ALGORITHMS and ANALYSIS:

We can have three cases to analyze an algorithm:
1) Worst Case
2) Average Case
3) Best Case

Worst Case Analysis:

- In the worst case analysis, we calculate upper bound on running time of an algorithm.
- We must know the case that causes maximum number of operations to be executed.

e.g.) For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search () functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be Θ (n).

Average Case Analysis:

- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.
- Sum all the calculated values and divide the sum by total number of inputs.
- We must know (or predict) distribution of cases.

e.g.) For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

Best Case Analysis:

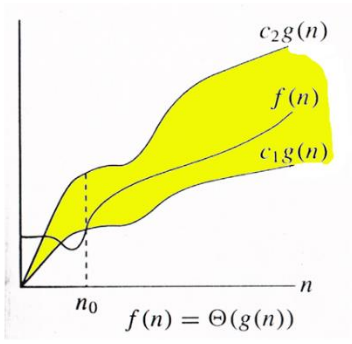
- In the best case analysis, we calculate lower bound on running time of an algorithm.
- We must know the case that causes minimum number of operations to be executed.

e.g.) In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Theta(1)$.

NOTE! Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which tells the maximum time taken by program is good information.

Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.
The following 3 asymptotic notations are mostly used to represent time complexity of algorithms

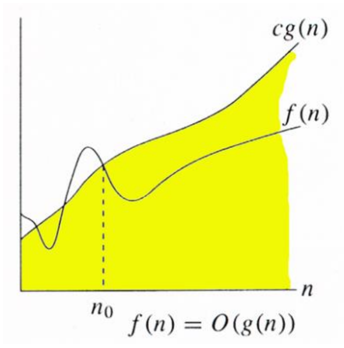
1) **Θ Notation:** The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
   A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants.
   For example, consider the following expression.
   $3n^3 + 6n^2 + 6000 = \Theta(n^3)$
   Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ has higher values than $\Theta n^2)$ irrespective of the constants involved.



For a given function g(n), we denote Θ(g(n)) is following set of functions.
Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

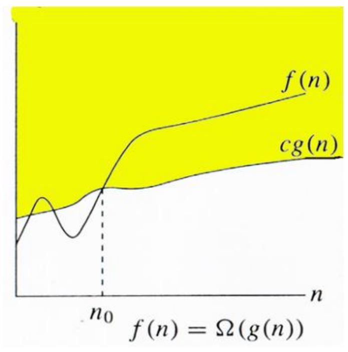
2) **Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.
   For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case.
   We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.
   If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:
   1. The worst case time complexity of Insertion Sort is Θ(n^2).
   2. The best case time complexity of Insertion Sort is Θ(n).



The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and n0 such that 0 <= f(n) <= cg(n) for all n >= n0}

3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.



Ω Notation can be useful when we have lower bound on time complexity of an algorithm. The best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and n0 such that 0 <= cg(n) <= f(n) for all n >= n0}.

Solving Recurrences

Many algorithms are recursive in nature. At the time of analysis we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes.
There are mainly three ways for solving recurrences.

- **Substitution Method**: We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.
  For example consider the recurrence T(n) = 2T(n/2) + n
  We guess the solution as T(n) = O(nLogn). Now we use induction to prove our guess.
  We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.

  ```
  T(n) = 2T(n/2) + n
     <= cn/2Log(n/2) + n
     =  cnLogn - cnLog2 + n
     =  cnLogn - cn + n
     <= cnLogn
  ```

- **Recurrence Tree Method:** In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels.
  To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels.
  The pattern is typically an arithmetic or geometric series.
  For example consider the recurrence relation:
  $T(n) = T(n/4) + T(n/2) + cn^2$

  ```
         cn^2
        /    \
    T(n/4)    T(n/2)
  ```

  If we further break down the expression T(n/4) and T(n/2), we get following recursion tree.

  ```
              cn^2
           /         \
      c(n^2)/16      c(n^2)/4
      /     \         /    \
  T(n/16)    T(n/8)  T(n/8)    T(n/4)
  ```

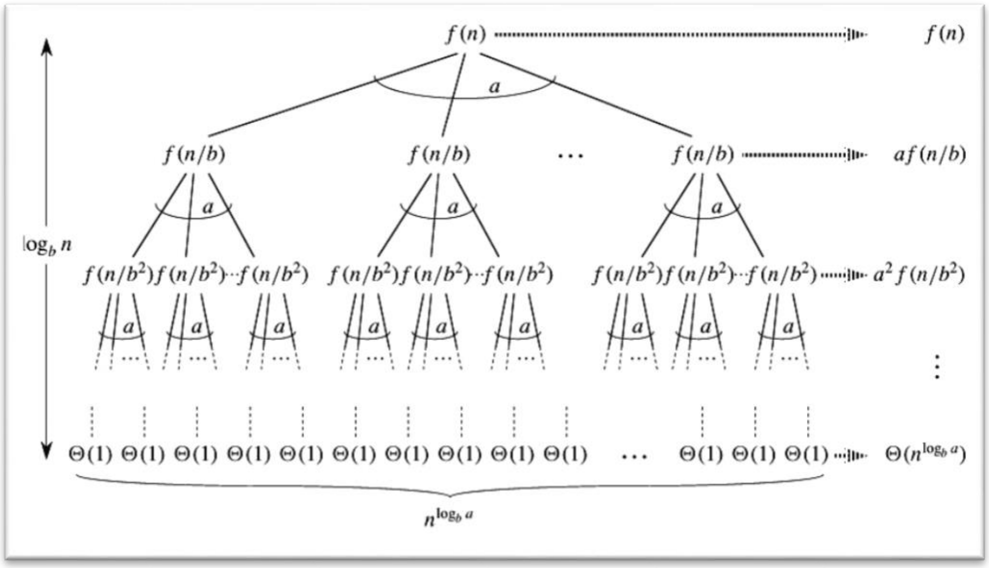
  Breaking down further gives us following

  ```
               cn^2
            /          \
      c(n^2)/16         c(n^2)/4
       /     \          /     \
  c(n^2)/256   c(n^2)/64  c(n^2)/64    c(n^2)/16
   /   \     /   \   /   \      /   \
  ```

To know the value of T (n), we need to calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series
T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...
The above series is geometrical progression with ratio 5/16.
To get an upper bound, we can sum the infinite series.
We get the sum as $(n^2)/(1 - 5/16)$ which is $O(n^2)$.

- **Master Method or Master's Theorem:** Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.
  T(n) = aT(n/b) + f(n) where a >= 1 and b > 1
  There are following three cases:
  - If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$
  - If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$
  - If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

  How does this work?
  Master method is mainly derived from recurrence tree method. If we draw recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at root is f(n) and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of recurrence tree is $\text{Log}_b n$.



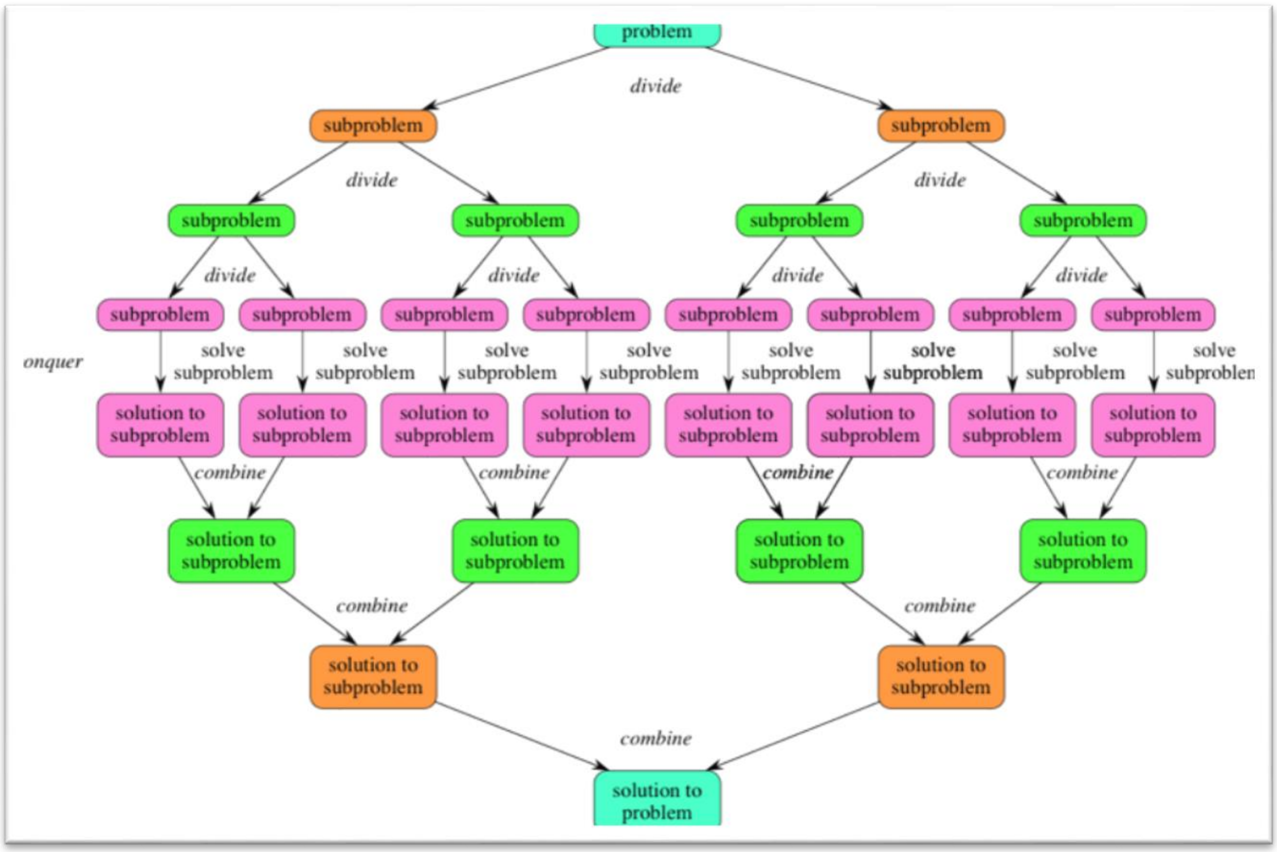
NOTE! In recurrence tree method, we calculate total work done.
If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1).
If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2).
If work done at root is asymptotically more, then our result becomes work done at root (Case 3).

DIVIDE AND CONQUER

In divide and conquer approach, the problem is divided into smaller sub-problems and then each problem is solved independently. A typical Divide and Conquer algorithm solves a problem using following three steps:

- *Divide:* Break the given problem into subproblems of same type.
- *Conquer:* Recursively solve these subproblems
- *Combine:* Appropriately combine the answers

When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved.
The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.



The following computer algorithms are based on divide-and-conquer programming approach −

- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication

SEARCHING

Searching the process of finding an element given by user in an array. For a large database searching is necessary because each element cannot be viewed individually.
Searching can be achieved using many techniques:

- Linear Search
- Binary Search

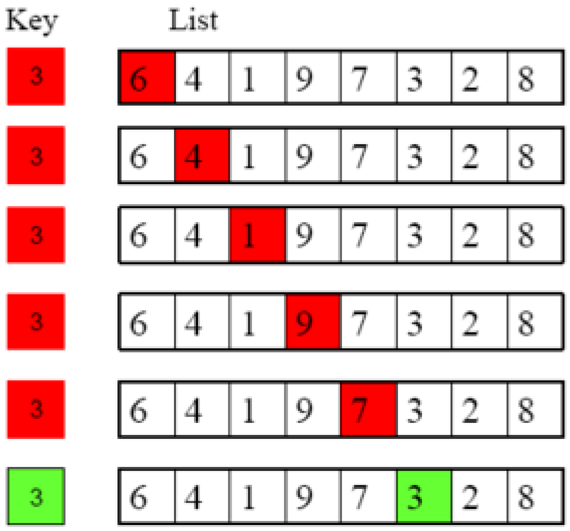
Other searches are also there,

- Jump Search
- Interpolation Search
- Exponential Search
- Ternary Search

We will discuss mainly used two searching techniques, i.e. Linear Search and Binary Search.

LINEAR SEARCH

A linear search is a search in which each element is compared with test variable to get the result.
e.g.) A simple approach is to do linear search, i.e
Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
If x matches with an element, return the index.
If x doesn't match with any of elements, return -1.



This is the most common approach for searching. But it takes time when compared to other searching techniques. Thus, least used.

The time complexity of above algorithm is **O(n).** As the worst case can be that the test value is not present in array, and in comparison the whole array is traversed.
Therefore the complexity depends on the size of array.

Algorithm:

```
for i=0 to i<n
{
 Check if(test_value==array[i])
        found
}
```

`Try implementing code of linear search.`

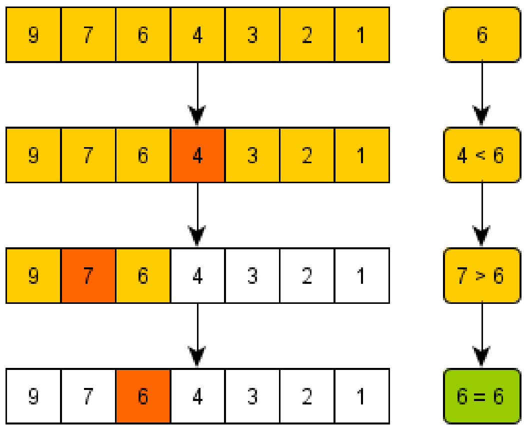
BINARY SEARCH

Binary search searches an element by dividing the array in two parts. Binary search works on sorted array.

First it will check with middle element whether the test value is equal to middle element, if true then perform some task,

Else, check test value is less than middle element then search in left part (in case of ascending order) if not then search in right part.

With each part follow the same steps.



Algorithm:

```
binary (arr[ ], first, last, test_value)
{
mid= last-first/2;
if  test_value==mid;
        true;
else if test_value<mid
        binary(arr[], first, mid, tes_value);
else binary(arr[], mid+1, last, test_value);
}
```

Here first is starting index of array and last is last index of array.

NOTE! In case of even size of array the array is divided into two equal parts, e.g.) for array size 8 array will be divided into 4 and 4, but in case of odd size the array is divided into two unequal parts, e.g.) for size 9, the array is divided into 4 and 5.

Time complexity: **O(Logn).**

`Try implementing code of Binary search.`

SORTING

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

Following are some of the examples of sorting in real-life scenarios −

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

The various searching techniques are:

- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Heap Sort
- Quick Sort

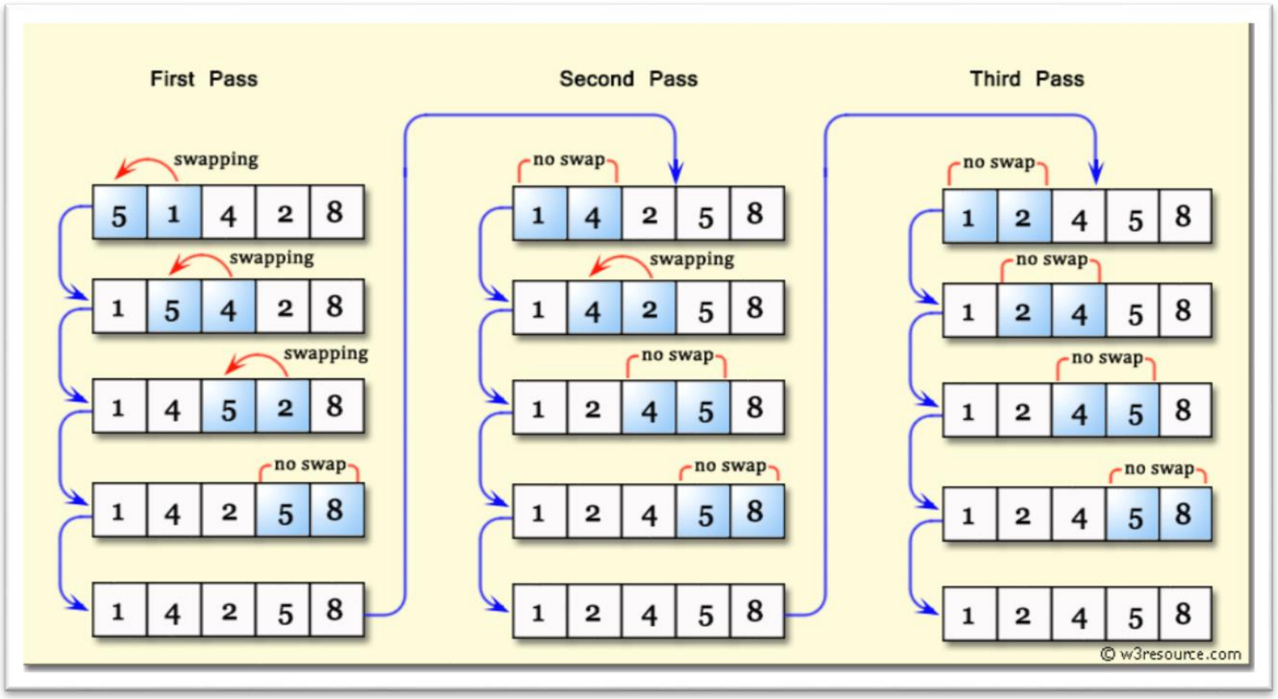
In sorting keep in mind:

- Ascending order: The order in which elements are in increasing order. e.g.) 0, 3, 4, 5, 6…
- Descending order: The order in which elements are in decreasing order. e.g.) 9, 8, 7, 6…
- Non Increasing order: The order in which the elements are not increasing as we traverse the array.
- Non decreasing order: The order in which the elements are not decreasing as we traverse the array.

BUBBLE SORT

Bubble sort is the easiest sort among all other techniques. In this sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

This algorithm is not suitable for large data sets as its average and worst case complexity are of **O(n^2)** where n is the number1 of items.



Algorithm:

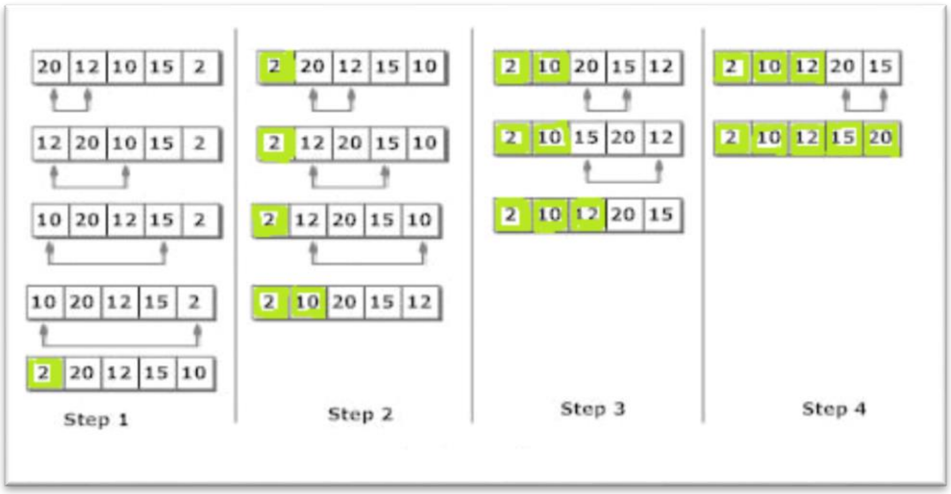
```
bubbleSort( arr[], n)
{
  for i = 0 to i < n-1
     for j = 0 to j < n-i-1
        if arr[j] > arr[j+1]
                 {
                 temp = arr[j];
                 arr[j] = arr[j+1];
                 arr[j+1]= temp;
                 }
}
```

SELECTION SORT

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- The subarray which is already sorted.
- Remaining subarray which is unsorted.

In every iteration of selection sort, the first element is compared with other element of array and the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.



The green part of array is sorted one and other is unsorted one.

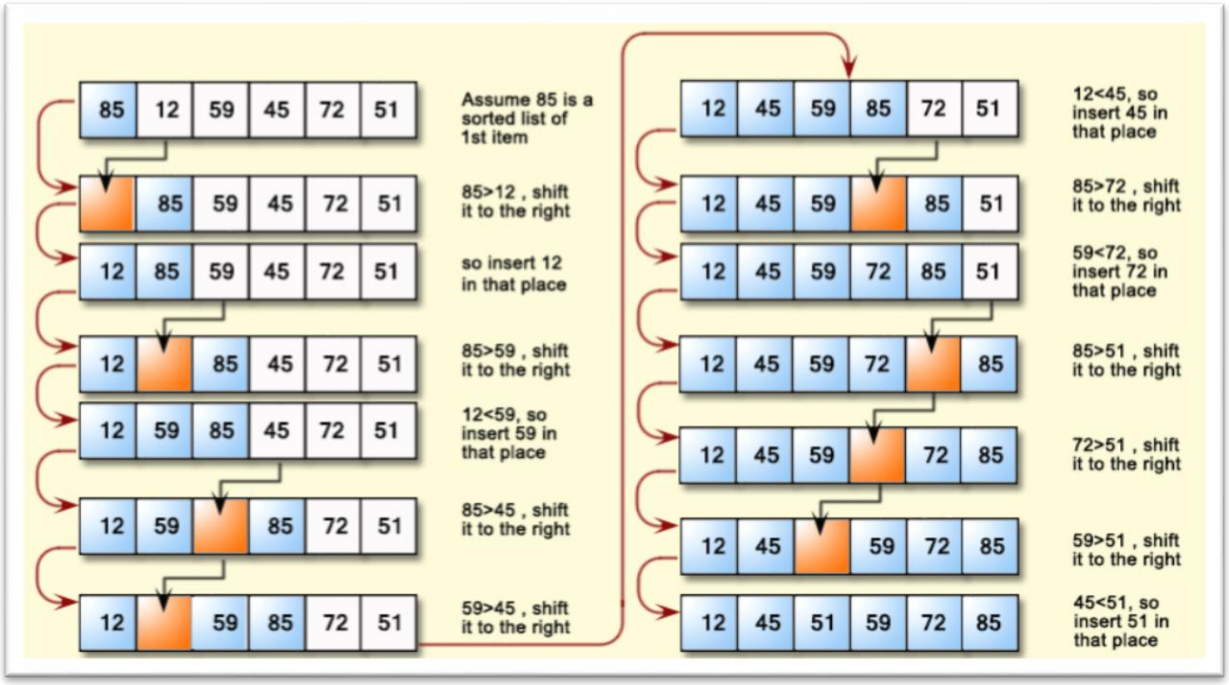
Algorithm:

```
list  : array of items
n     : size of list
for i = 1 to n – 1
            min = i       /* set current element as minimum*/
      /* check the element to be minimum */
      for j = i+1 to n
            if list[j] < list[min] then
            min = j;
            end if
      end for of j
/* swap the minimum element with the current element*/
if indexMin != i  then
      swap list[min] and list[i]
      end if
end for of i.
end procedure
```

Time Complexity: O(n^2) as there are two nested loops. The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

INSERTION SORT

- In Insertion Sort a sub-list is maintained which is always sorted. e.g.) the lower part of an array is maintained to be sorted.
- An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there.
- The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).
- This algorithm is not suitable for large data sets


85 12 59 45 72 51
Assume 85 is a sorted list of 1st item
85 59 45 72 51
85>12 , shift it to the right
12 85 59 45 72 51
so insert 12 in that place
12 85 45 72 51
85>59 , shift it to the right
12 59 85 45 72 51
12<59, so insert 59 in that place
12 59 85 72 51
85>45 , shift it to the right
12 59 85 72 51
59>45 , shift it to the right
12 45 59 85 72 51
12<45, so insert 45 in that place
12 45 59 85 51
85>72 , shift it to the right
12 45 59 72 85 51
59<72, so insert 72 in that place
12 45 59 72 85
85>51 , shift it to the right
12 45 59 72 85
72>51 , shift it to the right
12 45 59 72 85
59>51 , shift it to the right
12 45 51 59 72 85
45<51, so insert 51 in that place

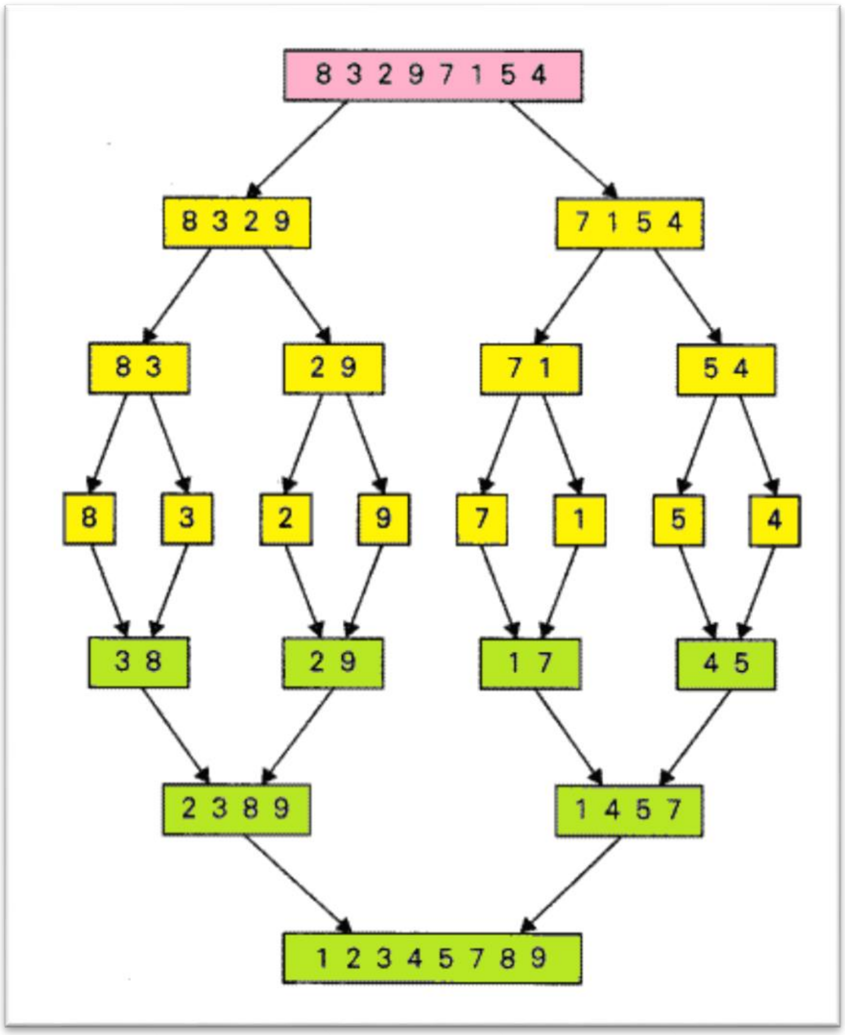
ALGORITHM:

```
void insertionSort( arr[], n)              //n is size of array
{
   i, key, j;
  for i = 1 to i < n
  {
     key = arr[i];
     j = i-1;     /* Move elements of arr[0..i-1], that are greater than key, to one position
ahead of their current position */
     while j >= 0 and  arr[j] > key
                arr[j+1] = arr[j];
                j = j-1;
     arr[j+1] = key;
  }
}
```

Time Complexity: O(n^2) because there are two loops.

MERGE SORT

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.
The merge () function is used for merging two halves.



Algorithm:

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
          middle m = (l+r)/2
     2. Call mergeSort for first half:
          Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
          Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3
          Call merge(arr, l, m, r)
```

```
merge (arr[], l,  m,  r)
{
    i, j, k;
    n1 = m - l + 1;                    // to find the size of first half matrix
    n2 =  r - m;                       /* to find the size of second half matrix
    L[n1], R[n2];                      /* create temp arrays */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];              /* Copy data to temp array L[]*/
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];           /* Copy data to temp array R[] */

     /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 and j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
 while (i < n1)
    {
        arr[k] = L[i];                 /* Copy the remaining elements of L[], if there are any */
        i++;
        k++;
    }
while (j < n2)
    {
        arr[k] = R[j];                 /* Copy the remaining elements of R[], if there are any */
        j++;
        k++;
    }
}
```

```
/* l is for left index and r is right index of the sub-array of arr to be sorted */
void mergeSort(arr[], l, r)
{
   if (l < r)
   {
     m = l+(r-l)/2;                       // Same as (l+r)/2, but avoids overflow for large l and h
     // Sort first and second halves
       mergeSort(arr, l, m);
       mergeSort(arr, m+1, r);
       merge(arr, l, m, r);
   }
}
```

Time Complexity: Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
$T(n) = 2T(n/2) + \Theta(n)$
The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\Theta(n \log n)$.
Time complexity of Merge Sort is **$\Theta(n \log n)$** in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

QUICK SORT

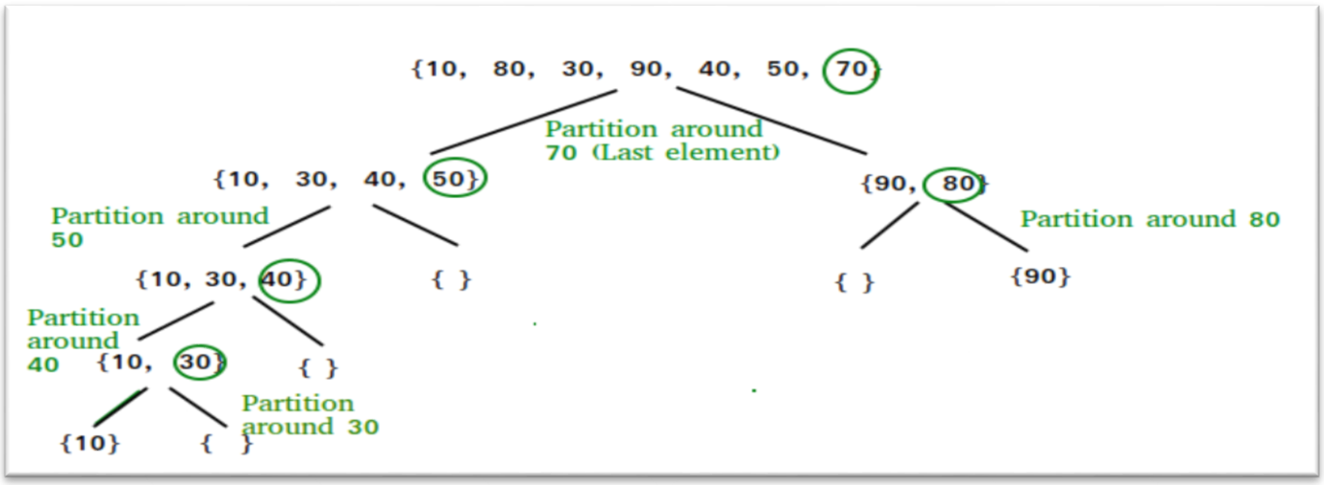
Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quick sort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

The key process in quick sort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[p] is now
         at right place */
      pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
```



Partition Algorithm:
There can be many ways to do partition. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[p] is now
         at right place */
      pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
```

Pseudo code for partition()
/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */

```
partition (arr[], low, high)
{
   // pivot (Element to be placed at right position)
   pivot = arr[high];

   i = (low - 1)  // Index of smaller element

   for (j = low; j <= high- 1; j++)
   {
      // If current element is smaller than or
      // equal to pivot
      if (arr[j] <= pivot)
      {
         i++;    // increment index of smaller element
         swap arr[i] and arr[j]
      }
   }
   swap arr[i + 1] and arr[high])
   return (i + 1)
}
```

GREEDY ALGORITHM

Greedy algorithms are used for optimization problems.
An optimization problem can be solved using Greedy if the problem has the following property:

- At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.

If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques.

For example: This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹ 1, 2, 5 and 10 and we are asked to count ₹ 18 then the greedy procedure will be −

- Select one ₹ 10 coin, the remaining count is 8
- Then select one ₹ 5 coin, the remaining count is 3
- Then select one ₹ 2 coin, the remaining count is 1
- finally, the selection of one ₹ 1 coins solves the problem

Following are some standard algorithms that are Greedy algorithms.

- Activity Selection Problem: You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

- Job Sequencing Problem: Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

- Huffman Coding: Huffman Coding is a loss-less compression technique. It assigns variable length bit codes to different characters. The Greedy Choice is to assign least bit length code to the most frequent character.

- Kruskal’s Minimum Spanning Tree: In Kruskal’s algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn’t cause a cycle in the MST constructed so far.

- Prim’s Minimum Spanning Tree: In Prim’s algorithm also, we create a MST by picking edges one by one. We maintain two sets: set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.

- Dijkstra’s Shortest Path Algorithm: The Dijkstra’s algorithm is very similar to Prim’s algorithm. The shortest path tree is built up, edge by edge. We maintain two sets: set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.

- Traveling Salesman Problem: Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

- Graph Coloring: The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color.

- Fractional Knapsack Problem: Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack. We can part of an item to fill the knapsack.

- Minimum number of coins required: Given a value V, if we want to make change for V Rs, and we have infinite supply of each of the denominations in any currency, what is the minimum number of coins and/or notes needed to make the change.

Fractional Knapsack

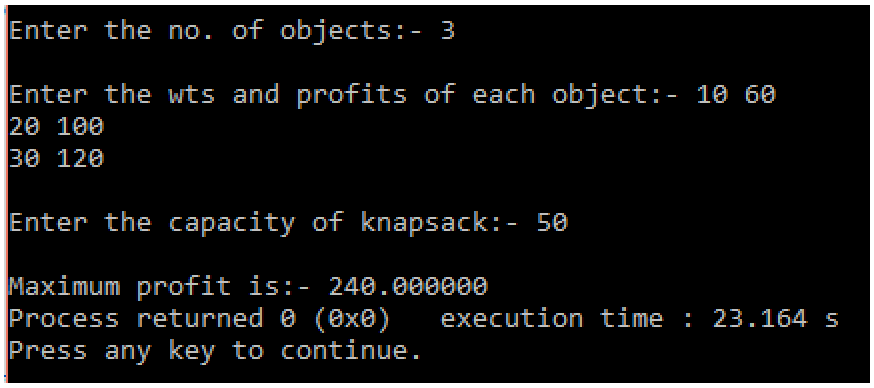
Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack.
In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don’t take it.
In Fractional Knapsack, we can break items for maximizing the total value of knapsack. This problem in which we can break item also called fractional knapsack problem.

```
Input :
  Items as (value, weight) pairs
 arr[] = {{60, 10}, {100, 20}, {120, 30}}
 Knapsack Capacity, W = 50;

Output :
  Maximum possible value = 240
  By taking full items of 10 kg, 20 kg and
  2/3rd of last item of 30 kg
```

The basic idea of greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with highest ratio and add them until we can’t add the next item as whole and at the end add the next item as much as we can. Which will always be optimal solution of this problem.

Code of Fractional Knapsack:

```
# include<stdio.h>
void knapsack(int n, float weight[], float profit[], float capacity)
{
  float x[20], tp = 0;
  int i, j, u;
  u = capacity;
  for (i = 0; i < n; i++)
    x[i] = 0.0;
  for (i = 0; i < n; i++)
   {
    if (weight[i] > u)
      break;
    else
    {
      x[i] = 1.0;
      tp = tp + profit[i];
      u = u - weight[i];
    }
  }
  if (i < n)
    x[i] = u / weight[i];
  tp = tp + (x[i] * profit[i]);
printf("\nMaximum profit is:- %f", tp);
}
int main()
{
  float weight[20], profit[20], capacity;
  int num, i, j;
  float ratio[20], temp;
  printf("\nEnter the no. of objects:- ");
  scanf("%d", &num);
  printf("\nEnter the wts and profits of each object:- ");
  for (i = 0; i < num; i++) {
    scanf("%f %f", &weight[i], &profit[i]);
  }
  printf("\nEnter the capacity of knapsack:- ");
  scanf("%f", &capacity);
  for (i = 0; i < num; i++)
   {
    ratio[i] = profit[i] / weight[i];
  }
  for (i = 0; i < num; i++)
  {
    for (j = i + 1; j < num; j++)
    {
      if (ratio[i] < ratio[j])
      {
        temp = ratio[j];
        ratio[j] = ratio[i];
        ratio[i] = temp;

        temp = weight[j];
        weight[j] = weight[i];
        weight[i] = temp;

        temp = profit[j];
        profit[j] = profit[i];
        profit[i] = temp;
      }
    }
  }
  knapsack(num, weight, profit, capacity);
  return(0);
}
```



```
Enter the no. of objects:- 3

Enter the wts and profits of each object:- 10 60
20 100
30 120

Enter the capacity of knapsack:- 50

Maximum profit is:- 240.000000
Process returned 0 (0x0)   execution time : 23.164 s
Press any key to continue.
```

In the code, value/weight is calculated, and then according to maximum profit items are added.

Try implementing using different sorting algorithm.

Minimum Spanning Tree???

- Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together.
- A single graph can have many different spanning trees.
- A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree.
- The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.
- A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Various application of MST:

- Network design– telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems– traveling salesperson problem, Steiner tree
  max bottleneck paths
- LDPC codes for error correction
- Image registration with Renyi entropy
- Learning salient features for real-time face verification
- Reducing data storage in sequencing amino acids in a protein
- Model locality of particle interactions in turbulent fluid flows
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network

Kruskal's Minimum Spanning Tree Algorithm

The Greedy Choice is to pick the smallest weight edge that does not because a cycle in the MST constructed so far.

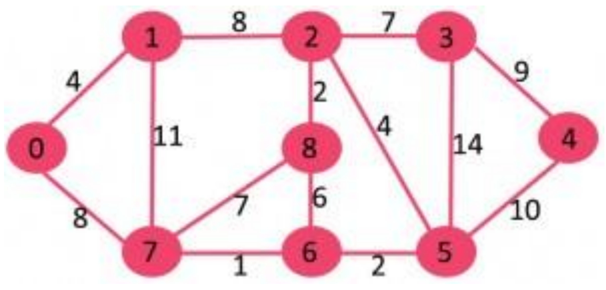
Algorithm:

- Sort all the edges in non-decreasing order of their weight.

- Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

- Repeat step#2 until there are (V-1) edges in the spanning tree.

Let us understand it with an example, Consider the below input graph.



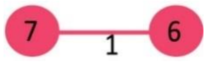
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

After sorting:

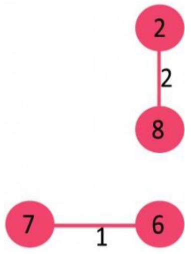
Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from sorted list of edges

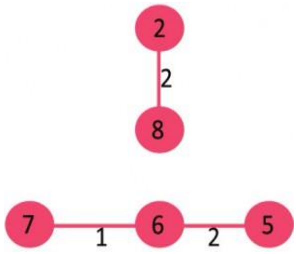
1. Pick edge 7-6*:* No cycle is formed, include it.



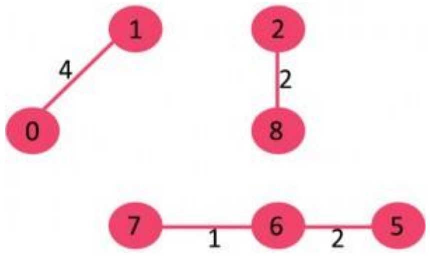
2. Pick edge 8-2: No cycle is formed, include it.



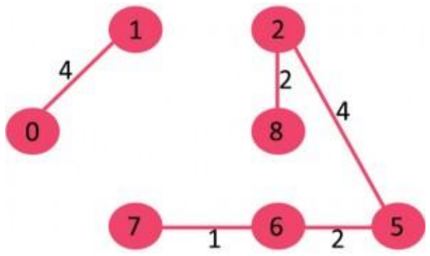
3. Pick edge 6-5*:* No cycle is formed, include it.



4. Pick edge 0-1*:* No cycle is formed, include it.

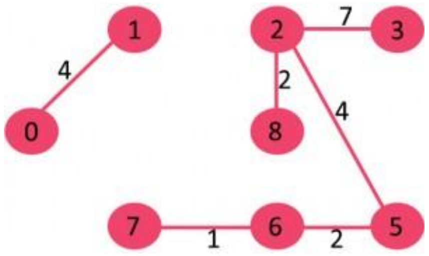


5. Pick edge 2-5*:* No cycle is formed, include it.



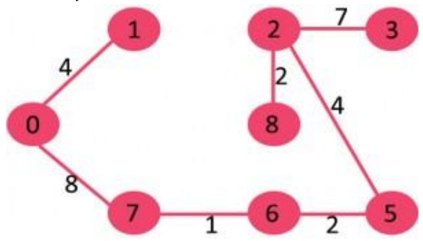
6. Pick edge 8-6: Since including this edge results in cycle, discard it.

7. Pick edge 2-3*:* No cycle is formed, include it.



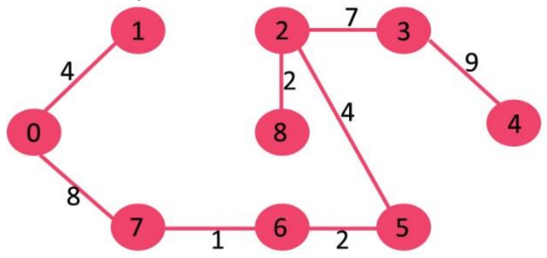
8. Pick edge 7-8: Since including this edge results in cycle, discard it.

9. Pick edge 0-7*:* No cycle is formed, include it.



10. Pick edge 1-2: Since including this edge results in cycle, discard it.

11. Pick edge 3-4: No cycle is formed, include it.



Since the number of edges included equals (V – 1), the algorithm stops here.

COMPLEXITY: O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take at most O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be at most O(V^2), so O(LogV) are O(LogE) same. Therefore, overall time complexity is O(ElogE) or O(ElogV)

WATCH HERE

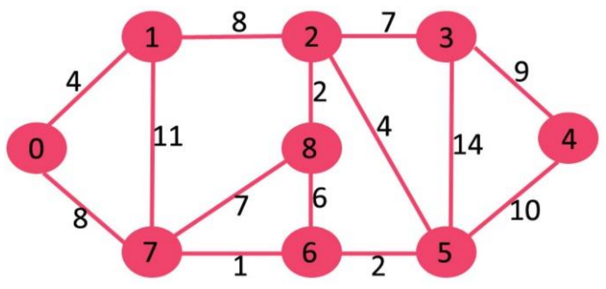
Prim’s Minimum Spanning Tree (MST)

- Like Kruskal’s algorithm, Prim’s algorithm is also a Greedy algorithm.
- It starts with an empty spanning tree. The idea is to maintain two sets of vertices.
- The first set contains the vertices already included in the MST, the other set contains the vertices not yet included.
- At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges.
- After picking the edge, it moves the other endpoint of the edge to the set containing MST.
- The idea behind Prim’s algorithm is the two disjoint subsets of vertices must be connected to make a Spanning Tree and they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

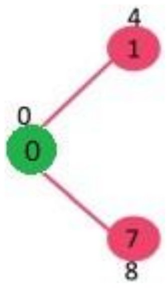
Algorithm

- Create a set mstSet that keeps track of vertices already included in MST.
- Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
- While mstSet doesn’t include all vertices
    - Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
    - Include *u* to mstSet.
    - Update key value of all adjacent vertices of *u*.
- To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u*-*v* is less than the previous key value of *v*, update the key value as weight of *u*-*v*

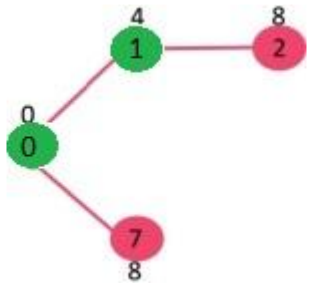
Let us understand with the following example:



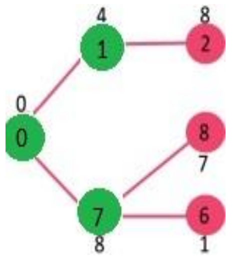
- The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite.
- Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*.
- So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices.
- Adjacent vertices of 0 are 1 and 7.
- The key values of 1 and 7 are updated as 4 and 8.
- Following subgraph shows vertices and their key values, only the vertices with finite key values are shown.
- The vertices included in MST are shown in green color.



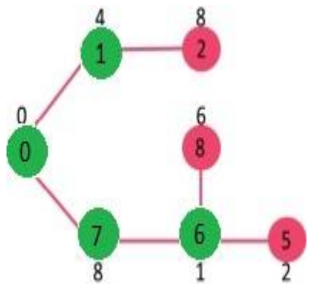
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



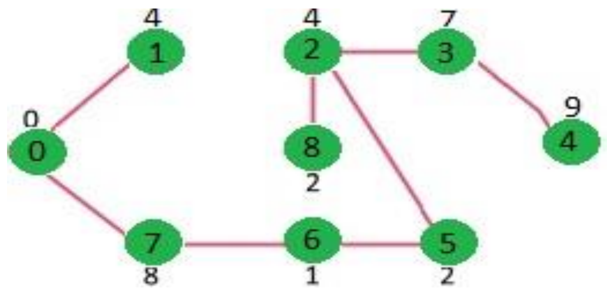
Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).



Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



COMPLEXITY: Time Complexity is O(V^2). If the input graph is represented using adjacency list, then the time complexity of Prim's algorithm can be reduced to O(E log V) with the help of binary heap

WATCH HERE

DIJKSTRA’S ALGORITHM

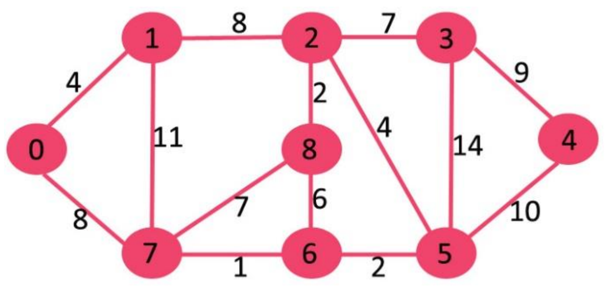
The main objective of Dijkstra’s algorithm is for a given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

- Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a *SPT (shortest path tree)* with given source as root.
- We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree.
- At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.
- Dijkstra doesn’t work on negative weights in graph, for negative weights Bellmen-ford works.

Algorithm:

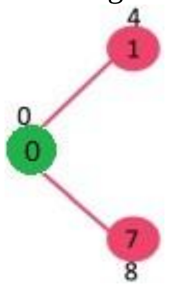
- Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE.
- Assign distance value as 0 for the source vertex so that it is picked first.
- While *sptSet* doesn’t include all vertices
    - Pick a vertex u which is not there in *sptSet*and has minimum distance value.
    - Include u to *sptSet*.
    - Update distance value of all adjacent vertices of u.
- To update the distance values, iterate through all adjacent vertices.
- For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Let us understand with the following example:

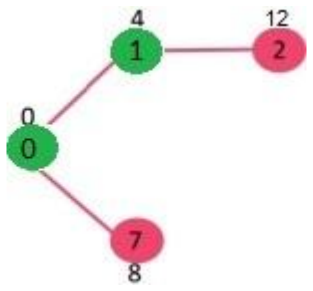


- The set *sptSet*is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite.
- Now pick the vertex with minimum distance value.
- The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}.
- After including 0 to *sptSet*, update distance values of its adjacent vertices.
- Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.

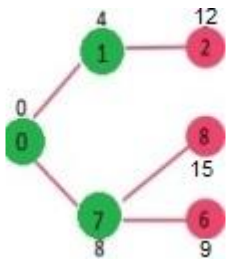
- Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown.
- The vertices included in SPT are shown in green color.



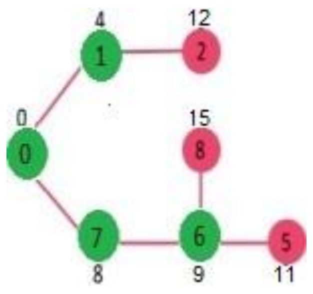
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



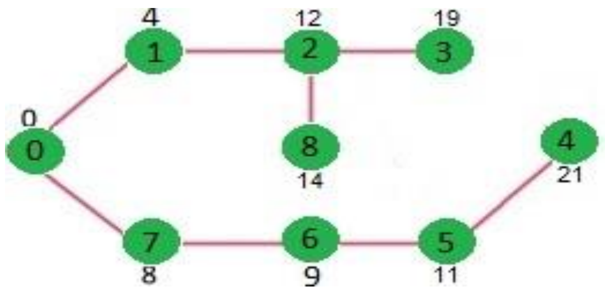
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* doesn’t include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



COMPLEXITY: Time Complexity of the implementation is O(V^2). If the input graph is represented using adjacency list, it can be reduced to O(E log V) with the help of binary heap.

WATCH HERE

HUFFMAN CODE

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-legth codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.
The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

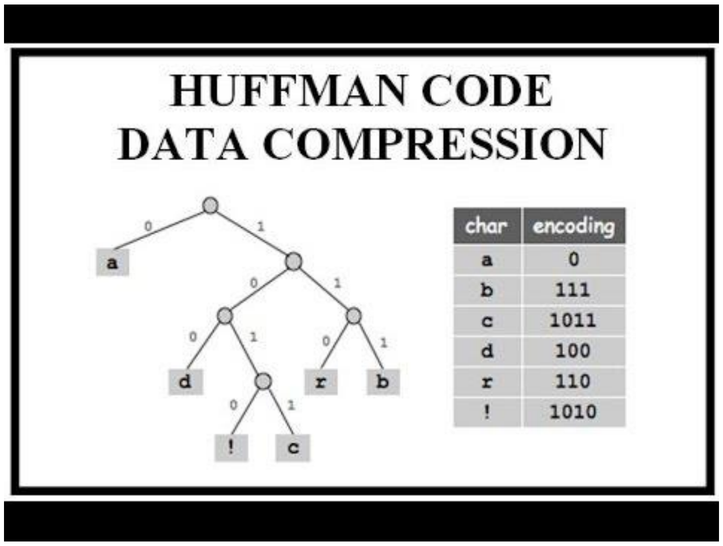
There are mainly two major parts in Huffman Coding

- Build a Huffman Tree from input characters.
- Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree

- Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.
- Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root).
- Extract two nodes with the minimum frequency from the min heap.
- Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
- Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Watch the video for understanding.



char	encoding
a	0
b	111
c	1011
d	100
r	110
!	1010

DYNAMIC PROGRAMMING

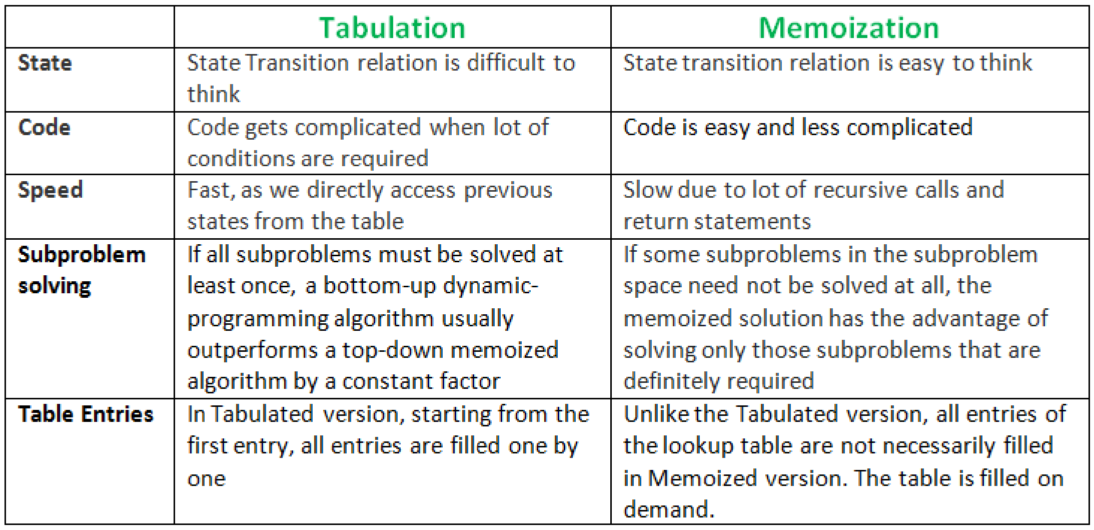
- Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.
- The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a modest expenditure in storage space.
- Dynamic programming algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem. In comparison, a greedy algorithm treats the solution as some sequence of steps and picks the locally optimal choice at each step. Using a greedy algorithm does not guarantee an optimal solution, because picking locally optimal choices may result in a bad global solution, but it is often faster to calculate.

If the coin denominations are 1,4,5,15,20 and the given amount is 23, this greedy algorithm gives a non-optimal solution of 20+1+1+1, while the optimal solution is 15+4+4.

NOTE! In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems. If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

There are following two different ways to store the values so that the values of a problem can be reused. Here, will discuss two patterns of solving DP problem:

- **Tabulation:** Bottom Up
- **Memoization:** Top Down



	Tabulation	Memoization
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

Some common problems of Dynamic programming are:

- Longest Common Subsequence: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”,etc are subsequences of “abcdefg.

- Largest Sum Contiguous Subarray: to find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum

- Knapsack Problem: Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don’t pick it

- Length of the longest substring without repeating characters: Given a string, find the length of the longest substring without repeating characters. For example, the longest substrings without repeating characters for “ABDEFGABEF” are “BDEFGA” and “DEFGAB”, with length 6.

- Floyd Warshall Algorithm: The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

- Bellman–Ford Algorithm: Given a graph and a source vertex src in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.
  Dijkstra doesn’t work for Graphs with negative weight edges, Bellman-Ford works for such graphs.

- Optimal Binary Search Tree: Given a sorted array *keys[0.. n-1]* of search keys and an array *freq[0.. n-1]* of frequency counts, where *freq[i]* is the number of searches to *keys[i]*. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

LONGEST COMMON SUBSEQUENCE

The main objective of this problem is finding the length of longest subsequence present in given two sequences. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.
For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, etc are subsequences of “abcdefg”. So a string of length n has 2^n different possible subsequences.

Examples:
LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.
LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4

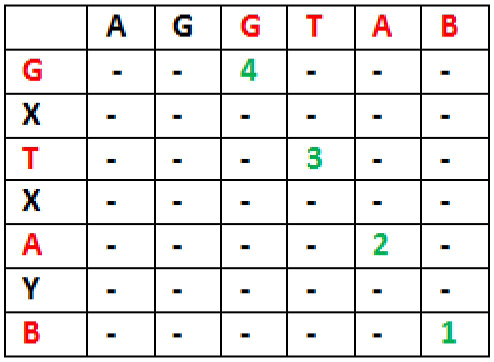
The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity.
Using DP:

- Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y. Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).

- If last characters of both sequences match (or X[m-1] == Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

- If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2])

Example:
Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:
L(“AGGTAB”, “GXTXAYB”) = 1 + L(“AGGTA”, “GXTXAY”)

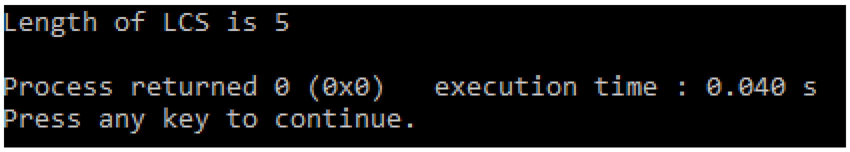


	A	G	G	T	A	B
G	-	-	4	-	-	-
X	-	-	-	-	-	-
T	-	-	-	3	-	-
X	-	-	-	-	-	-
A	-	-	-	-	2	-
Y	-	-	-	-	-	-
B	-	-	-	-	-	1

So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

CODE:

```
#include<stdio.h>
int max(int a, int b);
int lcs( char *X, char *Y, int m, int n )
{
   if (m == 0 || n == 0)
     return 0;
   if (X[m-1] == Y[n-1])
     return 1 + lcs(X, Y, m-1, n-1);
   else
     return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}
int max(int a, int b)
{     return (a > b)? a : b;
}
int main()
{
  char X[] = "ABCDEFGHIJ";
  char Y[] = "BKLDEOYFG";
   int m = strlen(X);
  int n = strlen(Y);
  printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );
  return 0;
}
```



```
Length of LCS is 5

Process returned 0 (0x0)   execution time : 0.040 s
Press any key to continue.
```

MAXIMUM SUM OF SUBARRAY

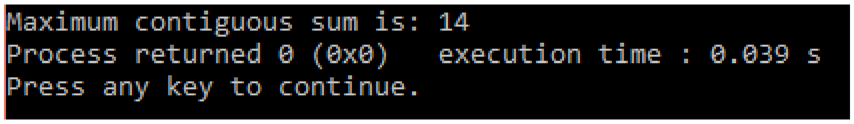
To find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum.

```
Initialize:
    max_so_far = 0
    max_ending_here = 0

Loop for each element of the array
  (a) max_ending_here = max_ending_here + a[i]
  (b) if(max_ending_here < 0)
            max_ending_here = 0
  (c) if(max_so_far < max_ending_here)
            max_so_far = max_ending_here
return max_so_far
```

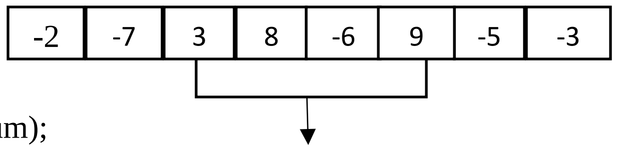
Simple idea is to look for all positive contiguous segments of the array (max_ending_here is used for this). And keep track of maximum sum contiguous segment among all positive segments (max_so_far is used for this). Each time we get a positive sum compare it with max_so_far and update max_so_far if it is greater than max_so_far.

CODE:

```c
#include<stdio.h>
int maxSubArraySum(int a[], int size)
{
    int i, max_so_far=0, INT_MIN=0, max_ending_here = 0;
    for ( i = 0; i < size; i++)
    {
        max_ending_here = max_ending_here + a[i];
        if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;
        if (max_ending_here < 0)
            max_ending_here = 0;
    }
    return max_so_far;
}
int main()
{
    int a[] = {-2, -7, 3, 8, -6, 9, -5, -3};
    int n = sizeof(a)/sizeof(a[0]);
    int max_sum = maxSubArraySum(a, n);
    printf("Maximum contiguous sum is: %d", max_sum);
    return 0;
}
```



```
Maximum contiguous sum is: 14
Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.
```



Maximum sum of contingent sub-array

0-1 KNAPSACK

Given weights and values of n items, put these items in a knapsack of capacity and the condition is one cannot break an item, either pick the complete item, or don’t pick it (0-1 property).
A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

Using Dynamic Property:
To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.
Therefore, the maximum value that can be obtained from n items is max of following two values.

- Maximum value obtained by n-1 items and W weight (excluding nth item).
- Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).
- If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

In the following recursion tree, K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W.
The recursion tree is for following sample inputs.
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

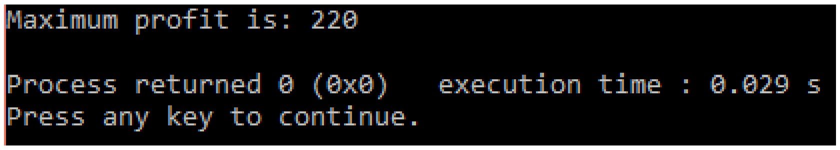
```
                        K(3, 2)          ---------> K(n, W)
                     /             \
                   /                 \
             K(2,2)                   K(2,1)
           /        \                   /    \
         /            \               /        \
       K(1,2)       K(1,1)         K(1,1)      K(1,0)
        /  \          /   \           /   \
      /      \      /       \       /       \
K(0,2)   K(0,1)  K(0,1)   K(0,0)  K(0,1)    K(0,0)
```

Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight

Code:

```
#include<stdio.h>
int max(int a, int b) { return (a > b)? a : b; }
int knapSack(int W, int wt[], int val[], int n)
{
  int i, w;
  int K[n+1][W+1];
  for (i = 0; i <= n; i++)
  {
     for (w = 0; w <= W; w++)
```

```
        {
            if (i==0 || w==0)
                K[i][w] = 0;
            else if (wt[i-1] <= w)
                  K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);
            else
                  K[i][w] = K[i-1][w];
        }
    }
   return K[n][W];
}
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int  W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("Maximum profit is: %d\n", knapSack(W, wt, val, n));
    return 0;
}
```



```
Maximum profit is: 220

Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.
```

Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.

Floyd Warshall Algorithm

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.
Example:
Input:

```
     graph[][] = { {0,   5,  INF, 10},
                   {INF,  0,  3,  INF},
                   {INF, INF, 0,   1},
                   {INF, INF, INF, 0} }
```

which represents the following graph

```
           10
       (0)------->(3)
        |         /|\
       5|          |
        |          | 1
       \|/         |
       (1)------->(2)
            3
```

Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

Output:
Shortest distance matrix

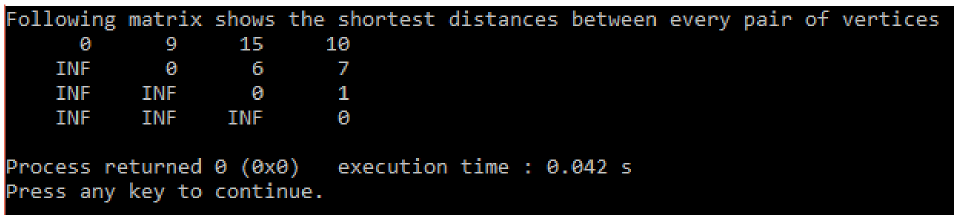
```
   0          5       8       9
  INF         0       3       4
  INF         INF     0       1
  INF         INF     INF     0
```

Algorithm:

- We initialize the solution matrix same as the input graph matrix as a first step.
- Then we update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2 ... k-1} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.
    - k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
    - k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].

Code:

```
#include<stdio.h>
#define V 4
#define INF 99999
void printSolution(int dist[][V]);
void floydWarshall (int graph[][V])
{
    int dist[V][V], i, j, k;
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            for (j = 0; j < V; j++)
            {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
    printSolution(dist);
}
void printSolution(int dist[][V])
{
    int i ,j;
    printf ("Following matrix shows the shortest distances"
            " between every pair of vertices \n");
    for ( i = 0; i < V; i++)
    {
        for ( j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf ("%7d", dist[i][j]);
        }
        printf("\n");
    }
}
int main()
{
    int graph[V][V] = { {0,   9,  INF, 10},
                        {INF, 0,   6, INF},
```



```
Following matrix shows the shortest distances between every pair of vertices
      0      9     15     10
    INF      0      6      7
    INF    INF      0      1
    INF    INF    INF      0

Process returned 0 (0x0)   execution time : 0.042 s
Press any key to continue.
```

```
                    {INF, INF, 0,   1},
                    {INF, INF, INF, 0}
                  };
    floydWarshall(graph);
    return 0;}
```

Time Complexity: O(V^3) The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.

BELLMEN FORD ALGORITHM

- The main objective of Bellmen Ford is for a given graph and a source vertex src in graph, finding the shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.
- Dijksra's algorithm is a Greedy algorithm and time complexity is O(VLogV*)*. Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs.
- Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems.

Algorithm:
Input: Graph and a source vertex *src*
Output: Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

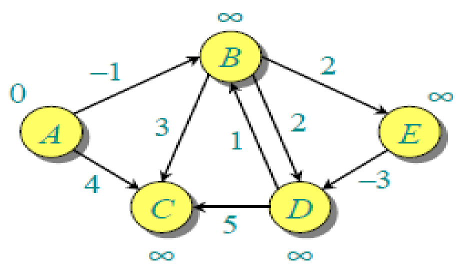
- This step initializes distances from source to all vertices as infinite and distance to source itself as 0.
- Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
- This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
- Do following for each edge u-v
  - If dist[v] > dist[u] + weight of edge uv, then update dist[v]
  - dist[v] = dist[u] + weight of edge uv
- This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
  - If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"

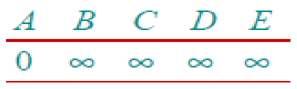
The idea of step 4 is, step 3 guarantees shortest distances if graph doesn't contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

How does this work?
Like other Dynamic Programming Problems, the algorithm calculate shortest paths in bottom-up manner.

- It first calculates the shortest distances for the shortest paths which have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on.
- After the ith iteration of outer loop, the shortest paths with at most i edges are calculated.
- There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times.
- The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges.

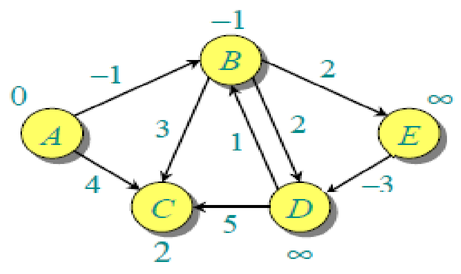
Let's understand the algorithm with following example graph.
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*

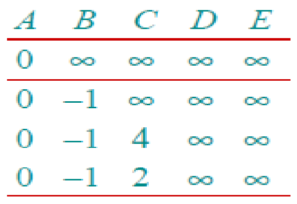




A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞

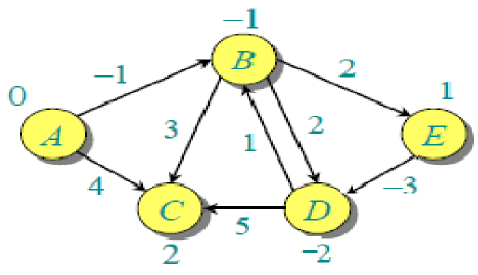
Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.

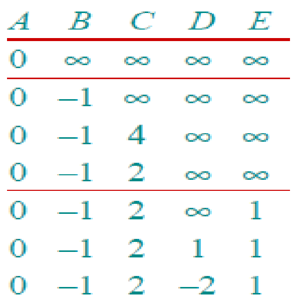




A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞

The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).





A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

Time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

CLICK HERE to watch

BACKTRACKING

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that “works”.
Problems which are typically solved using backtracking technique have following property in common.

- These problems can only be solved by trying every possible configuration and each configuration is tried only once.

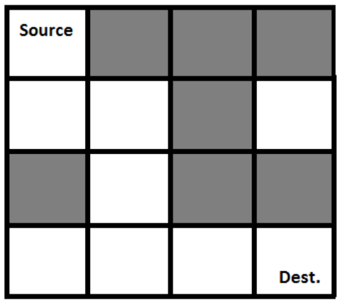
A Naive solution for these problems is to try all configurations and output a configuration that follows given problem constraints.
Backtracking works in incremental way and is an optimization over the Naive solution where all possible configurations are generated and tried.

The problems on backtracking are:

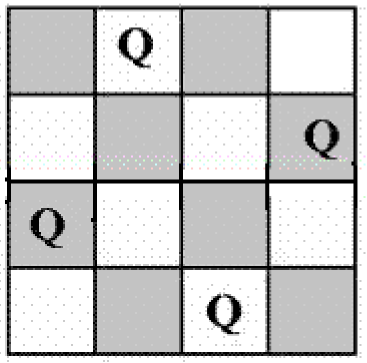
- **Print all permutations of a given string**: All the possible combination of characters of the strings.

- **The Knight’s tour problem**: The knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once.

- **Rat in a Maze**: A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach destination. The rat can move only in two directions: forward and down.
  In the maze matrix, 0 means the block is dead end and 1 means the block can be used in the path from source to destination.
  Gray blocks are dead ends (value = 0).

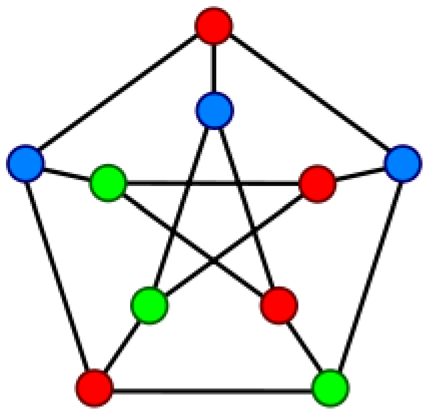


- **N Queen Problem**: The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen Problem.



- **Subset Sum**: Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

- **M-Coloring Problem**: Given an undirected graph and a number M, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.



- **Sudoku**: Given a partially filled 9×9 2D array 'grid[9][9]', the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.

- **Tug of War**: Given a set of n integers, divide the set in two subsets of n/2 sizes each such that the difference of the sum of two subsets is as minimum as possible. If n is even, then sizes of two subsets must be strictly n/2 and if n is odd, then size of one subset must be (n-1)/2 and size of other subset must be (n+1)/2.
For example, let given set be {3, 4, 5, -3, 100, 1, 89, 54, 23, 20}, the size of set is 10. Output for this set should be {4, 100, 1, 23, 20} and {3, 5, -3, 89, 54}. Both output subsets are of size 5 and sum of elements in both subsets is same (148 and 148).

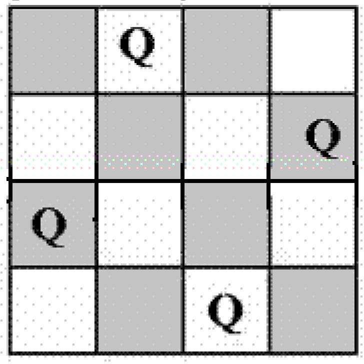
Another example where n is odd. Let given set be {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4}. The output subsets should be {45, -34, 12, 98, -1} and {23, 0, -99, 4, 189, 4}. The sums of elements in two subsets are 120 and 121 respectively.

- **Solving Cryptarithmetic Puzzles**: The goal here is to assign each letter a digit from 0 to 9 so that the arithmetic works out correctly. The rules are that all occurrences of a letter must be assigned the same digit, and no digit can be assigned to more than one letter. E.g.)

```
   SEND
 + MORE
 --------
  MONEY
 --------
```

N-QUEEN PROBLEM

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen Problem.



Naive Algorithm:
Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

```
while there are untried conflagrations
{
   generate the next configuration
   if queens don't attack in this configuration then
   {
      print this configuration;
   }
}
```

Backtracking Algorithm:
The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

- Start in the leftmost column
- If all queens are placed
  return true
- Try all rows in the current column. Do following for every tried row.
  a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
  b) If placing queen in [row, column] leads to a solution then return
  true.
  c) If placing queen doesn't lead to a solution then umark this [row, column] (Backtrack) and go to step (a) to try other rows.
- If all rows have been tried and nothing worked, return false to trigger
  backtracking.

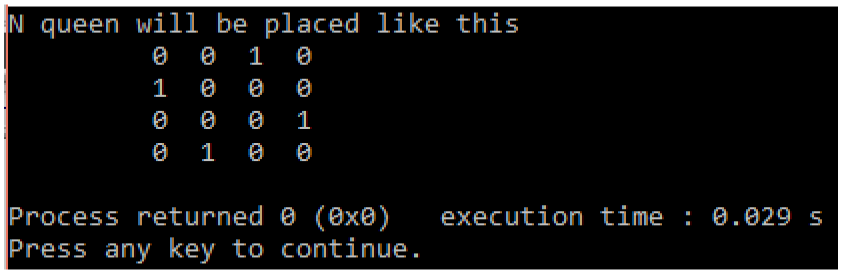
CODE:

```
#define N 4
#include<stdio.h>
void printSolution(int board[N][N])
{
int i,j;
printf("N queen will be placed like this\n");
    for ( i = 0; i < N; i++)
    {
        printf("\t");
            for ( j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}

/* A utility function to check if a queen can be placed on board[row][col]. Note that this function
is called when "col" queens are already placed in columns from 0 to col -1. So we need to check
only left side for attacking queens */
int isSafe(int board[N][N], int row, int col)
{
    int i, j;
for (i = 0; i < col; i++)            /* Check this row on left side */
        if (board[row][i])
            return 0;

 for (i=row, j=col; i>=0 && j>=0; i--, j--)      /* Check upper diagonal on left side */
        if (board[i][j])
            return 0;

 for (i=row, j=col; j>=0 && i<N; i++, j--)     /* Check lower diagonal on left side */
        if (board[i][j])
            return 0;

    return 3;
}

/* A recursive utility function to solve N Queen problem */
int solveNQUtil(int board[N][N], int col)
{
    int i;
    if (col >= N)               /* base case: If all queens are placed then return true */
        return 3;

 for (i = 0; i < N; i++) /* Consider this column and try placing this queen in all rows one by one */
```

```
    {
        if ( isSafe(board, i, col) )            /* Check if queen can be placed on board[i][col] */
        {
            board[i][col] = 1;        /* Place this queen in board[i][col] */

           if ( solveNQUtil(board, col + 1) )       /* recur to place rest of the queens */
                return 3;

            /* If placing queen in board[i][col] doesn't lead to a solution, then remove queen from
board[i][col] */
            board[i][col] = 0; // BACKTRACK
        }
    }

     /* If queen can not be place in any row in this colum col  then return false */
    return 0;
}
```

/* This function solves the N Queen problem using Backtracking. It mainly uses solveNQUtil() to solve the problem. It returns false if queens cannot be placed, otherwise return true and prints placement of queens in the form of 1s. Please note that there may be more than one solutions, this function prints one of the feasible solutions.*/

```
int solveNQ()
{
    int board[N][N] = { {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0}
    };
    if ( solveNQUtil(board, 0) == 0 )
    {
      printf("Solution does not exist");
      return 0;
    }
    printSolution(board);
    return 3;
}

// driver program to test above function
int main()
{
    solveNQ();
    return 0;
}
```



```
N queen will be placed like this
         0  0  1  0
         1  0  0  0
         0  0  0  1
         0  1  0  0

Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.
```

In this code change the value of N then the solution will be for the N*N matrix.

REFERENCES:

1. https://www.tutorialspoint.com/
2. http://www.geeksforgeeks.org/
3. http://cslibrary.stanford.edu/
4. https://www.hackerearth.com/