }
    // If we reach here, it means the arrays are not sorted
    throw new IllegalArgumentException();
}

  public static void main(String[] args)
  {

     Scanner sc=new Scanner(System.in);

     System.out.println("enter the  size of the 1st array");

       int n = sc.nextInt();

      int nums1[]=new int[n];

     System.out.println("enter the elements of 1st array ");

     for(int i=0;i<n;i++)
     {
       nums1[i] =sc.nextInt();
```

```
        }
    System.out.println("enter the  size of the 2nd array");

         int m = sc.nextInt();

        int nums2[]=new int[m];

     System.out.println("enter the elements of 2nd array ");

       for(int i=0;i<m;i++)
       {
          nums2[i] =sc.nextInt();
        }


       System.out.println("The Median of two sorted arrays is :" + findMedianSortedArray
s(nums1, nums2));


   }
}
```

Output:

```
enter the  size of the 1st array
4
enter the elements of 1st array
1
4
7
8
enter the  size of the 2nd array
4
enter the elements of 2nd array
2
3
5
6
The Median of two sorted arrays is :4.5
```

2. Find the fixed point in a given array.

- Given an array of n distinct integers sorted in ascending order, write a function that returns a Fixed Point in the array, if there is any Fixed Point present in array, else returns -1.
- Fixed Point in an array is an index i such that arr[i] is equal to i. Note that integers in array can be negative.

Example 1:
Input: [-10,-5,0,3,7]
Output: 3
Explanation:
For the given array, A[0] = -10, A[1] = -5, A[2] = 0, A[3] = 3, thus the output is 3.

Example 2:
Input: [0,2,5,8,17]
Output: 0
Explanation:
A[0] = 0, thus the output is 0.

Example 3:
Input: [-10,-5,3,4,7,9]
Output: -1
Explanation:
There is no such i that A[i] = i, thus the output is -1.

Note:
1 <= A.length < 10^4
-10^9 <= A[i] <= 10^9

Algorithm

The basic idea of binary search is to divide n elements into two roughly equal parts, and compare a[n/2] with x.

- If x=a[n/2], then find x and the algorithm stops;
- if x<a[n/2], as long as you continue to search for x in the left half of array a,
- if x>a[n/2], then as long as you search for x in the right half of array a.

Java program for Find the Fixed point in an array:

Findfixedpoint.java

```
import java.util.*;
class Findfixedpoint
{
   static int fixedpoint(int arr[], int low, int high)
   {
      if (high >= low) {
         int mid = low + (high - low) / 2;
         if (mid == arr[mid])
            return mid;
         int res = -1;
         if (mid + 1 <= arr[high])
            res = fixedpoint(arr, (mid + 1), high);
         if (res != -1)
            return res;
         if (mid - 1 >= arr[low])
            return fixedpoint(arr, low, (mid - 1));
      }

      /* Return -1 if there is no Fixed Point */
      return -1;
   }

   // main function
   public static void main(String args[])
   {

       Scanner sc=new Scanner(System.in);
       System.out.println("enter array size");
       int n = sc.nextInt();
        int array[]=new int[n];

  System.out.println("enter the elements of array ");

      for(int i=0;i<n;i++)
     {
        array[i] =sc.nextInt();
```

```
        }

        Arrays.sort(array);

        // Printing the array after sorting
        System.out.println("sorted array[]:"+ Arrays.toString(array));


        System.out.println("Fixed Point is " + fixedpoint(array, 0, n - 1));
    }
}
```

Output:

case=1

```
enter array size
10
enter the elements of array
11 30 50 0 3 100 -10 -1 10 102
sorted array[]:[-10, -1, 0, 3, 10, 11, 30, 50, 100, 102]
Fixed Point is 3
```

case=2

```
enter array size
6
enter the elements of array
3 9 4 7 -5 -10
sorted array[]:[-10, -5, 3, 4, 7, 9]
Fixed Point is -1
```

case=3

```
enter array size
5
enter the elements of array
8 2 5 17 0
sorted array[]:[0, 2, 5, 8, 17]
Fixed Point is 0
```

3. Find Smallest Common Element in All Rows.

Given a matrix where every row is sorted in increasing order. Write a function that finds and returns a smallest common element in all rows. If there is no common element, then returns **-1**.

Example-1:
Input: mat [4][5] = { {1, 2, 3, 4, 5},
{2, 4, 5, 8, 10},
{3, 5, 7, 9, 11},
{1, 3, 5, 7, 9}
};
Output: 5

Time complexity:

- A **O(m*n*n) simple solution** is to take every element of first row and search it in all other rows, till we find a common element.
- Time complexity of this solution is O(m*n*n) where m is number of rows and n is number of columns in given matrix.
- This can be improved to **O(m*n*logn)** if we **use Binary Search** instead of linear search.

Program: `SmallestCommonElement.java`

```
import java.util.*;

class SmallestCommonElement
{
   private boolean binarySearch(int[] arr, int low, int high, int target)
{
      while(low <= high) {
         int mid = (low + high)/2;
         if(arr[mid] == target)
                {
            return true;
         }
        else if(arr[mid] < target)
                 {
            low = mid+1;
         } else {
```

```
            high = mid-1;
          }
        }
        return false;
    }

    public int smallestCommonElement(int[][] mat)
     {
        if(mat.length == 1) return mat[0][0];
        for(int a : mat[0]) {
            int count = 0;
            for(int i=1; i<mat.length; i++) {
                if(binarySearch(mat[i], 0, mat[i].length-1, a))
                    {
                    count++;
                } else {
                    break;
                }
            }
            if(count == mat.length-1) return a;
        }
        return -1;
    }
    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        System.out.println( "enter size of two dimensional matrix" );
       System.out.println( "enter row size of matrix " );
       int m=sc.nextInt();
       System.out.println( "enter column size of matrix " );
        int n=sc.nextInt();
        int[][] arr = new int[m][n];
System.out.println( "enter the elements " );

        for(int i=0;i<m;i++)
        for(int j=0;j<n;j++)
         arr[i][j] = sc.nextInt();
```

```
System.out.println( "smallest comman element :"+new SmallestCommonElement().small
estCommonElement(arr) );
    }
}
```

case=1
input=

```
enter size of two dimensional matrix
enter row size of matrix
4
enter column size of matrix
5
enter elements
1 2 3 4 5
2 4 5 8 10
3 5 7 9 11
1 3 5 7 9
```

output=

```
smallest comman element :5
```

case=2
input=

```
enter size of two dimensional matrix
enter row size of matrix
2
enter column size of matrix
2
enter elements for matrix
1 2
3 4
```

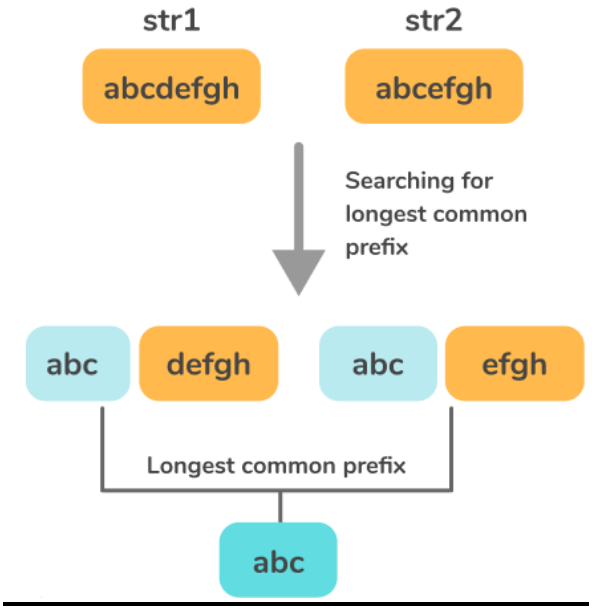
output=

```
smallest comman element :-1
```

4. Longest Common Prefix

Problem Statement:

- Given the array of strings S[], you need to find the longest string S which is the prefix of ALL the strings in the array.
- **Longest common prefix (LCP)** for a pair of strings S1 and S2 is the longest string S which is the prefix of both S1 and S2.
- For Example: longest common prefix of **"abcdefgh"** and **"abcefgh"** is **"ABC"**.



Examples:
Input: S[] = {"abcdefgh", "abcefgh"}
Output: "abc"
Explanation: Explained in the image description above

Input: S[] = {"abcdefgh", "aefghijk", "abcefgh"}
Output: "a"

Binary Search Approach

Algorithm:

- Consider the string with the smallest length. Let the length be **L**.
- Consider a variable **low = 0** and **high = L – 1**.
- Perform binary search:
  - Divide the string into two halves, i.e. **low – mid** and **mid + 1** to **high**.
  - Compare the substring upto the **mid** of this smallest string to every other character of the remaining strings at that index.
  - If the substring from **0** to **mid – 1** is common among all the substrings, update **low** with **mid + 1**, else update **high** with **mid – 1**
  - If **low == high**, terminate the algorithm and return the substring from **0** to **mid.**
  -

Java program for LongestCommonPrefix using Binary search approach

LongestCommonPrefix.java

```
import java.util.*;

class LongestCommonPrefix {
   public String longestCommonPrefix(String[] strs)
    {
      if (strs == null || strs.length == 0)
      return "";
         return longestCommonPrefix(strs, 0 , strs.length - 1);
   }

   private String longestCommonPrefix(String[] strs, int l, int r)
   {
      if (l == r)
      {
         return strs[l];
      }
      else
      {
         int mid = (l + r)/2;
         String lcpLeft =   longestCommonPrefix(strs, l , mid);
         String lcpRight =  longestCommonPrefix(strs, mid + 1,r);
         return commonPrefix(lcpLeft, lcpRight);
      }
   }
```

```
   String commonPrefix(String left,String right)
    {
       int min = Math.min(left.length(), right.length());
       for (int i = 0; i < min; i++)
        {
          if ( left.charAt(i) != right.charAt(i) )
              return left.substring(0, i);
       }
       return left.substring(0, min);
   }

   public static void main(String args[])
{
       Scanner sc= new Scanner(System.in);
       System.out.println("Enter Strings");
       String[] words = sc.nextLine().split(" ");

       System.out.println("Longest common Prefix is: "+new
LongestCommonPrefix().longestCommonPrefix(words));
   }
}
```

input:

Enter Strings

fly flower flow

output:

Longest common Prefix is: fl

input:

Enter Strings

f c i

output:

Longest common Prefix is:

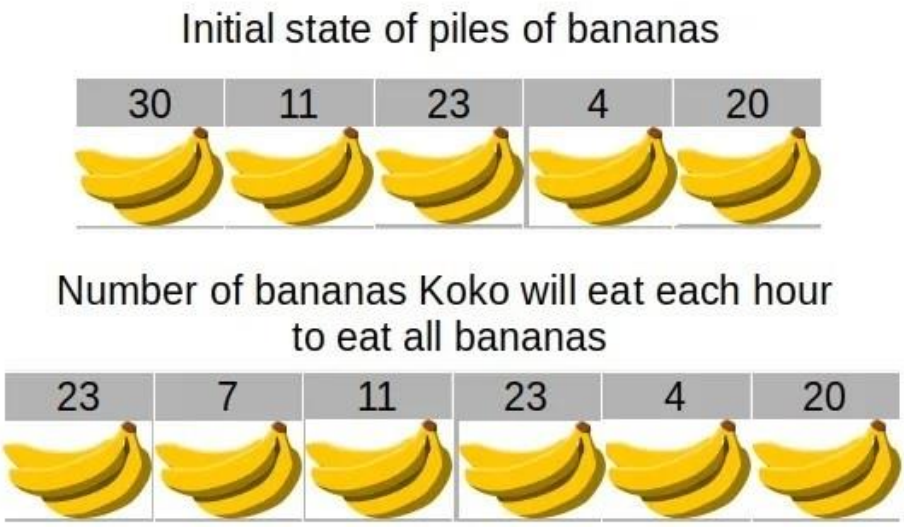
5. Koko Eating Bananas:

- Koko loves to eat bananas. There are n piles of bananas, the **ith** pile has **piles[i]** bananas. The guards have gone and will come back in h hours.
- Koko can decide her bananas-per-hour eating speed of **k**. Each hour, she chooses some pile of bananas and eats **k** bananas from that pile. If the pile has less than **k** bananas, she eats all of them instead and will not eat any more bananas during this hour.
- Koko likes to eat slowly but still wants to finish eating all the bananas before the guards return.
- Return *the minimum integer* k *such that she can eat all the bananas within* h *hours*.

Example 1:
Input: piles = [3,6,7,11], h = 8
Output: 4
Example 2:
Input: piles = [30,11,23,4,20], h = 5
Output: 30
Example 3:
Input: piles = [30,11,23,4,20], h = 6
Output: 23

Input:
piles = [**30**,**11**,**23**,**4**,**20**], H = **6**

output:
23



Koko will eat bananas in this way to eat all bananas in 6 hours:

First hour: 23
Second hour: 7
Third hour: 11
Fourth hour: 23

Approach for Koko Eating Bananas

The first and the most important thing to solve this problem is to bring out observations. Here are a few observations for our search interval:

1. Koko must eat at least one banana per hour. So this is the minimum value of K. let's name it as **Start**
2. We can limit the maximum number of bananas Koko can eat in one hour to the maximum number of bananas in a pile out of all the piles. So this is the maximum value of K. let's name it as **End.**

Now we have our search interval. Suppose the size of the interval is **Length** and the number of piles is **n.** The naive approach could be to check for each value in the interval. if for that value of K Koko can eat all bananas in H hour successfully then pick the minimum of them. The time complexity for the naive approach will be Length*n in worst case.

We can improve the time complexity by using Binary Search in place of Linear Search.
The time complexity using the Binary Search approach will be log(Length)*n.

Time complexity:
The time complexity of the above code is **O(n*log(W))** because we are performing a binary search between one and W this takes logW time and for each value, in the binary search, we are traversing the piles array. So the piles array is traversed logW times it makes the time complexity n*logW. Here n and W are the numbers of piles and the maximum number of bananas in a pile.
Space complexity:
The space complexity of the above code is **O(1)** because we are using only a variable to store the answer.

Java program for Kokoeatingbananas: Kokoeatingbanana.java

```
import java.util.*;
public class Kokoeatingbanana
        {
           public static int minEatingSpeed(int[] piles, int H)
           {
              int left = 1, right =Arrays.stream(piles).max().getAsInt();
              while (left < right)   {
                 int mid = (left + right) / 2, total = 0;
                 for (int p : piles)
                   total += (p + mid - 1) / mid;
                 if (total > H)
                 left = mid + 1;
                 else right = mid;
              }
              return left;
           }
          public static void main(String[] args) throws Exception
           {
                Scanner sc=new Scanner(System.in);
                 System.out.println("enter array size");
                 int n = sc.nextInt();
                 int arr[]=new int[n];
             System.out.println("enter the elements of array ");
                 for(int i=0;i<n;i++)    {
                 arr[i] =sc.nextInt();
                }
           System.out.println("enter hours");
            int H = sc.nextInt();
            int ans= minEatingSpeed(arr,H);
            System.out.println("minimum Eating bananas per hour :" +ans);
          }
        }
```

input=
enter array size
4
enter the elements of array
3 6 7 11
enter hours
8
output=
minimum Eating bananas per hour:4

II. Greedy Method:

- Among all the algorithmic approaches, the simplest and straightforward approach is the Greedy method.
- In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.
- Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit.
- This approach never reconsiders the choices taken previously
- This approach is mainly used to solve optimization problems.
- Greedy method is easy to implement and quite efficient in most of the cases.
- Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.
- In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.

Control abstraction (pseudo code) for Greedy Method

```
Algorithm GreedyMethod (a, n)
{
// a is an array of n inputs
        Solution: =Ø;
        for i: =0 to n do
        {
        s: = select (a);
        if (feasible (Solution, s)) then
        {
         Solution: = union (Solution, s);
         }
         else reject (); // if solution is not feasible reject it.
         }
         return solution;
 }
```

Three important activities:

1. A selection of solution from the given input domain is performed, i.e. s:= select(a).
2. The feasibility of the solution is performed, by using feasible '(solution, s)' and then all feasible solutions are obtained.
3. From the set of feasible solutions, the particular solution that minimizes or maximizes the given objection function is obtained. Such a solution is called optimal solution

Applications:

1. **Minimum product subset of an array**
2. **Best Time to Buy and Sell Stock**
3. **0/1 Knapsack Problem**
4. **Minimum cost spanning trees**
5. **Single source shortest path Problem**

1. Minimum product subset of an array

Given an array a, we have to find the minimum product possible with the subset of elements present in the array. The minimum product can be a single element also.

Examples:

Input : a[] = { -1, -1, -2, 4, 3 }
Output : -24
Explanation : Minimum product will be (-2 * -1 * -1 * 4 * 3) = -24

Input : a[] = { -1, 0 }
Output : -1
Explanation : -1(single element) is minimum product possible

Input : a[] = { 0, 0, 0 }
Output : 0

A simple solution is to generate all subsets, find the product of every subset and return the minimum product.

A better solution is to use the below facts.

->If there are even number of negative numbers and no zeros, the result is the product of all except the largest valued negative number.
->If there are an odd number of negative numbers and no zeros, the result is simply the product of all.
->If there are zeros and positive, no negative, the result is 0. The exceptional case is when there is no negative number and all other elements positive then our result should be the first minimum positive number.

Complexity Analysis:

Time Complexity: O(n)
Auxiliary Space: O(1)

Java program to find maximum product of a subset.

```
import java.util.*;

class MinProductSubset
{

   static int minProductSubset(int a[], int n)
   {
      if (n == 1)
         return a[0];

 /*Find count of negative numbers,count of zeros, maximum valued negative number,
minimum valued positive number and product of non-zero numbers */

        int negmax = Integer.MIN_VALUE;
        int posmin = Integer.MAX_VALUE;
       int count_neg = 0;
       int count_zero = 0;
        int product = 1;

        for (int i = 0; i < n; i++) {

           // if number is zero,count it  but dont multiply

  if (a[i] == 0) {
              count_zero++;
              continue;
           }

    // count the negative numbers  and find the max negative number
           if (a[i] < 0) {
              count_neg++;
              negmax = Math.max(negmax, a[i]);
           }

      // find the minimum positive number
           if (a[i] > 0 && a[i] < posmin)
              posmin = a[i];

           product *= a[i];
        }
```

```
        // if there are all zeroes or zero is present but no negative number is present

    if (count_zero == n  || (count_neg == 0 && count_zero > 0))
            return 0;

        // If there are all positive
        if (count_neg == 0)
            return posmin;

        // If there are even number except
        // zero of negative numbers
        if (count_neg % 2 == 0 && count_neg != 0) {

            // Otherwise result is product of
            // all non-zeros divided by maximum
            // valued negative.
            product = product / negmax;
        }

        return product;
      }

    public static void main(String[] args)
      {

        Scanner sc=new Scanner(System.in);
        System.out.println("enter size of the array");
        int n=sc.nextInt();
            System.out.println("enter elements");
            int a[]=new int[n];
        for(int i=0;i<n;i++)
        {
            a[i]=sc.nextInt();
        }
        System.out.println(minProductSubset(a, n));
    }
    }
```

case=1
input=
enter size of the array
5
enter elements
-1 -1 -2 4 3
output=
-24

case=2
input=
enter size of the array
2
enter elements
-1 0
output=
-1

case=3
input=
enter size of the array
3
enter elements
0 0 0
output=
0

2. Best Time to Buy and Sell Stock

Type I: At most one transaction is allowed

Given an array prices[] of length N, representing the prices of the stocks on different days, the task is to find the maximum profit possible for buying and selling the stocks on different days using transactions where at most one transaction is allowed.

Note: Stock must be bought before being sold.

Input: prices[] = {7, 1, 5, 3, 6, 4]
Output: 5
Explanation:
The lowest price of the stock is on the 2nd day, i.e. price = 1. Starting from the 2nd day, the highest price of the stock is witnessed on the 5th day, i.e. price = 6.
Therefore, maximum possible profit = 6 – 1 = 5.

Input: prices[] = {7, 6, 4, 3, 1}
Output: 0
Explanation: Since the array is in decreasing order, no possible way exists to solve the problem.

Time Complexity: O(N). Where N is the size of prices array.
Auxiliary Space: O(1). We do not use any extra space.

Java program for Best Time to Buy and Sell Stock with at most one transaction

BTBS_Atmostonetime.java

```
import java.util.*;

class BTBS_Atmostonetime
{
static int maxProfit(int prices[], int n)
{
 int minprice=Integer.MAX_VALUE;
 int max_profit = 0;

 for (int i = 0; i < n; i++)
 {
 if (prices[i]< minprice)
        minprice= prices[i];
```

```
 else if (prices[i] -minprice > max_profit)
  max_profit = prices[i] -minprice;
 }
 return max_profit;
}
public static void main(String args[])
{
       Scanner sc =new Scanner(System.in);
       System.out.println("Enter array size");

    int size=sc.nextInt();

        int prices[]=new int[size];

      System.out.println("Enter elements");

    for(int i=0;i<size;i++)
 {
     prices[i]=sc.nextInt();
     }
 int max_profit = maxProfit(prices,size);
 System.out.println("Maximum Profit is:" + max_profit);
}
}
```

Output:

case=1
Enter array size
6
Enter elements
7 1 5 3 6 4
Maximum Profit is:5

case=2
Enter array size
6
Enter elements
6 5 4 3 2 1
Maximum Profit is:0

Type II: Infinite transactions are allowed

Given an array price[] of length N, representing the prices of the stocks on different days, the task is to find the maximum profit possible for buying and selling the stocks on different days using transactions where any number of transactions are allowed.

input: prices[] = {7, 1, 5, 3, 6, 4}
Output: 7

Explanation:
Purchase on 2nd day. Price = 1.
Sell on 3rd day. Price = 5.
Therefore, profit = 5 – 1 = 4.
Purchase on 4th day. Price = 3.
Sell on 5th day. Price = 6.
Therefore, profit = 4 + (6 – 3) = 7.

Input: prices = {1, 2, 3, 4, 5}
Output: 4
Explanation:
Purchase on 1st day. Price = 1.
Sell on 5th day. Price = 5.
Therefore, profit = 5 – 1 = 4.

Approach: The idea is to maintain a boolean value that denotes if there is any current purchase ongoing or not. If yes, then at the current state, the stock can be sold to maximize profit or move to the next price without selling the stock. Otherwise, if no transaction is happening, the current stock can be bought or move to the next price without buying.

Java program for Best Time to Buy and Sell Stock with Infinite transactions :

BTBS_InfiniteTransactions.java

```
 import java.util.*;
   class BTBS_InfiniteTransactions
{
      public static void main(String [] args)
  {
         Scanner sc = new Scanner(System.in);
         int n=sc.nextInt();
         int arr[] = new int[n];
         for(int i=0;i<n;i++){
            arr[i]=sc.nextInt();
         }
        System.out.println(prof(arr,n));
      }
     public static int prof(int arr[] , int n){
        int profit=0;
         for(int i=0;i<n-1;i++)
        {
            if(arr[i]<arr[i+1])
           {
               profit+=(arr[i+1]-arr[i]);
            }
         }
        return profit;
      }
   }
```

input=

7

7 1 5 3 6 4

Output=

7

3. 0/1 Knapsack Problem:

- In this problem we have a Knapsack that has a weight limit M.
- There are items i1, i2, ..., in each having weight w1, w2, … wn and some benefit (value or profit) associated with it p1, p2, ..., pn
- Our objective is to maximize the benefit such that the total weight inside the knapsack is at most M, and we are also allowed to take an item in fractional part.
- There are **n items in the store**
  -weight of **ith item $wi > 0$**
  - Profit for **ith item $pi > 0$ and**
  -Capacity of the Knapsack is **M.**
- In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction ***xi of ith item.*** $0 \leq xi \leq 1$
- The ith item contributes the weight $xi.\ wi$ to the total weight in the knapsack and profit $xi*pi$ to the total profit.
- Hence, the objective of this algorithm is to $maximize\ \Sigma_{n=1\ to\ n}\ (xi.\ pi)$
- subject to constraint, $\Sigma_{n=1\ to\ n}(xi.\ wi) \leq M$
- It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.
- Thus, an optimal solution can be obtained by $\Sigma_{n=1\ to\ n}(xi.\ wi) = M$
- In this context, first we need to sort those items according to the value of pi/wi, **so that** $(pi+1)/(wi+1) \leq pi/wi$.
- **Here,** ***x is an array to store the fraction of items.***

Algorithm:

```
void GreedyKnapsack(float m, int n)
// p[1:n] and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
// p[i]/w[i] >= p[i+1]/w[i+1]. m is the knapsack
// size and x[1:n] is the solution vector.
{
   for (int i=1; i<=n; i++) x[i] = 0.0; // Initialize x.
   float U = m;
   for (i=1; i<=n; i++) {
     if (w[i] > U) break;
     x[i] = 1.0;
     U -= w[i];
   }
   if (i <= n) x[i] = U/w[i];
}
```

Analysis:

If the provided items are already sorted into a decreasing order of $\boldsymbol{pi/wi}$, then the while loop takes a time in ($\boldsymbol{n}$); Therefore, the total time including the sort is in $\boldsymbol{O(n\ log\ n)}$.

Example 1:

- Let us consider that the capacity of the knapsack $\boldsymbol{W = 60}$ and the list of provided item are shown in below

Profits=(p1,p2,p3,p4)=(280,100,120,120)

Weigts=(w1,w2,w3,w4)=(40,10,20,24)

- As the provided items are not sorted based on **pi / wi. After sorting, the items are as** shown in the following table.

ITEM	B	A	C	D
PROFIT	100	280	120	120
WEIGHT	10	40	20	24
RATIO (Pi/Wi)	10	7	6	5

- After sorting all the items according to $\boldsymbol{pi/wi}$. **First all of *B is chosen as weight of B is less*** than the capacity of the knapsack.
- Next, item ***A is chosen, as the available capacity of the*** knapsack is greater than the weight of ***A.***
- ***Now, C is chosen as the next item. However,***the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C.***

  Hence, fraction of ***C (i.e. (60 − 50)/20) is chosen.***
- Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.
- Feasible solution is **(x1,x2,x3,x4)=(1,1,0.5,0)**
- The total weight of the selected items is $\boldsymbol{10 + 40 + 20 * (10/20) = 60}$
- And the total profit is $\boldsymbol{100 + 280 + 120 * (10/20) = 380 + 60 = 440}$
- This is the optimal solution. We cannot gain more profit selecting any different combination of items.

Java Program for 0/1 knapsack : Greedy_Knapsack.java

```
import java.util.*;

class Greedy_Knapsack
{
   public static void main(String args[])
   {
      int m,i,j,k[],temp1;
      float max,temp;
      float  w[],p[],r[],x[],n,w1=0,sum=0;
      Scanner sc=new Scanner(System.in);
      System.out.println("enter kanpsack size ");
      n=sc.nextFloat();
      System.out.println("enter the no.of Items ");
      m=sc.nextInt();
      w=new float[m];
      p=new float[m];
      r=new float[m];
      x=new float[m];
      k=new int[m];
      System.out.println("enter the weigths of Items ");
      for(i=0;i<m;i++)
      {
         k[i]=i+1;
         w[i]=sc.nextInt();
      }
      System.out.println("enter the profits of Items");
      for(i=0;i<m;i++)
      {
         p[i]=sc.nextInt();
      }
      for(i=0;i<m;i++)
      {
         r[i]=p[i]/w[i];
      }
      for(i=0;i<m-1;i++)
      {
         for(j=0;j<m-i-1;j++)
         {
            if(r[j+1]>r[j])
            {
               temp=r[j];
```

```
                r[j]=r[j+1];
                r[j+1]=temp;

                temp=p[j];
                p[j]=p[j+1];
                p[j+1]=temp;

                temp=w[j];
                w[j]=w[j+1];
                w[j+1]=temp;

                temp1=k[j];
                k[j]=k[j+1];
                k[j+1]=temp1;
            }
        }
    }
    for(i=0;i<m;i++)
    {
        if(n>w[i]) {
            n=n-w[i];
            x[i]=1;
        }
        else if(n==0){
            x[i]=0;
        }
        else
        {
            x[i]=n/w[i];
            n=0;
        }
    }
    System.out.println("feasible solution is ");
    System.out.println("Itemno"+"\t"+"Weights"+"\t"+"Profits"+"\t"+"pi/wi Ratio"+"\t"
+"Selected");
    for(i=0;i<m;i++)
    {
        System.out.print(k[i]+"\t"+w[i]+"\t"+p[i]+"\t"+r[i]+"\t"+"\t"+ x[i]+"\n");
    }
    for(i=0;i<m;i++)
    {
        sum=sum+(x[i]*p[i]);
    }
```

```
        System.out.println("Optimal Solution: Maximum Profit is ");
        System.out.println(sum);
        /*for(i=0;i<m;i++)
        {
           w1=w1+w[i]*x[i];//for checing whether total equal to actual weight of the bag
        }*/
        //System.out.println(w1);
    }
}
```

Output:

```
enter kanpsack size
60
enter the no.of Items
4
enter the weigths of Items
40 10 20 24
enter the profits of Items
280 100 120 120
feasible solution is
Itemno    Weights Profits   pi/wi Ratio     Selected
  2          10.0    100.0      10.0          1.0
  1          40.0    280.0       7.0          1.0
  3          20.0    120.0       6.0          0.5
  4          24.0    120.0       5.0          0.0
Optimal Solution: Maximum Profit is
440.0
```

Disjoint Sets:

- A disjoint-set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.
- A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

  Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

  Union: Join two subsets into a single subset.

  In this post, we will discuss the application of Disjoint Set Data Structure.
  The application is to check whether a given graph contains a cycle or not.

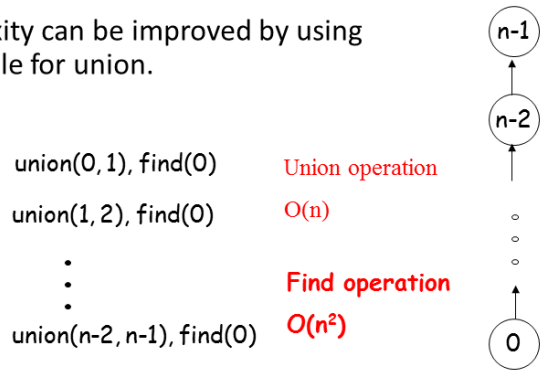
- Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an algorithm to detect cycle.

- This is another method based on Union-Find. This method assumes that the graph doesn't contain any self-loops.
- Two sets A and B are said to be **disjoint** if there are no common elements
  i.e., $A \cap B = \varnothing$.
- Example:
  - S_1={1,7,8,9}, S_2={2,5,10}, and S_3={3,4,6} are three disjoint sets.

- We identify a set by choosing a **representative element** of the set. It doesn't matter which element we choose, but once chosen, it can't be changed.
- **Disjoint set operations:**
  - **FIND-SET(x)**: Returns the representative of the set
    - containing x.
  - **UNION(i,j):** Combines the two sets i and j into one new
    - set. A new representative is selected.
    - (Here i and j are the representatives of the sets)

Analysis Union-Find Operations

- For a set of *n* elements each in a set of its own, then the result of the union function is a degenerate tree.
- The time complexity of the following union-find operation is ***O(n²).***
- The complexity can be improved by using weighting rule for union.



Weighted Rule for Union(i,j):

- If the number of nodes in the tree with root *i* is less than the number in the tree with root *j*, then make *j* the parent of *i* ; otherwise make *i* the parent of *j*.

Pseudo code

```
Algorithm WeightedUnion(i, j)
// Union sets with roots i and j, i ≠ j, using the
// weighting rule. p[i] = -count[i] and p[j] = -count[j].
{
    temp := p[i] + p[j];
    if (p[i] > p[j]) then
    { // i has fewer nodes.
        p[i] := j; p[j] := temp;
    }
    else
    { // j has fewer or equal nodes.
        p[j] := i; p[i] := temp;
    }
}
```

Array-Representation:

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent				-1	-1	-1	-1	-1	-1	-1

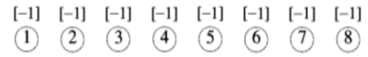
```
void union2 (int i, int j)
{
  int temp = parent[i] + parent[j];
  if ( parent[i]>parent[j])
{   // i has fewer nodes
     parent[i]=j;
     parent[j]=temp;
  }
  else
{   // j has fewer nodes or equal nodes
     parent[j]=i;
     parent[i]=temp;
  }
}
```

unoin2 (0 , 1)
unoin2 (0 , 2)
unoin2 (0 , 3)

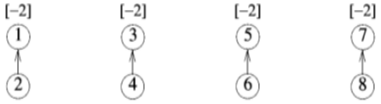
temp = -3

0 ◦ ◦ ◦ n-1

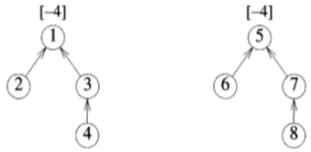
EX: unoin2 (0 , 1) , unoin2 (0 , 2) , unoin2 (0 , 2)



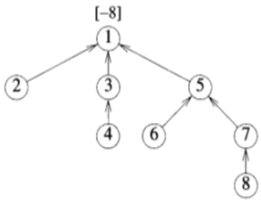
(a) Initial height-1 trees



(b) Height-2 trees following *Union*(1,2), (3,4), (5,6), and (7,8)



(c) Height-3 trees following *Union*(1,3) and (5,7)



(d) Height-4 tree following *Union*(1,5)

Collapsing Rule(finding an element):

- If j is a node on the path from i to its root and parent[i]≠ root(i), then set parent[j] to root(i).
-

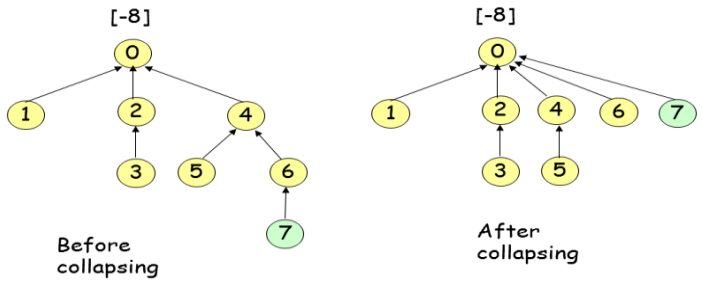
The first run of find operation will collapse the tree. Therefore, all following find operation of the same element only goes up one link to find the root.

Pseudo code :

```
1   Algorithm CollapsingFind(i)
2   // Find the root of the tree containing element i. Use the
3   // collapsing rule to collapse all nodes from i to the root.
4   {
5       r := i;
6       while (p[r] > 0) do r := p[r]; // Find the root.
7       while (i ≠ r) do  // Collapse nodes from i to root r.
8       {
9           s := p[i]; p[i] := r; i := s;
10      }
11      return r;
12  }
```

Algorithm 2.15 Find algorithm with collapsing rule

Set Find with Collapsing Rule



4. Minimum cost spanning trees:

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.
- Hence, a spanning tree does not have cycles and it cannot be disconnected.

  Note 1: Every connected and undirected Graph G has at least one spanning tree.

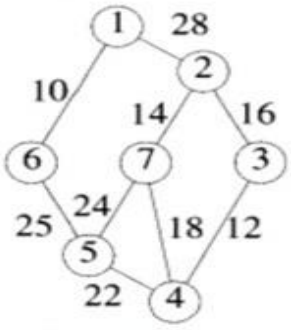
  Note 2: A disconnected graph does not have any spanning tree.
- A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes.

4.1. Kruskal's Algorithm:

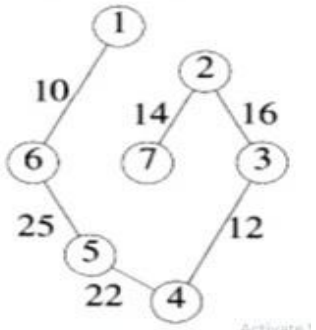
Step 1 - Remove all loops and Parallel Edges.

Step 2 - Arrange all edges in their increasing order of weight.

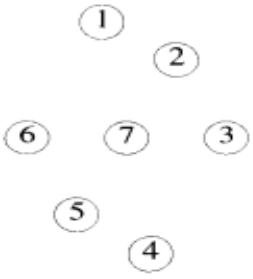
Step 3 - Add the edge which has the least weightage iff it does not form cycle.



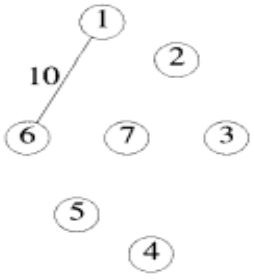
Graph



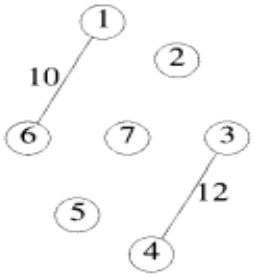
Resultant Graph



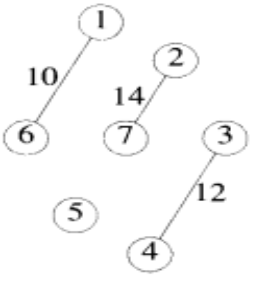
(a)



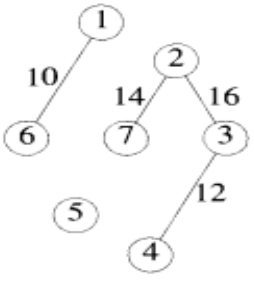
(b)



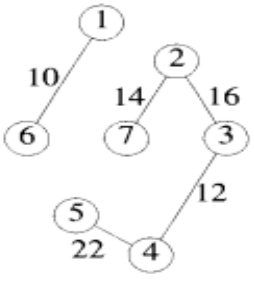
(c)



(d)



(e)



(f)

Pseudo Code:

```
1   Algorithm Kruskal(E, cost, n, t)
2   // E is the set of edges in G. G has n vertices. cost[u, v] is the
3   // cost of edge (u, v). t is the set of edges in the minimum-cost
4   // spanning tree. The final cost is returned.
5   {
6       Construct a heap out of the edge costs using Heapify;
7       for i := 1 to n do parent[i] := -1;
8       // Each vertex is in a different set.
9       i := 0; mincost := 0.0;
10      while ((i < n - 1)  and (heap not empty)) do
11      {
12          Delete a minimum cost edge (u, v) from the heap
13          and reheapify using Adjust;
14          j := Find(u); k := Find(v);
15          if (j ≠ k) then
16          {
17              i := i + 1;
18              t[i, 1] := u; t[i, 2] := v;
19              mincost := mincost + cost[u, v];
20              Union(j, k);
21          }
22      }
23      if (i ≠ n - 1) then write ("No spanning tree");
24      else return mincost;
25  }
```

Time Complexity: O(ElogE)+O(E*4*alpha), ElogE for sorting and E*4*alpha for findParent operation ‘E’ times

Space Complexity: O(N). Parent array+Rank Array

Java Program for Kruskal's algorithm: Kruskals_Algo.java

```
import java.util.*;

class Node
{
   private int u;
   private int v;
   private int weight;

   Node(int _u, int _v, int _w)
   {
      u = _u; v = _v; weight = _w;
   }
   int getV()
   {
      return v;
   }
   int getU()
   {
      return u;
   }
   int getWeight()
   {
      return weight;
      }
}

class SortComparator implements Comparator<Node>
{
   @Override
   public int compare(Node node1, Node node2)
   {
      if (node1.getWeight() < node2.getWeight())
         return -1;
      if (node1.getWeight() > node2.getWeight())
         return 1;
      return 0;


   }
}
```

```
class Kruskals_Algo
{
   private int findPar(int u, int parent[])
   {
      if(u==parent[u]) return u;
      return parent[u] = findPar(parent[u], parent);
   }
   private void union(int u, int v, int parent[], int rank[])
   {
      u = findPar(u, parent);
      v = findPar(v, parent);
      if(rank[u] < rank[v])
      {
         parent[u] = v;
      }
      else if(rank[v] < rank[u])
      {
         parent[v] = u;
      }
      else
      {
         parent[v] = u;
         rank[u]++;
      }
   }
   void KruskalAlgo(ArrayList<Node> adj, int N)
   {
      Collections.sort(adj, new SortComparator());
      int parent[] = new int[N];
      int rank[] = new int[N];

      for(int i = 0;i<N;i++)
      {
         parent[i] = i;
         rank[i] = 0;
      }

      int costMst = 0;
      ArrayList<Node> mst = new ArrayList<Node>();
      for(Node it: adj)
      {
         if(findPar(it.getU(), parent) != findPar(it.getV(), parent))
         {
```

```
            costMst += it.getWeight();
            mst.add(it);
            union(it.getU(), it.getV(), parent, rank);
          }
        }
        System.out.println("minimum cost is:"+costMst);
        System.out.println("Spanning Tree is");
        for(Node it: mst)
        {
          System.out.println(it.getU() + " - " +it.getV());
        }
    }
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);

        //show custom message
        System.out.println("Enter number of vertices: ");

        //store user entered value into variable v
       int  n = sc.nextInt();
        ArrayList<Node> adj = new ArrayList<Node>();

         for(int i = 0; i < n; i++)
         {
            System.out.println("Enter weight for edge");
            int x = sc.nextInt();

            System.out.println("Enter source value for edge");
            int y = sc.nextInt();

            System.out.println("Enter destination value for edge");
            int z= sc.nextInt();

              adj.add(new Node(x, y,z ));

        }

        Test obj = new Test();
        obj.KruskalAlgo(adj, n);

    }
}
```

input=

```
Enter number of vertices:
5
Enter weight for edge
0
Enter source value for edge
1
Enter destination value for edge
2
Enter weight for edge
0
Enter source value for edge
3
Enter destination value for edge
6
Enter weight for edge
1
Enter source value for edge
3
Enter destination value for edge
8
Enter weight for edge
1
Enter source value for edge
2
Enter destination value for edge
3
Enter weight for edge
1
Enter source value for edge
4
Enter destination value for edge
5
```

output=

```
minimum cost is:16
Spanning Tree is
0 - 1
1 - 2
1 - 4
0 - 3
```

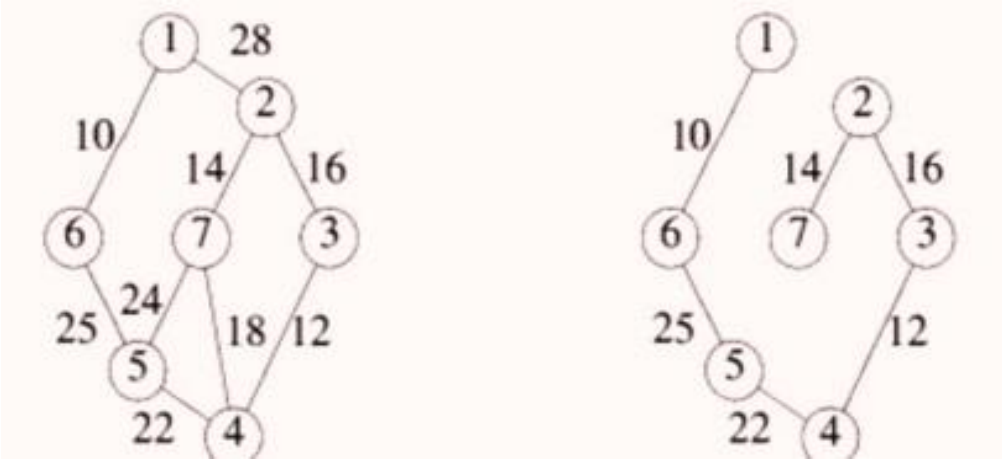
4.2.Prim's Algorithm:

- Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

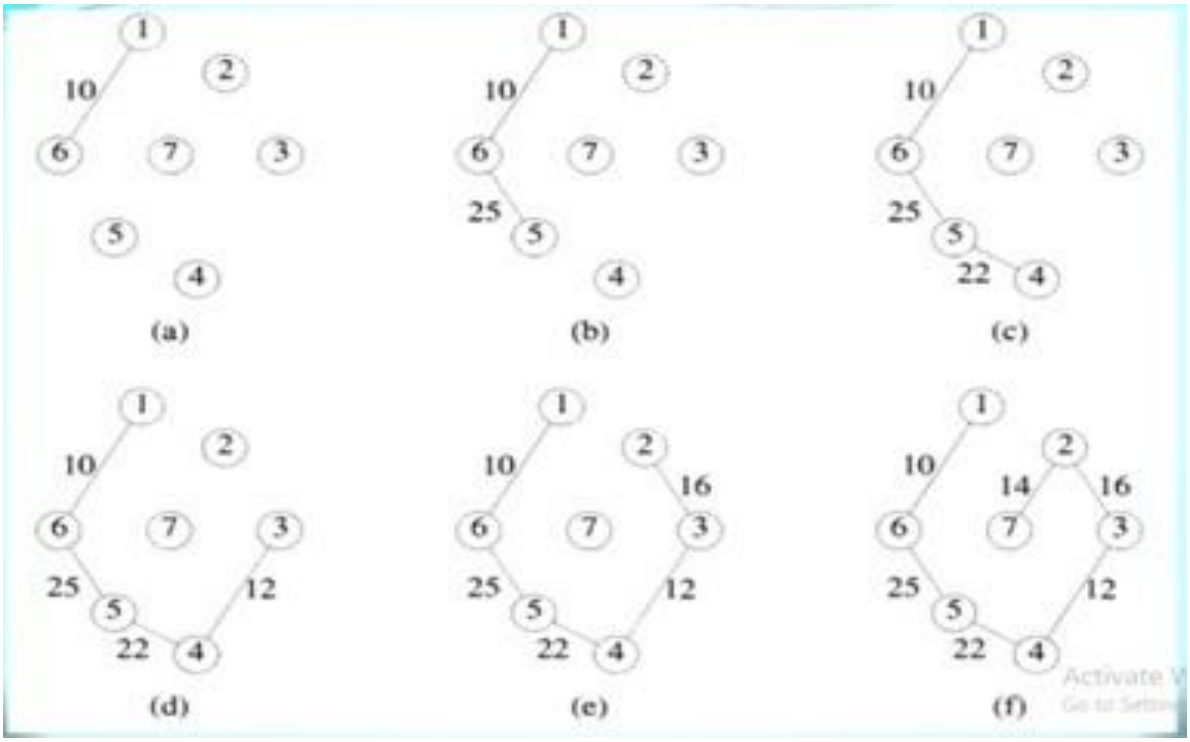
  Step 1 - Remove all loops and parallel edges.

  Step 2 - Choose any arbitrary node as root node.

  Step 3 - Check outgoing edges and select the one with less cost.



Graph **Resultant Graph**



Pseudo Code:

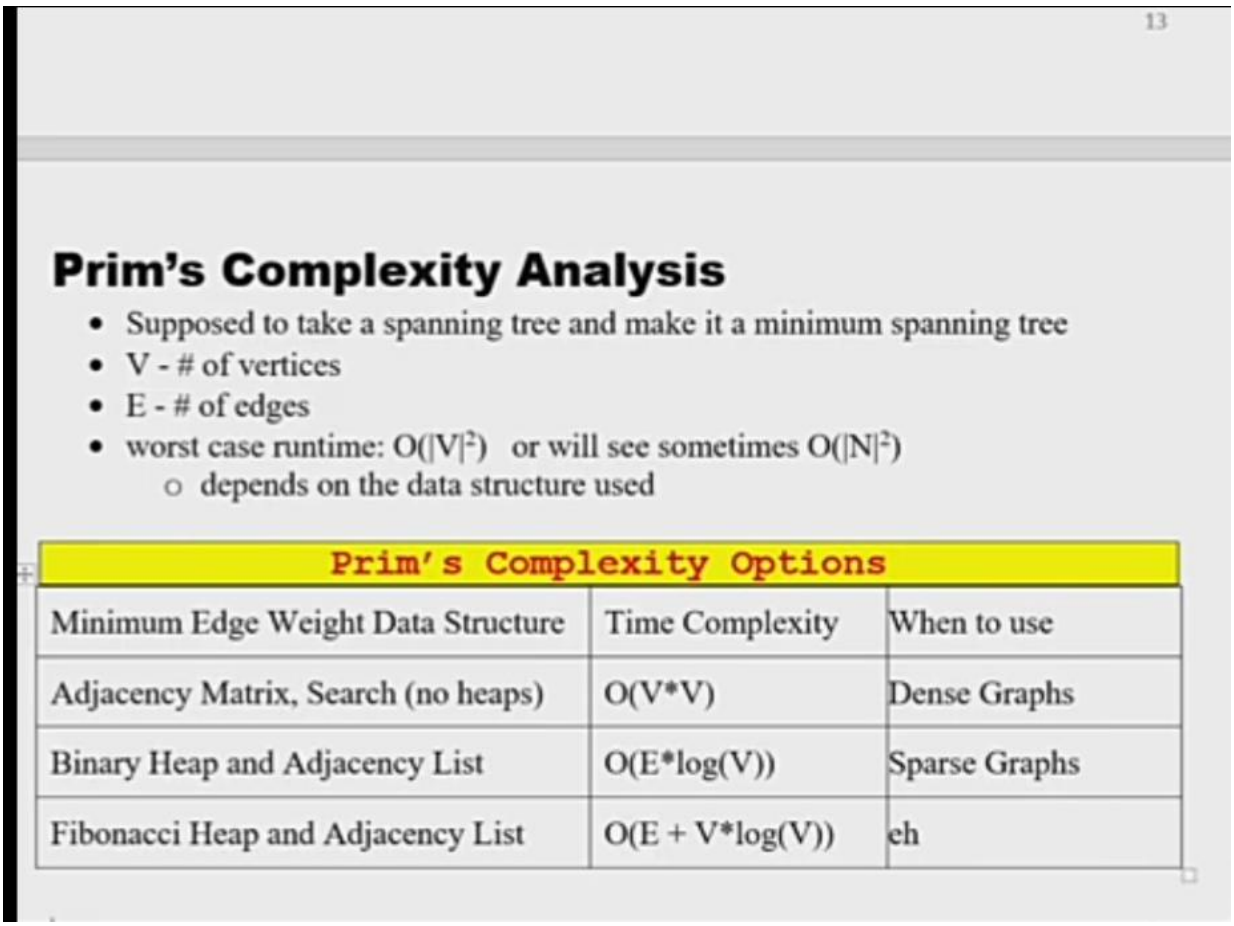
```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9       Let (k, l) be an edge of minimum cost in E;
10      mincost := cost[k, l];
11      t[1, 1] := k; t[1, 2] := l;
12      for i := 1 to n do  // Initialize near.
13          if (cost[i, l] < cost[i, k]) then near[i] := l;
14          else near[i] := k;
15      near[k] := near[l] := 0;
16      for i := 2 to n − 1 do
17      { // Find n − 2 additional edges for t.
18          Let j be an index such that near[j] ≠ 0 and
19          cost[j, near[j]] is minimum;
20          t[i, 1] := j; t[i, 2] := near[j];
21          mincost := mincost + cost[j, near[j]];
22          near[j] := 0;
23          for k := 1 to n do // Update near[ ].
24              if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                  then near[k] := j;
26      }
27      return mincost;
28  }
```

Java Program for Prims algorithm using Adjacency List: Prims_Algo.java

```
import java.util.*;

class Prims_Algo
{
  public void Prim(int G[][], int V)
  {
    int INF = 9999999;

    int no_edge; // number of edge

    // create a array to track selected vertex    // selected will become true otherwise false

    boolean[] selected = new boolean[V];

    // set selected false initially
    Arrays.fill(selected, false);

    // set number of edge to 0
    no_edge = 0;

    // the number of egde in minimum spanning tree will be
    // always less than (V -1), where V is number of vertices in
    // graph

    // choose 0th vertex and make it true
    selected[0] = true;

    // print for edge and weight
    System.out.println("Edge : Weight");

    while (no_edge < V - 1) {
      // For every vertex in the set S, find the all adjacent vertices
      // , calculate the distance from the vertex selected at step 1.
      // if the vertex is already in the set S, discard it otherwise
      // choose another vertex nearest to selected vertex at step 1.

      int min = INF;
      int x = 0; // row number
      int y = 0; // col number

      for (int i = 0; i < V; i++) {
```

```
      if (selected[i] == true) {
        for (int j = 0; j < V; j++) {
          // not in selected and there is an edge
          if (!selected[j] && G[i][j] != 0) {
            if (min > G[i][j]) {
              min = G[i][j];
              x = i;
              y = j;
            }
          }
        }
      }
    }
    System.out.println(x + " - " + y + " :  " + G[x][y]);
    selected[y] = true;
    no_edge++;
  }
 }
 public static void main(String[] args)
 {
      Test g = new Test();
     Scanner sc=new Scanner(System.in);
    System.out.println("enter number vertices");

    int  V=sc.nextInt();
    System.out.println("enter row  size of the matrix");
    int x=sc.nextInt();
    System.out.println("enter column  size of the matrix");
    int y=sc.nextInt();

    int G[][]= new int[x][y];
     System.out.println("adjacency matrix is");
  for (int i=0;i<x;i++)
  {
      for(int j=0;j<y;j++)
      {
         G[i][j]=sc.nextInt();
      }
  }

  g.Prim(G, V);
 }
}
```

input=
enter number vertices
5
enter row size of the matrix
5
enter column size of the matrix
5
adjacency matrix is
0 9 75 0 0
9 0 95 19 42
75 95 0 51 66
0 19 51 0 31
0 42 66 31 0
output=
Edge : Weight
0 - 1 : 9
1 - 3 : 19
3 - 4 : 31
3 - 2 : 51



5. Single source shortest path Problem:

- For a given source node in the graph, the algorithm finds the shortest path between that node and every other.
- It also used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

Dijkstra's Algorithm requires a graph and source vertex to work.The algorithm is purely based on greedy approach and thus finds the locally optimal choice(local minima in this case) at each step of the algorithm.

In this algorithm each vertex will have two properties defined for it-

- **Visited property**:-
  - This property represents whether the vertex has been visited or not.
  - We are using this property so that we don't revisit a vertex.
  - A vertex is marked visited only after the shortest path to it has been found.
- **Path property**:-
  - This property stores the value of the current minimum path to the vertex. Current minimum path means the shortest way in which we have reached this vertex till now.
  - This property is updated whenever any neighbour of the vertex is visited.
  - The path property is important as it will store the final answer for each vertex.

Initially all the vertices are marked unvisited as we have not visited any of them. Path to all the vertices is set to infinity excluding the source vertex. Path to the source vertex is set to zero(0).

Then we pick the source vertex and mark it visited.After that all the neighbours of the source vertex are accessed and **relaxation** is performed on each vertex. Relaxation is the process of trying to lower the cost of reaching a vertex using another vertex.

In relaxation, the path of each vertex is updated to the minimum value amongst the current path of the node and the sum of the path to the previous node and the path from the previous node to this node.

Assume that p[v] is the current path value for node v, p[n] is the path value upto the previously visited node n, and w is the weight of the edge between the curent node and previously visited node(edge weight between v and n)

Mathematically, relaxation can be represented as: **p[v]=minimum(p[v] , p[n] + w)**

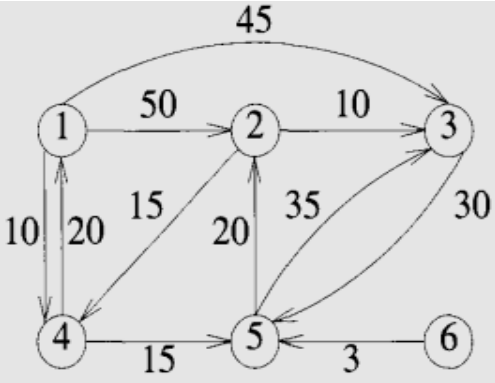
Then in every subsequent step, an unvisited vertex with the least path value is marked visited and its neighbour's paths updated.

The above process is repeated till all the vertices in the graph are marked visited.

Whenever a vertex is added to the visited set, the path to all of its neighbouring vertices is changed according to it.

If any of the vertex is not reachable(disconnected component), its path remains infinity. If the source itself is a disconected component, then the path to all other vertices remains infinity.

➢



(a) Graph

	Path	Length
1)	1, 4	10
2)	1, 4, 5	25
3)	1, 4, 5, 2	45
4)	1, 3	45

(b) Shortest paths from 1

Pseudo Code:

```
1   Algorithm ShortestPaths(v, cost, dist, n)
2   // dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
3   // path from vertex v to vertex j in a digraph G with n
4   // vertices. dist[v] is set to zero. G is represented by its
5   // cost adjacency matrix cost[1 : n, 1 : n].
6   {
7       for i := 1 to n do
8       { // Initialize S.
9           S[i] := false; dist[i] := cost[v, i];
10      }
11      S[v] := true; dist[v] := 0.0; // Put v in S.
12      for num := 2 to n − 1 do
13      {
14          // Determine n − 1 paths from v.
15          Choose u from among those vertices not
16          in S such that dist[u] is minimum;
17          S[u] := true; // Put u in S.
18          for (each w adjacent to u with S[w] = false) do
19              // Update distances.
20              if (dist[w] > dist[u] + cost[u, w])) then
21                  dist[w] := dist[u] + cost[u, w];
22      }
23  }
```

Java Program for Single source shortest path (Dijkstra's Algorithm):

Dijkstra's_Shortestpath_algo.java

```
import java.util.*;
import java.io.*;
import java.lang.*;
import java.util.*;

class Dijkstra's_Shortestpath_algo
{
   // A utility function to find the vertex with minimum distance value, from the set of
  // vertices not yet  included in shortest path tree

 static final int V = 9;
   int minDistance(int dist[], Boolean sptSet[])
   {
      // Initialize min value
      int min = Integer.MAX_VALUE, min_index = -1;

      for (int v = 0; v < V; v++)
         if (sptSet[v] == false && dist[v] <= min) {
            min = dist[v];
            min_index = v;
         }

      return min_index;
   }

   // A utility function to print the constructed distance
   // array
   void printSolution(int dist[])
   {
      System.out.println(
         "Vertex \t\t Distance from Source");
      for (int i = 0; i < V; i++)
         System.out.println(i + " \t\t " + dist[i]);
   }

  // Function that implements Dijkstra's single source  shortest path algorithm for a graph
represented using  adjacency matrix representation
```

```
void dijkstra(int graph[][], int src)
   {
      int dist[] = new int[V]; // The output array.
     //dist[i] will hold the shortest distance from src to i

      // sptSet[i] will true if vertex i is included in
 // shortest path tree or shortest distance from src     To i is finalized

      Boolean sptSet[] = new Boolean[V];

      // Initialize all distances as INFINITE and stpSet[]  as false
      for (int i = 0; i < V; i++) {
         dist[i] = Integer.MAX_VALUE;
         sptSet[i] = false;
      }

      // Distance of source vertex from itself is always 0
      dist[src] = 0;

      // Find shortest path for all vertices
      for (int count = 0; count < V - 1; count++)
       {
         // Pick the minimum distance vertex from the set of vertices not yet processed.
        //u is always equal to src in first iteration.
         int u = minDistance(dist, sptSet);

         // Mark the picked vertex as processed
         sptSet[u] = true;

         // Update dist value of the adjacent vertices of  the picked vertex.

    for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is an edge from u to v, and total
            // weight of path from src to v through u is  smaller than current value of dist[v]

 if (!sptSet[v] && graph[u][v] != 0  && dist[u] != Integer.MAX_VALUE && dist[u] +
graph[u][v] < dist[v])
                {
                        dist[v] = dist[u] + graph[u][v];
                }
      }
```

```
        // print the constructed distance array
        printSolution(dist);
    }

    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
            System.out.println("enter row  size of the matrix");
            int x=sc.nextInt();
            System.out.println("enter column  size of the matrix");
            int y=sc.nextInt();

            int graph[][]= new int[x][y];

        System.out.println("adjacency matrix is");
        for (int i=0;i<x;i++)
        {
            for(int j=0;j<y;j++)
            {
                graph[i][j]=sc.nextInt();
            }
        }

        Test t = new Test();

        // Function call
        t.dijkstra(graph, 0);
    }
}
```

Input:

```
0 4 0 0 0 0 0 8 0
4 0 8 0 0 0 0 11 0
0 8 0 7 0 4 0 0 2
0 0 7 0 9 14 0 0 0
0 0 0 9 0 10 0 0 0
0 0 4 14 10 0 2 0 0
0 0 0 0 0 2 0 1 6
8 11 0 0 0 0 1 0 7
0 0 2 0 0 0 6 7 0
```

Output:

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

Time Complexity: O(V2)
Auxiliary Space: O(V)

III. Backtracking: General method, Applications: N Queens Problem, Hamiltonian Cycle, Brace Expansion, Gray Code, Path with Maximum Gold, Generalized Abbreviation, Campus Bikes II.

General method:

Backtracking is a type of technique that is based on a particular algorithm to solve a basic problem. It basically uses the recursive call function to get a particular solution by creating or building a solution stepwise with increasing values and time.
This algorithm is applied to the particular specific types of problems:
1. Problems that require decision-making and are used to find a good solution for the problem.

2. The problems that can be optimized and are used to find the better solution that can be applied.

3. In the case of enumeration kind of problems, the algorithm is used to find the set of all easy and workable solutions for the given problem.

For Backtracking problems, the algorithm finds a path sequence for the solution of the problem that keeps some checkpoints,i.e, a point from where the given problem can take a backtrack if no workable solution is found out for the problem.
Backtracking – a schematic process of trying out different types of sequences of the various decision until it reaches the conclusion.
Some terms related to backtracking are :

- **Live Node:** Nodes that can further generate are known live nodes.
- **E Node:** the nodes whose children are generated and become a successor node.
- **Success Node:** if the node provides a solution that is feasible.
- **Dead Node:** A node that cannot be further generated and does not provide a particular solution.

There are many problems that can be solved by the use of a backtracking algorithm, and it can be used over a complex set of variables or constraints, they are basically of two types :
1. **Implicit constraint:** a particular rule used to check how many each element is in a proper sequence is related to each other.
2. **Explicit constraint:** the rule that restricts every element to get chosen from the particular set.

Recursive Backtracking Algorithm:

```
1   Algorithm Backtrack(k)
2   // This schema describes the backtracking process using
3   // recursion. On entering, the first k − 1 values
4   // x[1], x[2], ..., x[k − 1] of the solution vector
5   // x[1 : n] have been assigned. x[ ] and n are global.
6   {
7       for (each x[k] ∈ T(x[1], ..., x[k − 1])) do
8       {
9           if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
10          {
11              if (x[1], x[2], ..., x[k] is a path to an answer node
12                  then write (x[1 : k]);
13              if (k < n) then Backtrack(k + 1);
14          }
15      }
16  }
```

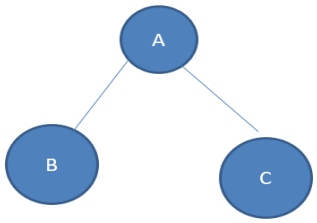
Algorithm 7.1 Recursive backtracking algorithm

Iterative Backtracking Algorithm:

```
1   Algorithm IBacktrack(n)
2   // This schema describes the backtracking process.
3   // All solutions are generated in x[1 : n] and printed
4   // as soon as they are determined.
5   {
6       k := 1;
7       while (k ≠ 0) do
8       {
9           if (there remains an untried x[k] ∈ T(x[1], x[2], ...,
10              x[k − 1]) and B_k(x[1], ..., x[k]) is true) then
11          {
12                  if (x[1], ..., x[k] is a path to an answer node)
13                      then write (x[1 : k]);
14                  k := k + 1; // Consider the next set.
15          }
16          else k := k − 1; // Backtrack to the previous set.
17      }
18  }
```

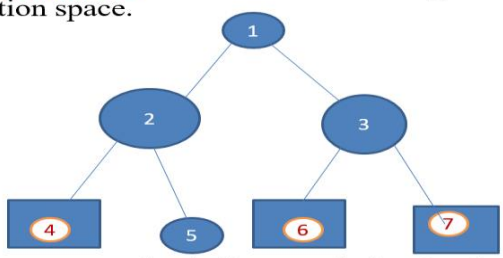
BACKTRACKING: Solution Space

- Tuples that satisfy the explicit constraints define a solution space.
- The solution space can be organized into a tree.
- Each node in the tree defines a problem state.



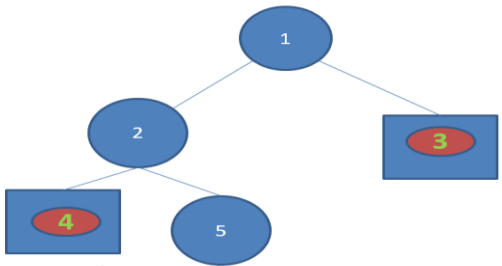
Here A,B, and C are problem States.

- All paths from the root to other nodes define the state-space of the problem.
- **Solution states** are those states leading to a tuple in the solution space.



- Here square nodes indicates solution for the solution space, there exits 3 solution states. These are represented inform of tuples. (1,2,4) ,(1,3,6) and (1,3,7).

- **Answer nodes** are those solution states leading to an answer-tuple(i.e. tuples which satisfy implicit constraints).

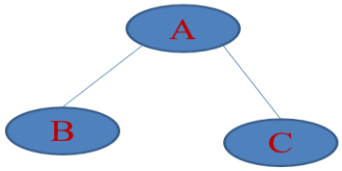


- Here node 3 and 4 are answer states (1,3) and (1,2,4) are solution states.

BACKTRACKING -Terminology

- **LIVE NODE** A node which has been generated and all of whose children are not yet been generated .

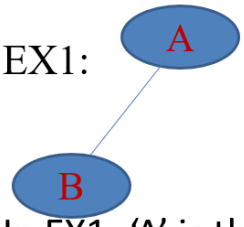
– Here **'A'** is the **Live node.**



– Here **'A'** is the **AliveNode** and **B&C** are **Live nodes**

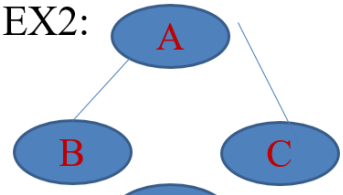
- **E-NODE** (Node being expanded) - The live node whose children are currently being generated .

- EX1:

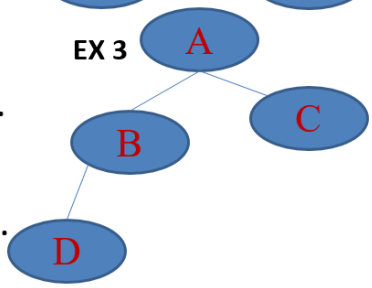


- In EX1 -'A' is the E-node.
- In EX2-'A' is the E-node.
- In EX3- 'B' is the E-node.

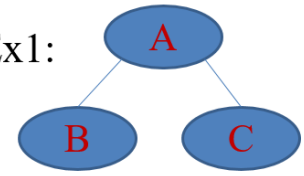
EX2:



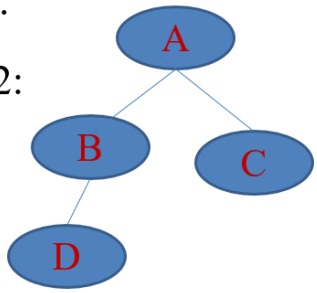
- EX 3



- **DEAD NODE -** A node that is either not to be expanded further, or for which all of its children have been generated.
- Ex1:



Ex2:



- In Ex1-A,B & C areDead nodes
- In Ex2:A,C & D are Dead nodes

- **DEPTH FIRST NODE GENERATION-**
  In this, as soon as a new child 'C' of the current E-node 'R' is generated, 'C' will become the new E-node.
- **BOUNDING FUNCTION** - will be used to kill live nodes without generating all their children.
- **BACTRACKING**-is depth – first node generation with bounding functions.

- **BRANCH-and-BOUND** is a method in which E-node remains E-node until it is dead.
- **BREADTH-FIRST-SEARCH** : Branch-and Bound with each new node placed in a queue .The front of the queen becomes the new E-node.
- **DEPTH-SEARCH (D-Search) :** New nodes are placed in to a stack.The last node added is the first to be explored.

Applications:

1. **N Queens Problem.**
2. **Hamiltonian Cycle.**
3. **Brace Expansion.**
4. **Gray Code.**
5. **Path with Maximum Gold.**
6. **Generalized Abbreviation.**
7. **Campus Bikes II.**

1. N Queens Problem:

N Queen problem is the classical Example of backtracking. N-Queen problem is defined as, "given N x N chess board, arrange N queens in such a way that no two queens attack each other by being in same row, column or diagonal".

- **For N = 1, this is trivial case. For N = 2 and N = 3, solution is not possible**. So we start with N = 4 and we will generalize it for N queens.

Example: 4-Queen Problem

Problem : Given 4 x 4 chessboard, arrange four queens in a way, such that no two queens attack each other. That is, no two queens are placed in the same row, column, or diagonal.

- We have to arrange four queens, Q1, Q2, Q3 and Q4 in 4 x 4 chess board. We will put ith queen in ith row. Let us start with position (1, 1). Q1 is the only queen, so there is no issue. partial solution is <1>
- We cannot place Q2 at positions (2, 1) or (2, 2). Position (2, 3) is acceptable. partial solution is <1, 3>.
- Next, Q3 cannot be placed in position (3, 1) as Q1 attacks her. And it cannot be placed at (3, 2), (3, 3) or (3, 4) as Q2 attacks her. There is no way to put Q3 in third row. Hence, the algorithm backtracks and goes back to the previous solution and readjusts the position of queen Q2. Q2 is moved from positions (2, 3) to (2, 4). Partial solution is <1, 4>
- Now, Q3 can be placed at position (3, 2). Partial solution is <1, 4, 3>.
- Queen Q4 cannot be placed anywhere in row four. So again, backtrack to the previous solution and readjust the position of Q3. Q3 cannot be placed on (3, 3) or(3, 4). So the algorithm backtracks even further.
- All possible choices for Q2 are already explored, hence the algorithm goes back to partial solution <1> and moves the queen Q1 from (1, 1) to (1, 2). And this process continues until a solution is found.
- All possible solutions for 4-queen are shown in fig (a) & fig. (b)

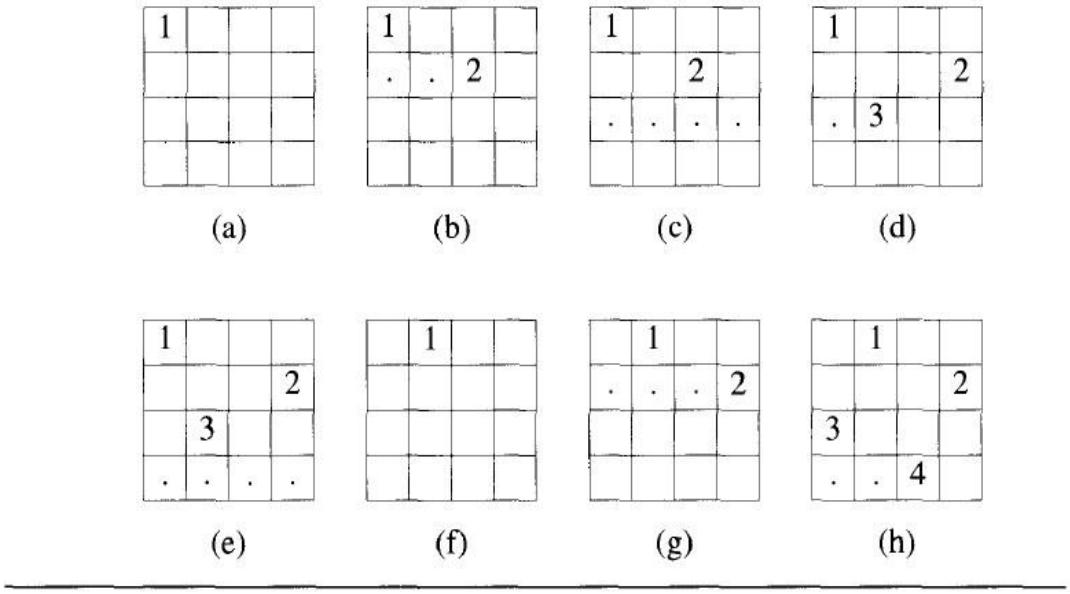


Figure 7.5 Example of a backtrack solution to the 4-queens problem

- We can see that backtracking is a simple brute force approach, but it applies some intelligence to cut out unnecessary computation. **The solution tree for the 4-queen problem is shown in Fig. (7.2)**

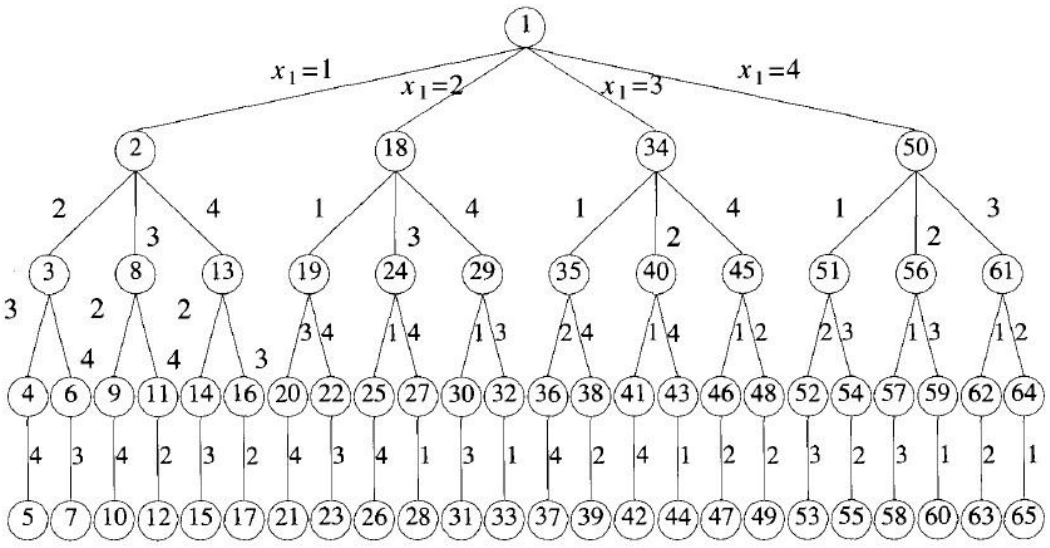


Figure 7.2 Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

Fig. (7.6) describes the backtracking sequence for the 4-queen problem.

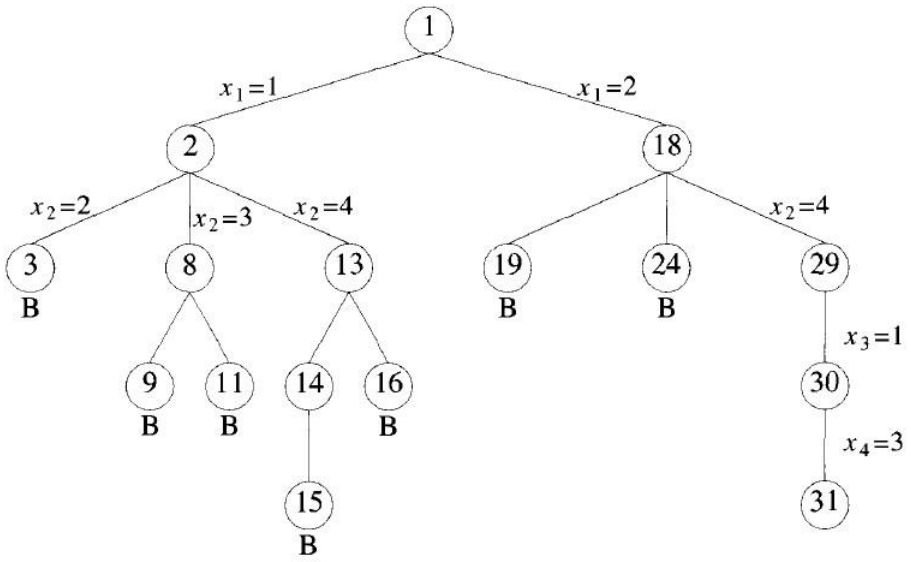


Figure 7.6 Portion of the tree of Figure 7.2 that is generated during backtracking

- The solution of the 4-queen problem can be seen as four tuples (x_1, x_2, x_3, x_4), where x_i represents the column number of queen Q_i. Two possible solutions for the 4-queen problem are **(2, 4, 1, 3) and (3, 1, 4, 2).**

TimeComplexity: O(N!)
Auxiliary Space: O(N^2)

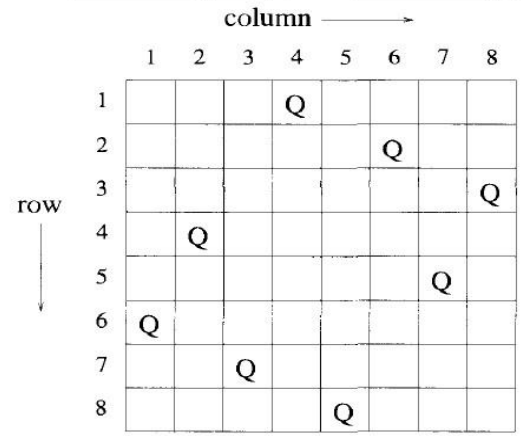


Figure 7.1 One solution to the 8-queens problem

Pseudo Code of N-Queen Problem:

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9                or (Abs(x[j] − i) = Abs(j − k)))
10                   // or in the same diagonal
11               then return false;
12      return true;
13  }
```

Algorithm 7.4 Can a new queen be placed?

```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
```

Java Program for N-Queen Application: **NQueenProblem.java**

```
import java.util.*;

public class NQueenProblem
{
   int N;
   void printSolution(int board[][])
   {
      for (int i = 0; i < N; i++)
      {
         for (int j = 0; j < N; j++)
            System.out.print(board[i][j]);
         System.out.println();
      }
   }

   boolean isSafe(int board[][], int row, int col)
   {
      int i, j;

      for (i = 0; i < col; i++)
         if (board[row][i] == 1)
            return false;

      for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
         if (board[i][j] == 1)
            return false;

      for (i = row, j = col; j >= 0 && i < N; i++, j--)
         if (board[i][j] == 1)
            return false;

      return true;
   }
```

```
boolean solveNQUtil(int board[][], int col)
{
   if (col >= N)
      return true;

   for (int i = 0; i < N; i++)
      {
      if (isSafe(board, i, col)).
      {
         board[i][col] = 1;

         if (solveNQUtil(board, col + 1) == true)
            return true;

         board[i][col] = 0;
      }
   }

   return false;
}

boolean solveNQ()
{
   int board[][] = new int[N][N];

   if (solveNQUtil(board, 0) == false)
      {
      System.out.print("No Solution");
      return false;
   }

   printSolution(board);
   return true;
}
```

```
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        NQueenProblem Queen = new NQueenProblem();
        Queen.N=sc.nextInt();
        Queen.solveNQ();
    }
}
```

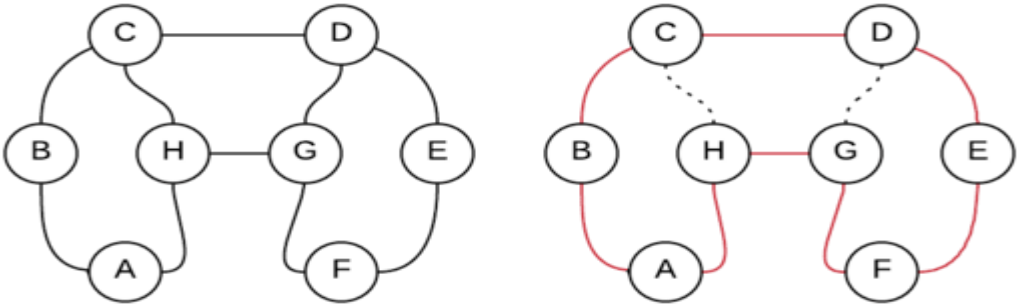
Ouput:

```
case =1
input =1
output ="1"

case =2
input =3
output =""

case =3
input =4
output ="0100
0001
1000
0010"
output ="0010
1000
0001
0100"
```

2. Hamiltonian Cycle:

*The **Hamiltonian cycle** is the cycle in the graph which visits all the vertices in graph exactly once and terminates at the starting node. It may not include all the edges*

- **The** Hamiltonian cycle problem is the problem of finding a Hamiltonian cycle in a graph if there exists any such cycle.
- The input to the problem is an undirected, connected graph. For the graph shown in Figure (a), a path A – B – C – D – E – F—G—H—A forms a Hamiltonian cycle. It visits all the vertices exactly once.



-
  - The Hamiltonian cycle problem is also both, decision problem and an optimization problem. A decision problem is stated as, "Given a path, is it a Hamiltonian cycle of the graph?".
  - The optimization problem is stated as, "Given graph G, find the Hamiltonian cycle for the graph."
- We can define the constraint for the Hamiltonian cycle problem as follows:
  - In any path, vertex i and (i + 1) must be adjacent.
  - 1st and (n – 1)th vertex must be adjacent (nth of cycle is the initial vertex itself).
  - Vertex i must not appear in the first (i – 1) vertices of any path.
- With the adjacency matrix representation of the graph, the adjacency of two vertices can be verified in constant time.

Complexity Analysis:

Looking at the state space graph, in worst case, total number of nodes in tree would be,

$T(n) = 1 + (n – 1) + (n – 1)^2 + (n – 1)^3 + … + (n – 1)^{n – 1}$
= frac{(n-1)^n - 1}{n - 2}=*frac*(*n*−1)*n*−1*n*−2

T(n) = O(n^n). Thus, the Hamiltonian cycle algorithm runs in exponential time.

Pseudo Code Of Hamiltonian Cycle:

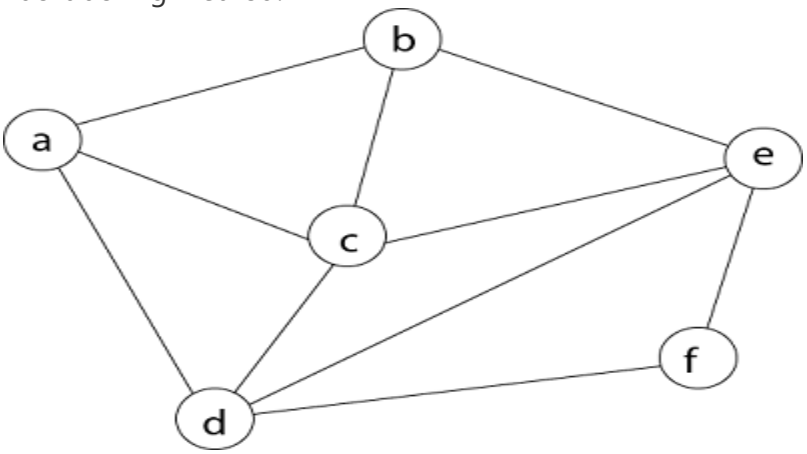
```
1   Algorithm Hamiltonian(k)
2   // This algorithm uses the recursive formulation of
3   // backtracking to find all the Hamiltonian cycles
4   // of a graph. The graph is stored as an adjacency
5   // matrix G[1 : n, 1 : n]. All cycles begin at node 1.
6   {
7       repeat
8       { // Generate values for x[k].
9           NextValue(k); // Assign a legal next value to x[k].
10          if (x[k] = 0) then return;
11          if (k = n) then write (x[1 : n]);
12          else Hamiltonian(k + 1);
13      } until (false);
14  }
```

Algorithm 7.10 Finding all Hamiltonian cycles

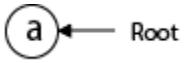
```
1   Algorithm NextValue(k)
2   // x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
3   // no vertex has as yet been assigned to x[k]. After execution,
4   // x[k] is assigned to the next highest numbered vertex which
5   // does not already appear in x[1 : k − 1] and is connected by
6   // an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
7   // in addition x[k] is connected to x[1].
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
12          if (x[k] = 0) then return;
13          if (G[x[k − 1], x[k]] ≠ 0) then
14          { // Is there an edge?
15              for j := 1 to k − 1 do if (x[j] = x[k]) then break;
16                              // Check for distinctness.
17              if (j = k) then // If true, then the vertex is distinct.
18                  if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
19                      then return;
20          }
21      } until (false);
22  }
```

Algorithm 7.9 Generating a next vertex

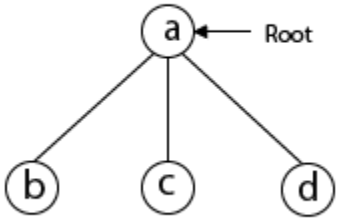
Example: Consider a graph G = (V, E) shown in fig. we have to find a Hamiltonian circuit using Backtracking method.



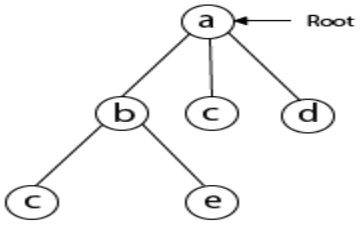
Firstly, we start our search with vertex 'a.' this vertex 'a' becomes the root of our implicit tree.



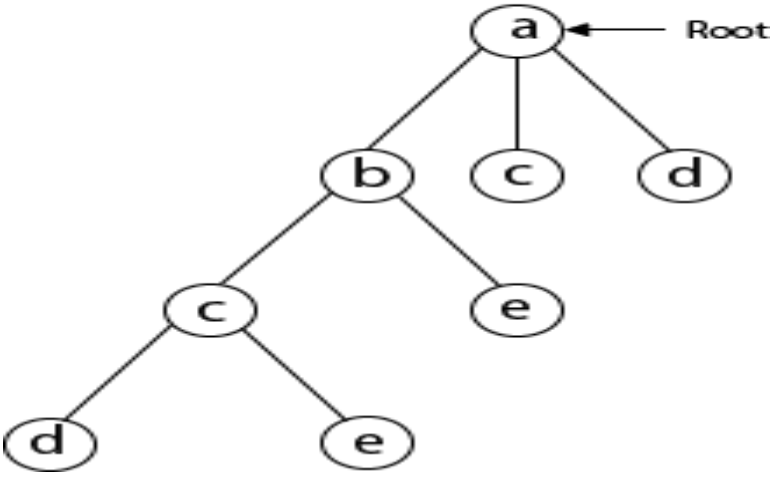
Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d).



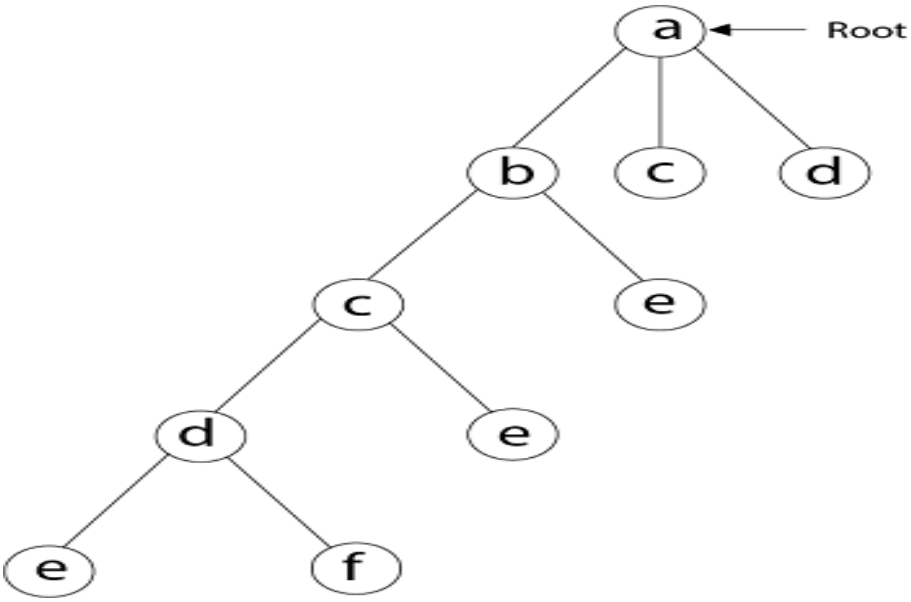
Next, we select 'c' adjacent to 'b.'



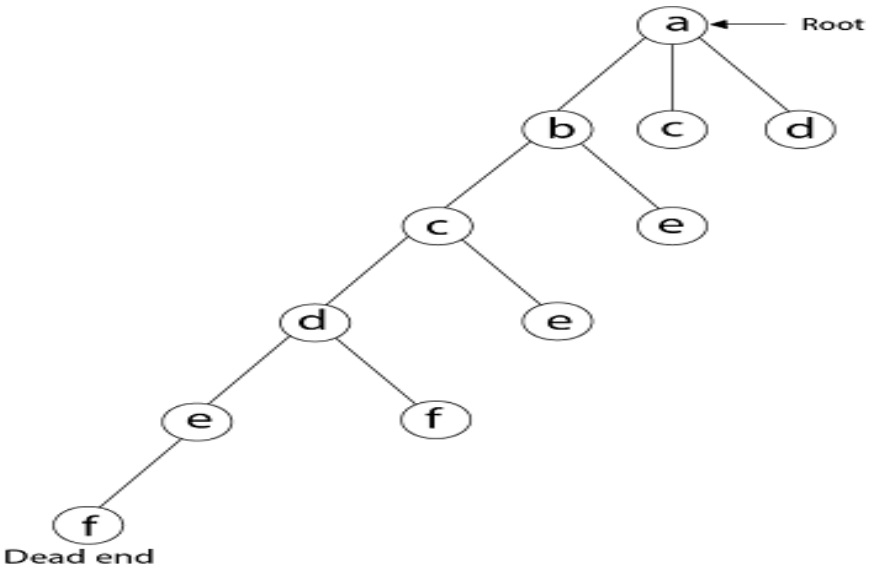
Next, we select 'd' adjacent to 'c.'



Next, we select 'e' adjacent to 'd.'

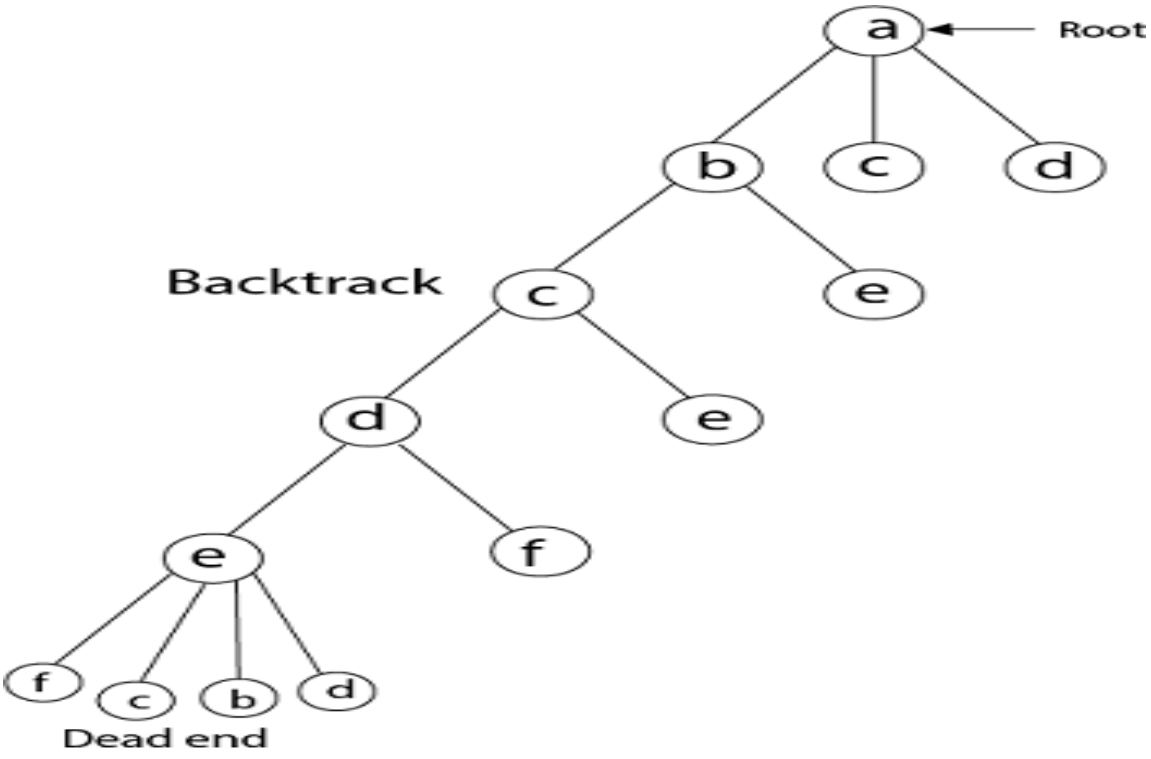


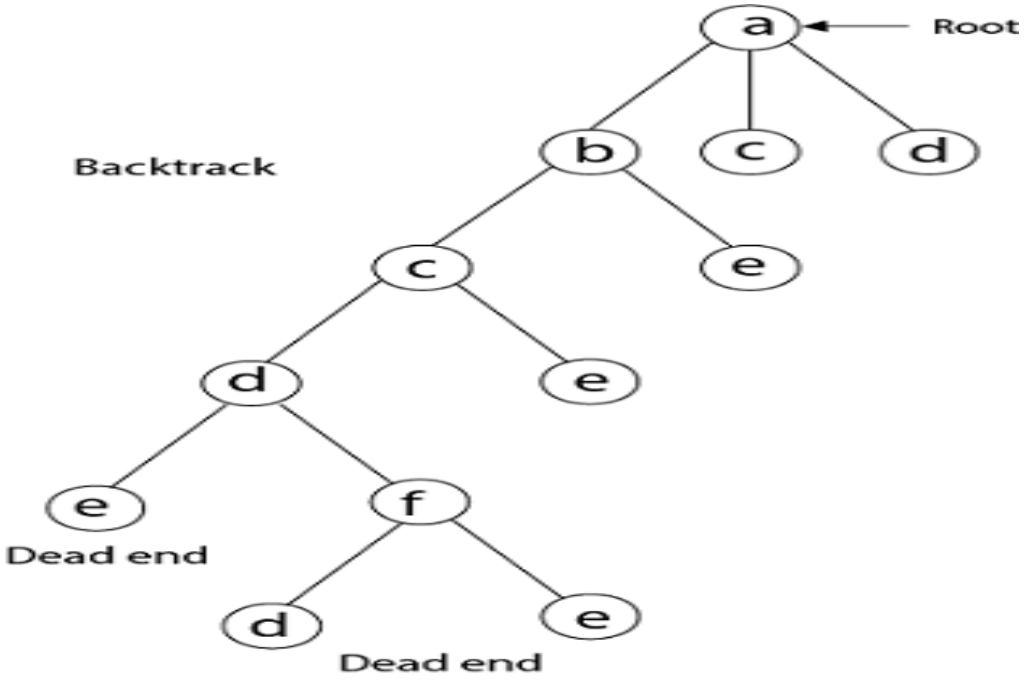
Next, we select vertex 'f' adjacent to 'e.' The vertex adjacent to 'f' is d and e, but they have already visited. Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution.



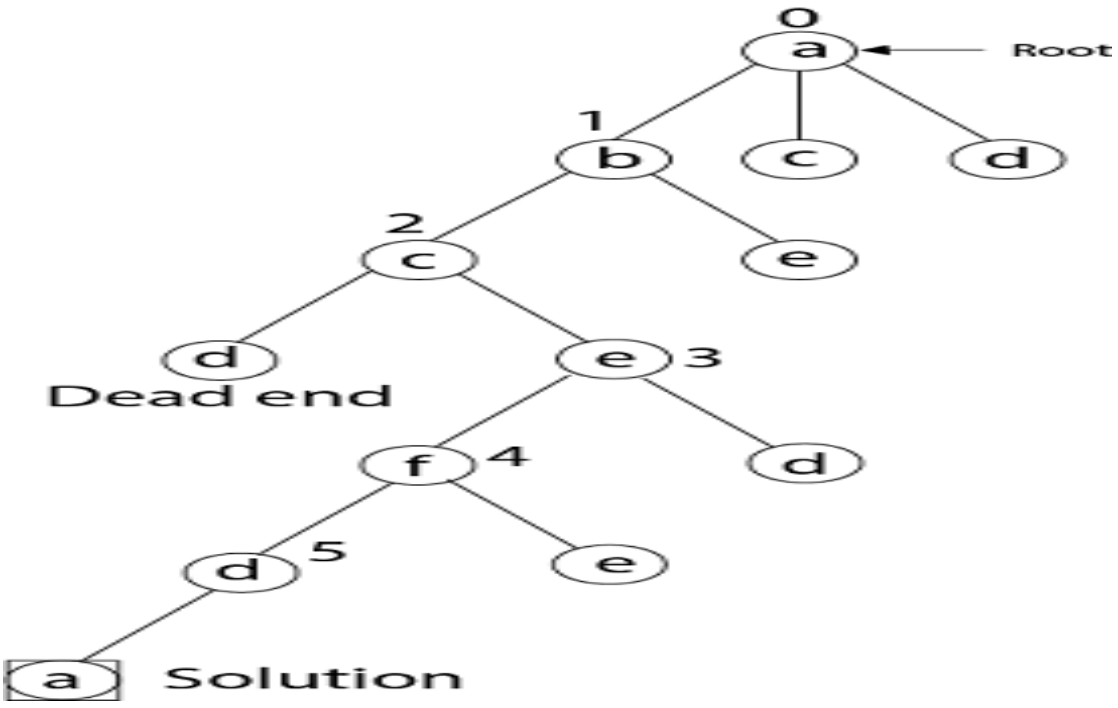
From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to d are e, f from which e has already been checked, and adjacent of 'f' are d and e. If 'e' vertex, revisited them we get a dead state. So again we backtrack one step.

Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a.' Here, we get the Hamiltonian Cycle as all the vertex other than the start vertex 'a' is visited only once. (a - b - c - e - f -d - a).





Again Backtrack



Here we have generated one Hamiltonian circuit, but another Hamiltonian circuit can also be obtained by considering another vertex.

Java Program for Hamiltonian cycle: **HamiltonianCycle.java**

```
import java.util.*;

class HamiltonianCycle
{
   static int V ;
   int path[];

   boolean isSafe(int v, int graph[][], int path[], int pos)
   {
      if (graph[path[pos - 1]][v] == 0)
         return false;

      for (int i = 0; i < pos; i++)
         if (path[i] == v)
            return false;

      return true;
   }

   boolean hamCycleUtil(int graph[][], int path[], int pos)
   {
      if (pos == V)
      {
         if (graph[path[pos - 1]][path[0]] == 1)
            return true;
         else
            return false;
      }

      for (int v = 1; v < V; v++)
      {
         if (isSafe(v, graph, path, pos))
         {
            path[pos] = v;

            if (hamCycleUtil(graph, path, pos + 1) == true)
               return true;

            path[pos] = -1;
         }
      }
```

```
        return false;
    }

    int hamCycle(int graph[][])
    {
        path = new int[V];
        for (int i = 0; i < V; i++)
            path[i] = -1;

        path[0] = 0;
        if (hamCycleUtil(graph, path, 1) == false)
        {
            System.out.println("\nNo Solution");
            return 0;
        }

        printSolution(path);
        return 1;
    }

    void printSolution(int path[])
    {
        for (int i = 0; i < V; i++)
            System.out.print(" " + path[i] + " ");

        System.out.println(" " + path[0] + " ");
    }

    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        V = sc.nextInt()
        int graph[][]=new int[V][V];
        for(int i=0;i<V;i++)
        for(int j=0;j<V;j++)
            graph[i][j]=sc.nextInt();
        System.out.println(new HamiltonianCycle().hamCycle(graph));
    }
}
```

Sample Input-1:

5
0 1 0 1 0
1 0 1 1 1
0 1 0 0 1
1 1 0 0 1
0 1 1 1 0

Sample Output-1:
---
0 1 2 4 3 0

Sample Input-2:

5
0 1 0 1 0
1 0 1 1 1
0 1 0 0 1
1 1 0 0 0
0 1 1 0 0

Sample Output-2:

No Solution

3. Brace Expansion:

Under the grammar given below, strings can represent a set of lowercase words. Let R(expr) denote the set of words the expression represents.

The grammar can best be understood through simple examples:

- Single letters represent a singleton set containing that word.
  - R("a") = {"a"}
  - R("w") = {"w"}
- When we take a comma-delimited list of two or more expressions, we take the union of possibilities.
  - R("{a,b,c}") = {"a","b","c"}
  - R("{{a,b},{b,c}}") = {"a","b","c"} (notice the final set only contains each word at most once)
- When we concatenate two expressions, we take the set of possible concatenations between two words where the first word comes from the first expression and the second word comes from the second expression.
  - R("{a,b}{c,d}") = {"ac","ad","bc","bd"}
  - R("a{b,c}{d,e}f{g,h}") = {"abdfg", "abdfh", "abefg", "abefh", "acdfg", "acdfh", "acefg", "acefh"}

Formally, the three rules for our grammar:

- For every lowercase letter x, we have R(x) = {x}.
- For expressions e_1, e_2, ... , e_k with k >= 2, we have R({e_1, e_2, ...}) = R(e_1) ∪ R(e_2) ∪ ...
- For expressions e_1 and e_2, we have R(e_1 + e_2) = {a + b for (a, b) in R(e_1) × R(e_2)}, where + denotes concatenation, and × denotes the cartesian product.

Given an expression representing a set of words under the given grammar, return *the sorted list of words that the expression represents*.

Example 1:

Input: expression = "{a,b}{c,{d,e}}"

Output: ["ac","ad","ae","bc","bd","be"]

Example 2:

Input: expression = "{{a,z},a{b,c},{ab,z}}"

Output: ["a","ab","ac","z"]

Explanation: Each distinct word is written only once in the final answer.

Constraints:

- 1 <= expression.length <= 60
- expression[i] consists of '{', '}', ','or lowercase English letters.
- The given expression represents a set of words based on the grammar given in the description.

Java Program for BraceExpansion: BraceExpression.java

```
import java.util.*;

public class BraceExpression
{
    public String[] expand(String S)
    {
     List<String> res = new ArrayList<>();
     dfs(S, 0, new StringBuilder(), res);

     String[] out = new String[res.size()];
     for (int i = 0; i < res.size(); i++) { out[i] = res.get(i); }
     return out;
    }

    private void dfs(String s, int index, StringBuilder sb, List<String> res)
    {
        if (index == s.length())
        {
        if (sb.length() > 0) { res.add(sb.toString()); }
        return;
        }

     char c = s.charAt(index);
     int position = sb.length();
     if (c == '[') {
        List<Character> charList = new ArrayList<>();
        int endIndex = index + 1;
        while (endIndex < s.length() && s.charAt(endIndex) != ']') {
           if (Character.isLetter(s.charAt(endIndex))) { charList.add(s.charAt(endIndex));
}
```

```
            endIndex++;
             }

        Collections.sort(charList);
        for (char d : charList) {
            sb.append(d);
            dfs(s, endIndex + 1, sb, res);
            sb.setLength(position);
             }

        } else if (Character.isLetter(c)) {
        sb.append(s.charAt(index));
        dfs(s, index + 1, sb, res);
        }
    }
     public static void main(String args[] )
 {
              Scanner sc = new Scanner(System.in);
             String str=sc.next();
             System.out.println(Arrays.deepToString(expand(str)));
    }
}
```

case =1
input =[a,b,c,d]e[x,y,z]
output =[aex, aey, aez, bex, bey, bez, cex, cey, cez, dex, dey, dez]

case =2
input =[ab,cd]x[y,z]
output =[abxy, abxz, cdxy, cdxz]

4. Gray Code:

An **n-bit gray code sequence** is a sequence of 2^n integers where:

- Every integer is in the **inclusive** range $[0, 2^n - 1]$,
- The first integer is 0,
- An integer appears **no more than once** in the sequence,
- The binary representation of every pair of **adjacent** integers differs by **exactly one bit**, and
- The binary representation of the **first** and **last** integers differs by **exactly one bit**.

Given an integer n, return any valid **n-bit gray code sequence**.

Example 1:

Input: n = 2

Output: [0,1,3,2]

Explanation:

The binary representation of [0,1,3,2] is [00,01,11,10].

- 0<u>0</u> and 0<u>1</u> differ by one bit

- <u>0</u>1 and <u>1</u>1 differ by one bit

- 1<u>1</u> and 1<u>0</u> differ by one bit

- <u>1</u>0 and <u>0</u>0 differ by one bit

[0,2,3,1] is also a valid gray code sequence, whose binary representation is [00,10,11,01].

- <u>0</u>0 and <u>1</u>0 differ by one bit

- 1<u>0</u> and 1<u>1</u> differ by one bit

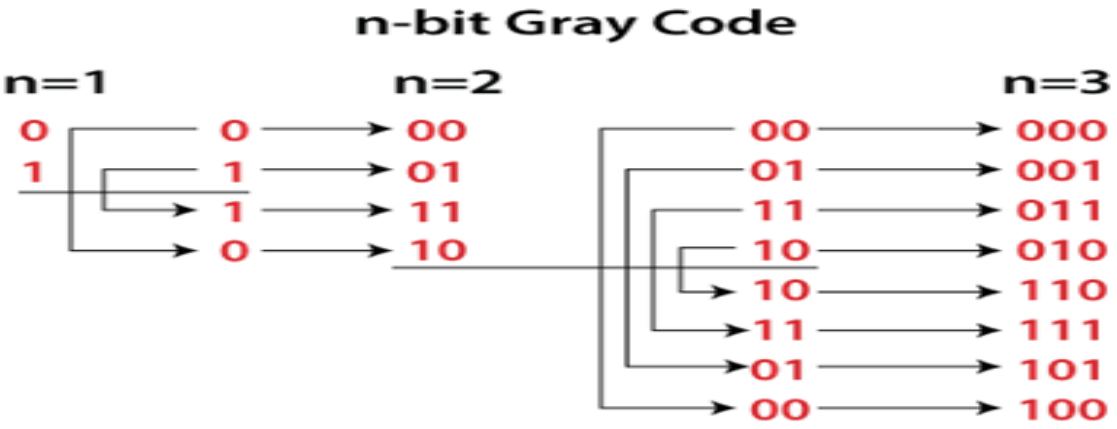
- <u>1</u>1 and <u>0</u>1 differ by one bit

- 0<u>1</u> and 0<u>0</u> differ by one bit

Example 2:

Input: n = 1

Output: [0,1]



Java program for Gray Code: **GrayCode.java**

```
import java.util.*;

class GrayCode
 {
   int nums = 0;
   public List<Integer> grayCode(int n)
   {
      List<Integer> ret = new ArrayList<>();
      backTrack(n, ret);
      return ret;
   }

   private void backTrack(int n, List<Integer> ret)
    {
      if (n == 0)
     {
         ret.add(nums);
         return;
      }
      else
      {
         backTrack(n - 1, ret);
         nums=num ^ (1 << n - 1);  //using X-OR with left shift
         backTrack(n - 1, ret);
      }
```

```
    }
    public static void main( String args[])
    {
        Scanner sc=new Scanner(System.in);
        int N=sc.nextInt();
        System.out.println(new GrayCode().grayCode(N));
    }
}
```

Output:

case =1
input =2
output =[0, 1, 3, 2]

case =2
input =3
output =[0, 1, 3, 2, 6, 7, 5, 4]

5. Path with Maximum Gold:

In a gold mine grid of size m x n, each cell in this mine has an integer representing the amount of gold in that cell, 0 if it is empty.

Return the maximum amount of gold you can collect under the conditions:

- Every time you are located in a cell you will collect all the gold in that cell.
- From your position, you can walk **one step to the left, right, up, or down.**
- You can't visit the same cell more than once.
- Never visit a cell with 0 gold.
- You can start and stop collecting gold from **any** position in the grid that has some gold.

Example 1:

Input: grid = [[0,6,0],[5,8,7],[0,9,0]]

Output: 24

Explanation:

[[0,6,0],

[5,8,7],

[0,9,0]]

Path to get the maximum gold, 9 -> 8 -> 7.

Example 2:

Input: grid = [[1,0,7],[2,0,6],[3,4,5],[0,3,0],[9,0,20]]

Output: 28

Explanation:

[[1,0,7],

[2,0,6],

[3,4,5],

[0,3,0],

[9,0,20]]

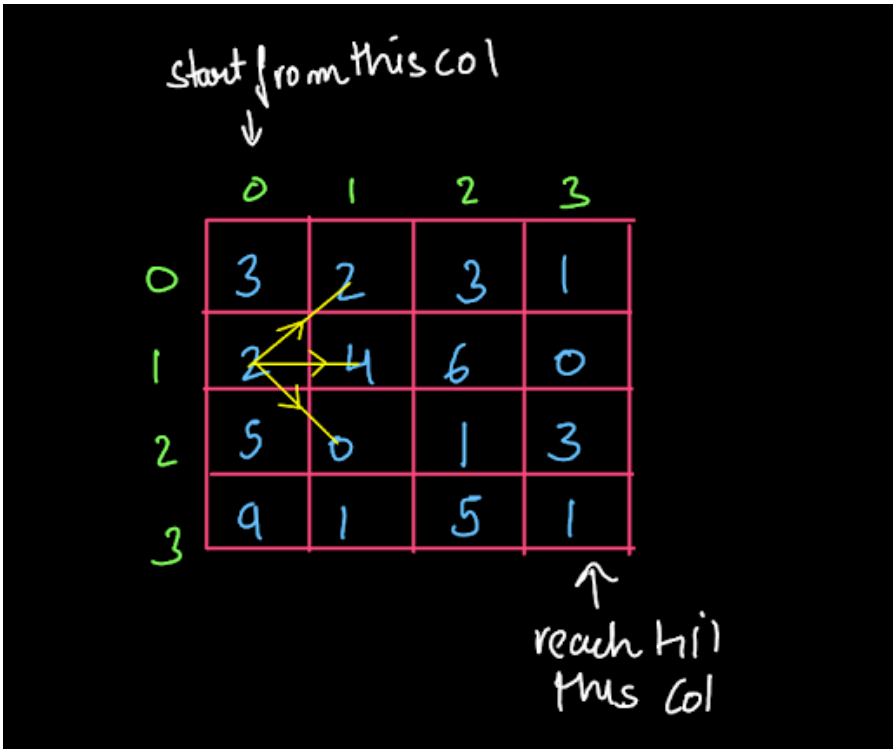
Path to get the maximum gold, 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7.

Constraints:

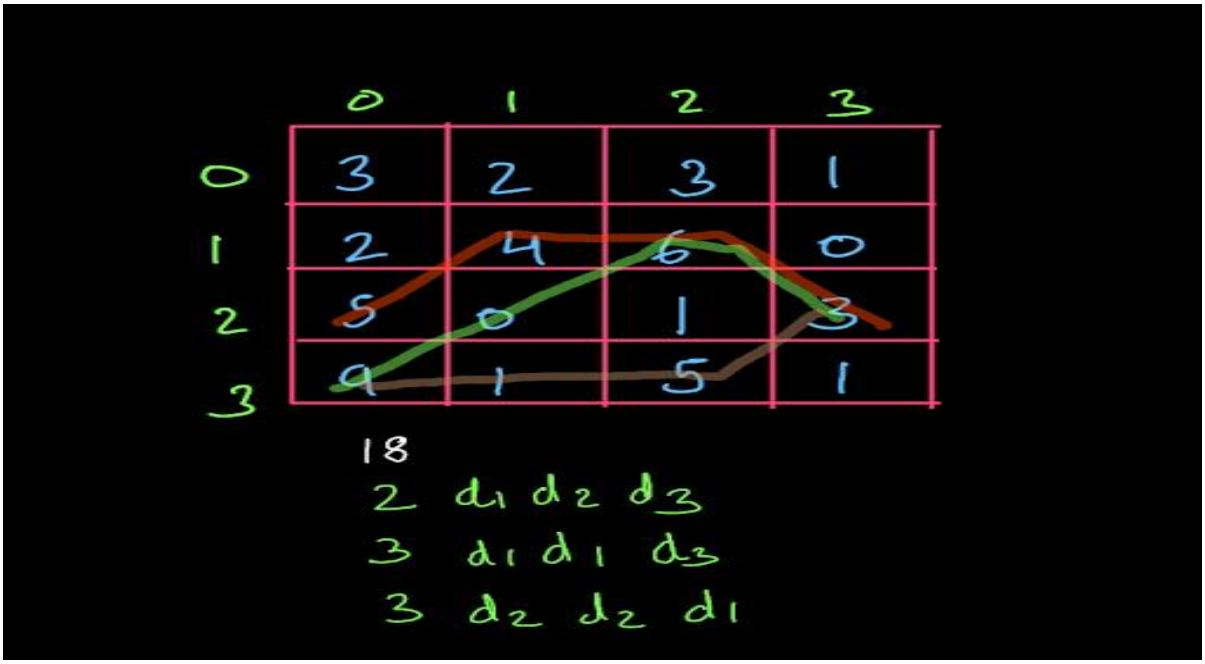
- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 15
- 0 <= grid[i][j] <= 100
- There are at most **25** cells containing gold.

Time Complexity: O(n x m) where n is the number of rows and m is the number of columns of the given input matrix.

Space Complexity: O(n x m)



- Each cell in this matrix has an amount of gold present at it. We have to start digging from the 0^{th} column and reach the last column and in the procedure, we can move either diagonally one step upward (called "d1") or one step forward in the same row (called "d2") or diagonally one step downward (called "d3") collecting the gold at the cell we reach. So, we have to find the paths with the maximum gold.
- For instance, in the matrix shown above, the maximum gold that we can collect is 18 and there are 3 different paths to reach 18 as shown in the image below:



- As you can see, the maximum amount of gold is 18. The first path is "2 d1 d2 d3". This means that we are starting from the 2nd row and the path is "d1 d2 d3".
- We have already discussed the meaning of "d1", "d2" and "d3". Why are we mentioning only the starting row and not the starting column in the paths? This is because we have already been told in the question that we have to start from the 0th column only.

Java program for Path with Maximum Gold: **GetMaximumGold.java**

```
import java.util.*;

class GetMaximumGold
{
   public int getMaximumGold(int[][] grid)
  {
      int maxGold = 0;
      for (int i = 0; i < grid.length; i++) {
         for (int j = 0; j < grid[0].length; j++) {
            maxGold = Math.max(maxGold, getMaximumGoldBacktrack(grid, i, j, 0));
         }
      }
      return maxGold;
   }

   private int getMaximumGoldBacktrack(int[][] grid, int i, int j, int curGold) {
      if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length || grid[i][j] == 0)
         return curGold;
      curGold += grid[i][j];
      int temp = grid[i][j];
      int maxGold = curGold;

      grid[i][j] = 0;
      maxGold = Math.max(maxGold, getMaximumGoldBacktrack(grid, i+1, j,
curGold));
      maxGold = Math.max(maxGold, getMaximumGoldBacktrack(grid, i, j+1,
curGold));
      maxGold = Math.max(maxGold, getMaximumGoldBacktrack(grid, i-1, j, curGold));
      maxGold = Math.max(maxGold, getMaximumGoldBacktrack(grid, i, j-1, curGold));
      grid[i][j] = temp;

      return maxGold;
   }

   public static void main(String args[])
   {
```

```
        Scanner sc=new Scanner(System.in);
        int m=sc.nextInt();
        int n=sc.nextInt();
        int grid[][]=new int[m][n];
        for(int i=0;i<m;i++)
            for(int j=0;j<n;j++)
                grid[i][j]=sc.nextInt();
        System.out.println(new GetMaximumGold().getMaximumGold(grid));
    }
}
```

Output:

input =3 3
0 6 0
5 8 7
0 9 0
output =24

case =2
input =5 3
1 0 7
2 0 6
3 4 5
0 3 0
9 0 20
output =28

6. Generalized Abbreviation:

Write a function to generate the generalized abbreviations of a word.

- **Note:** The order of the output does not matter.
- **Example:**
- **Input:** "word"
  Output:
  ["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

Understanding the problem:

1. You are given a word.
2. You have to generate all abbreviations of that word.

For example:
Sample Input-> pep

Sample Output-> pep pe1 p1p p2 1ep 1e1 2p 3 (in different lines)
HOW?
First of all, generate all the binaries of the length equal to the length of the input string.

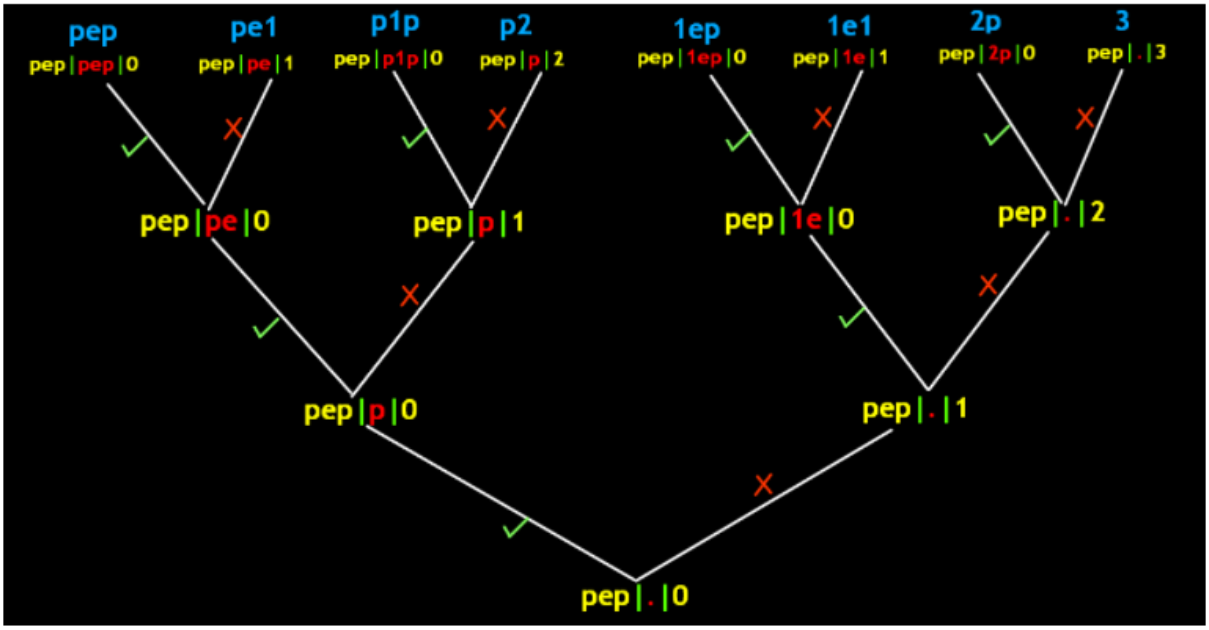
Binaries -

000 -> pep
001 -> pe1
010 -> p1p
011 -> p2
100 -> 1ep
101 -> 1e1
110 -> 2p
111 -> 3

Approach:

- We can observe that whenever the bit is OFF (zero), we will use the character of the string at that position and whenever the bit is ON (one), we will count all the ON (one) bits that are together and then replace them with the count of the ON (one) bits.
- The number of abbreviations will be equal to 2^n, where n = length of the string (as the number of binaries with given n is equal to 2^n).
- In the above example n = 3, therefore the number of abbreviations will be 8 (2^3 = 8).
- We can see that this question is like the subsequence problem, we must maintain one more variable which will count the characters that were not included in the subsequence.

Let's try to draw the recursion tree diagram for the word, "pep". In recursion diagram, (a | b | c) a represents the test string, b represents the answer so far and c represents the count.



- In the above diagram, (pep|.|0) has two options, either it can include the first p in the answer or not. If it includes the p in the answer then p is concatenated in the answer so far string and count remains 0.
- Then at (pep|p|0), we further explore the options for character e. It has two options, either it can include e in the answer or not. If it includes e in the answer then e is concatenated in the answer so far string and count remains 0.
- Then at (pep|pe|0), we further explore the options for character p (second p). It has two options, either it can include p in the answer or not. If it includes p in the answer then p is concatenated in the answer so far string and count remains 0. **Since we have reached the end of the string, and the count is 0, therefore, the asf i.e. pep is the first possible output.**
- Then we come back at (pep|pe|0) and explore the path if the second p is not added in the asf. In this case, p will not be added in psf and therefore count will become 1.

And again, we have reached the end of the string, but count is 1 this time, therefore, this count will be concatenated at the end of the asf, which will give us the next possible output, pe1.

- We have explored both the possibilities via (pep|pe|0) therefore, we go back at (pep|p|0), and see what happens if e was not added to asf.
- First of all count becomes 1, and we reach (pep|p|1).
- Then at (pep|p|1), we further explore the options for character p (second p). It has two options, either it can include p in the answer or not. If p is to be included in the answer then we see that our previous count is not 0, so we concatenate the previous count in the asf and then second p.
  Since we have reached the end of the string, and the count is 0, therefore, the asf i.e. p1p is the next possible output.
- Then we come back at (pep|p|1) and explore the path if the second p is not added in the asf. In this case, p will not be added in psf and therefore count will become 2.
  And again, we have reached the end of the string, but count is 2 this time, therefore, this count will be concatenated at the end of the asf, which will give us the next possible output, p2.
- We have explored both the possibilities via (pep|p|1) therefore; we go back at (pep|p|0).
- (pep|p|0) has also explored both the options, if "e" were added or not) therefore; we go back at (pep|.|0).
- At (pep|.|0), now we explore the possibilities of, if first p were not added in the asf. If "p" is not added in the answer so far then count becomes 1.
- Then at (pep|.|1), we further explore the options for character e. It has two options, either it can include e in the answer or not. If it includes e in the answer then e is concatenated with the current count in the answer so far string and count is set to 0.
- Then at (pep|1e|0), we further explore the options for character p (second p). It has two options, either it can include p in the answer or not. If it includes p in the answer then p is concatenated in the answer so far string and count remains 0.
- Since we have reached the end of the string, and the count is 0, therefore, the asf i.e. 1ep is the next possible output.
- Then we come back at (pep|1e|0) and explore the path if the second p is not added in the asf. In this case, p will not be added in psf and therefore count will become 1.
  And again, we have reached the end of the string, but count is 1 this time, therefore, this count will be concatenated at the end of the asf, which will give us the next possible output, 1e1.
- We have explored both the possibilities via (pep|1e|0) therefore, we go back at (pep|.|1), and see what happens if e was not added to asf.
- Then at (pep|.|2), we further explore the options for character p (second p). It has two options, either it can include p in the answer or not.
  If p is to be included in the answer then we see that our previous count is not 0, so we concatenate the previous count (2) in the asf and then second p.
  Since we have reached the end of the string, and the count is 0, therefore, the asf i.e. 2p is the next possible output.
- Then we come back at (pep|.|2) and explore the path if the second p is not added in the asf. In this case, p will not be added in psf and therefore count will become 3.
  And again, we have reached the end of the string, but count is 3 this time, therefore,

this count will be concatenated at the end of the asf (empty at this stage), which will give us the next possible output, 3.

- We have explored both the possibilities via (pep|.|2) therefore; we go back at (pep|.|1).
- (pep|.|1) has also explored both the options, if "e" were added or not) therefore; we go back at (pep|.|0).
- At (pep|.|0), we have explored both the possibilities that is if first p were not added in the asf.

Java Program for Generalized Abbreviations: **GenerateAbbreviations.java**

```
import java.util.*;

class GenerateAbbreviations
{
   public List<String> makeShortcutWords(String word)
{
       List<String> ret = new ArrayList<String>();
       backtrack(ret, word, 0, "", 0);
       Collections.sort(ret);
       return ret;
    }

   private void backtrack(List<String> ret, String word, int pos, String cur, int count)
  {
       if(pos==word.length())
     {
          if(count > 0) cur += count;
          ret.add(cur);
       }
       else{
          backtrack(ret, word, pos + 1, cur, count + 1);
          backtrack(ret, word, pos+1, cur + (count>0 ? count : "") + word.charAt(pos), 0);
       }
   }

   public static void main(String args[])
   {
```

```
        Scanner sc=new Scanner(System.in);
        String s=sc.next();
        System.out.println(new GenerateAbbreviations().makeShortcutWords(s));
    }
}
```

case =1
input =kmit
output =[1m1t, 1m2, 1mi1, 1mit, 2i1, 2it, 3t, 4, k1i1, k1it, k2t, k3, km1t, km2, kmi1, kmit]

case =2
input =cse
output =[1s1, 1se, 2e, 3, c1e, c2, cs1, cse]

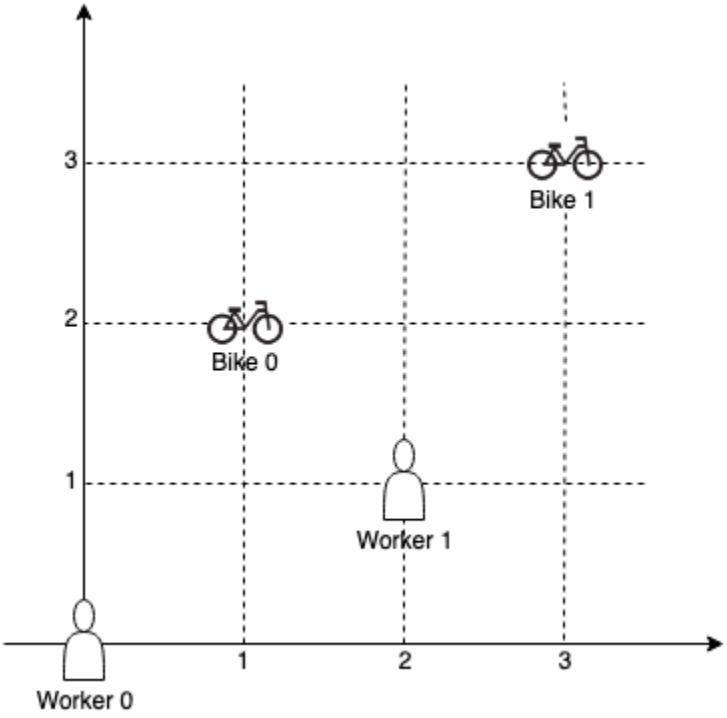
case =3
input =elite
output =[1l1t1, 1l1te, 1l2e, 1l3, 1li1e, 1li2, 1lit1, 1lite, 2i1e, 2i2, 2it1, 2ite, 3t1, 3te, 4e, 5, e1i1e, e1i2, e1it1, e1ite, e2t1, e2te, e3e, e4, el1t1, el1te, el2e, el3, eli1e, eli2, elit1, elite]

case =4
input =r
output =[1, r]

7. Campus Bikes II:

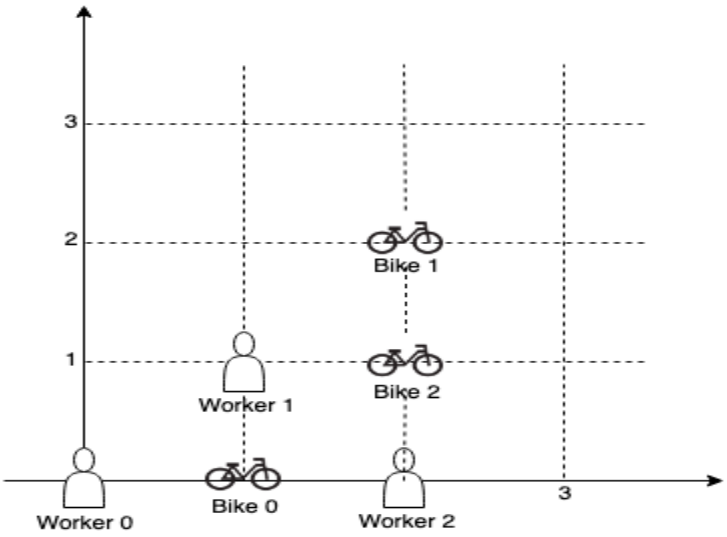
- On a campus represented as a 2D grid, there are N workers and M bikes, with N <= M. Each worker and bike is a 2D coordinate on this grid.
- We assign one unique bike to each worker so that the sum of the Manhattan distances between each worker and their assigned bike is minimized.
- The Manhattan distance between two points p1 and p2 is Manhattan(p1, p2) = |p1.x - p2.x| + |p1.y - p2.y|.
- Return the minimum possible sum of Manhattan distances between each worker and their assigned bike.

Example 1:



Input: workers = [[0,0],[2,1]], bikes = [[1,2],[3,3]]
Output: 6
Explanation:
We assign bike 0 to worker 0, bike 1 to worker 1. The Manhattan distance of both assignments is 3, so the output is 6.

Example 2:



Input: workers = [[0,0],[1,1],[2,0]], bikes = [[1,0],[2,2],[2,1]]
Output: 4
Explanation:
We first assign bike 0 to worker 0, then assign bike 1 to worker 1 or worker 2, bike 2 to worker 2 or worker 1. Both assignments lead to sum of the Manhattan distances as 4.

Note:

1. 0 <= workers[i][0], workers[i][1], bikes[i][0], bikes[i][1] < 1000
2. All worker and bike locations are distinct.
3. 1 <= workers.length <= bikes.length <= 10

Solution:

To solve this, we will follow these steps −

- Define a function helper(). This will take a,b
  - return |a[0]-b[0]| + |a[1] - b[1]|
- Define a function solve(). This will take bikes, workers,bikev,i:= 0
- info := a list with i and bikev
- if info is present in memo, then
  - return memo[info]
- if i is same as size of workers, then
  - return 0
- temp := infinity
- for j in range 0 to size of bikes, do
  - if not bikev[j] is non-zero, then
    - bikev[j]:= 1
    - temp := minimum of temp, helper(workers[i], bikes[j]) +solve(bikes, workers, bikev, i+1)

- bikev[j]:= 0
- memo[info]:= temp
- return temp
- Define a function assignBikes(). This will take workers, bikes
- bikev := a list whose size is same as the size of bikes, fill this with false
- memo:= a new map
- return solve(bikes, workers, bikev)

Java program for Campus Bikes-II: **CampusBikes.java**

```
import java.util.*;

class CampusBikes
{
int min = Integer.MAX_VALUE;
   public int assignBikes(int[][] workers, int[][] bikes)
   {
      backtrack(new boolean[bikes.length],0,workers,bikes,0);
      return min;
   }

   void backtrack(boolean[] visited, int i, int[][] workers, int[][] bikes, int distance)
   {
      if (i==workers.length && distance < min) min = distance;
      if (i>=workers.length) return;
      if (distance>min) return;
      for (int j=0; j<bikes.length; j++){
         if (visited[j]) continue;
         visited[j] = true;
         backtrack(visited, i+1, workers, bikes, distance+dist(i,j,workers,bikes));
         visited[j] = false;
      }
   }

   int dist(int i, int j, int[][] workers, int[][] bikes){
      return Math.abs(workers[i][0]-bikes[j][0])+Math.abs(workers[i][1]-bikes[j][1]);
   }

   public static void main(String[] args) {
```

```
        Scanner sc=new Scanner(System.in);
        int m=sc.nextInt();
        int n=sc.nextInt();
        int bikes[][]=new int[n][2];
        int men[][]=new int[m][2];
        for(int i=0;i<m;i++){
            men[i][0]=sc.nextInt();
            men[i][1]=sc.nextInt();
        }
        for(int i=0;i<n;i++){
            bikes[i][0]=sc.nextInt();
            bikes[i][1]=sc.nextInt();
        }
        System.out.println(new CampusBikes().assignBikes(men,bikes));
    }
}
```

Sample Input-1:

```
3 3    //No of workers and vehicles
0 1     // co-ordinates of workers
1 2
1 3
4 5     // co-ordinates of vehicles
2 5
3 6
```

Sample Output-1:

```
17
```

Sample Input-2:

```
2 2      //No of workers and vehicles
0 0     // co-ordinates of workers
2 1
1 2     // co-ordinates of vehicles
3 3
```

Sample Output-2:

```
6
```