BHARATI VIDYAPEETH DEEMED UNIVERSITY
COLLEGE OF ENGINEERING, PUNE - 43

DEPARTMENT OF COMPUTER ENGINEERING

Lab Manual

Algorithm Design and Analysis

B.Tech Computer Sem V(2020 Course)

VISION OF THE INSTITUTE

“To be World Class Institute for Social Transformation through Dynamic Education”

MISSION OF THE INSTITUTE

- To provide quality technical education with advanced equipment, qualified faculty members, infrastructure to meet needs of profession and society.
- To provide an environment conductive to innovation, creativity, research and entrepreneurial leadership.
- To practice and promote professional ethics, transparency and accountability for social community, economic and environmental conditions.

VISION OF THE DEPARTMENT

“To pursue and excel in the endeavor for creating globally recognized computer engineers through quality education.”

MISSION OF THE DEPARTMENT

- To impart engineering knowledge and skills confirming to a dynamic curriculum.
- To develop professional, entrepreneurial & research competencies encompassing continuous intellectual growth.
- To produce qualified graduates exhibiting societal and ethical responsibilities in working environment

PROGRAM EDUCATIONAL OBJECTIVES (PEOs):

1. Demonstrate technical and professional competencies by applying engineering fundamentals, computing principles and technologies.
2. Learn, Practice, and grow as skilled professionals/ entrepreneur/researchers adapting to the evolving computing landscape.
3. Demonstrate professional attitude, ethics, understanding of social context and interpersonal skills

leading to a successful career.

PROGRAM SPECIFIC OUTCOMES (PSO)s:

PSO 1: To design, develop and implement computer programs on hardware towards solving problems.
PSO 2: To employ expertise and ethical practice through continuing intellectual growth and adapting to the working environment.

PROGRAMME OUTCOMES (POs):

Upon completion of the course the graduate engineers will be able to:
1. Apply the knowledge of mathematics, science, engineering fundamentals, and computing for the solution of complex engineering problems.
2. Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using computer engineering foundations, principles, and technologies.
3. Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4. Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues, and the consequent responsibilities relevant to the professional engineering practice.
7. Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and the need for sustainable development.
8. Apply ethical principles while committed to professional responsibilities and norms of the engineering practice.
9. Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings
10. Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. Apply the engineering and management principles to one’s work, as a member and leader in a team.
12. Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

LABORATORY USAGE

Students use turbo C/C++ compiler for executing the lab experiments, document the results andto prepare the technical documents for making the lab reports.

OBJECTIVES:-

To implement design and analysis of algorithms.

PRACTICAL PRE-REQUISITE:-

1. Data structures.
2. Knowledge of C/C++

HARDWARE/ SOFTWARE REQUIREMENTS:-

1. C/C++ Compiler
2. VS Code

COURSE OUTCOMES

1) Analyze time complexity
2) Analyze space complexity
3) Discuss Divide and Conquer Method
4) Design algorithms using greedy Methods
5) Infer Backtracking
6) Outline NP-Hard and NP-Complete Problems

HOW OUTCOMES ARE ASSESSED?

Course Outcome	**Assignment Number**	**Level**	**Proficiency evaluated by**
Design and Analyze time complexity	1,2,3,4,5,7,9	3,3,3,3,3,3,3	Performing Practical and reporting results
Design and Analyze space complexity	1,3	3	Problem definition &Performing Practical and reporting results
Discuss Divide and Conquer Method.	2	2	Performing experiments and reporting results
Design algorithms using greedy Methods	8,9,10	3,3,3	Performing experiments and reporting results
Infer Backtracking	9	3,3	Performing experiments and reporting results
Outline NP-Hard and NP-Complete Problems	10	3,3	Performing experiments and reporting results

CO-PO mapping

CO Statements	**PO 1**	**PO 2**	**PO 3**	**PO 4**	**PO 5**	**PO 6**	**PO 7**	**PO 8**	**PO 9**	**PO 10**	**PO 11**	**PO 12**	**PSO 1**	**PSO 2**
CO1	3		3					2		3		2	3	
CO2	2	3	3		3			2		3		2	3	
CO3	2		3		3			2		3		2		3
CO4	2	3	3	3	3			2		3		2	2	3
CO5	3	3	3	3	3			2		3		2	1	
CO6	1	3		1	1			2		3		2		2

Guidelines for Student's Lab Journal

- The laboratory assignments are to be submitted by student in the form of journal. The Journal consists of prologue, Certificate, table of contents, and handwritten write-up of each assignment (Title, Objectives, Problem Statement, Date of Completion, Assessment grade/marks and assessor's sign, Theory- Concept, algorithms, printouts of the code written using coding standards, sample test cases etc.
- Practical Examination will be based on the term work submitted by the student in the form of journal
- Candidate is expected to know the theory involved in the experiment
- The practical examination should be conducted if the journal of the candidate is completed in all respects and certified by concerned faculty and head of the department
- All the assignment mentioned in the syllabus must be conducted

LIST OF ASSIGNMENTS

1. Introduction to Algorithms: Design & implement an algorithm to calculate the time and space complexity of bubble sort algorithm.
2. Study of Divide and Conquer: Write a program to implement Quick Sort and finding out minimum and maximum using divide and conquer.
3. Implement basic algorithms and perform analysis of complexities(stack, queue, tree,graph)
4. Greedy method: Design, Implement and analyse Prim's and Kruskal's algorithm for minimum cost spanning Tree.
5. Implement Job Sequencing Problem and find its complexity
6. Dynamic programming I: Design, Implement and analyse shortest path algorithm.
7. Dynamic Programming II: Design, Implement and analyse Optimal Binary Search Trees.
8. Backtracking I:Write a program to implement 8 queens problem
9. Backtracking II: Implement Hamiltonian Cycle.
10. Study NP Hard Graph , NP Hard Scheduling problem.

EXPERIMENT NO: 1

Aim: Introduction to Algorithms: Design & implement an algorithm to calculate the time and space complexity of bubble sort algorithm.

Objective: To study the concept of time and space complexity.

Theory:

The word algorithm comes from the name of a Persian author, Abu Ja'far Mohammed ibn musa al Khwarizmi(c825 A.D.),who wrote a text book on mathematics. This word has taken on a special significance in computer science, where "algorithm" has come to refer to a method that can be used by a computer for the solution of a problem. This is what makes algorithm different from words such as process, technique or method.

An algorithm is a finite set of instructions that if followed, accomplishes a particular task. In addition all algorithms must satisfy the following criteria:

1. Input: zero or more quantities are externally supplied.
2. Output: At least one quantity is produced.
3. Definiteness: each instruction is clear and unambiguous

4: finiteness: if we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after finite number of steps.

5. Effectiveness: Every instruction must be very basic so that it can be carried out, in principle by a person using only a pencil and paper. It is not enough that each operation be definite as in criterion 3 it also must be feasible.

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. The term "analysis of algorithms" was coined by Donald Knuth.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as time complexity, or volume of memory, known as space complexity.

The need for Analysis:

In this chapter, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using differentalgorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need tocalculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

- **Worst-case** − The maximum number of steps taken on any instance of size **a**.
- **Best-case** − The minimum number of steps taken on any instance of size **a**.
- **Average case** − An average number of steps taken on any instance of size **a**.
- **Amortized** − A sequence of operations applied to the input of size **a** averaged over time.

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa.In this context, if we compare bubble sort and merge sort. Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

Bubble Sort is an elementary sorting algorithm, which works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.

Algorithm: Sequential-Bubble-Sort (A)

```
fori← 1 to length [A] do
for j ← length [A] down-to i +1 do
  if A[A] < A[j - 1] then
    Exchange A[j] ↔ A[j-1]
```

Implementation:

```
  void bubbleSort (int numbers[], int array_size){
    int i, j, temp;
   for (i = (array_size - 1); i >= 0; i--){
        for (j = 1; j <= i; j++)
            if (numbers[j - 1] > numbers[j]) {
                temp = numbers[j-1]; numbers[j - 1] = numbers[j]; numbers[j] = temp;
            }
    }
}
```

Analysis:

Here, the number of comparisons are

$1 + 2 + 3 +...+ (n - 1) = n(n - 1)/2 = O(n^2)$

Clearly, the graph shows the ***n^2*** nature of the bubble sort.

In this algorithm, the number of comparison is irrespective of the data set, i.e. whether the provided input elements are in sorted order or in reverse order or at random.

Memory requirement:

Unsorted list:	5 2 1 4 3 7 6

1st iteration:

5 > 2 swap	2 5 1 4 3 7 6
5 > 1 swap	2 1 5 4 3 7 6
5 > 4 swap	2 1 4 5 3 7 6
5 > 3 swap	2 1 4 3 5 7 6
5 < 7 no swap	2 1 4 3 5 7 6
7 > 6 swap	2 1 4 3 5 6 7

2nd iteration:

2 > 1 swap	1 2 4 3 5 6 7
2 < 4 no swap	1 2 4 3 5 6 7
4 > 3 swap	1 2 3 4 5 6 7
4 < 5 no swap	1 2 3 4 5 6 7
5 < 6 no swap	1 2 3 4 5 6 7

There is no change in 3rd, 4th, 5th and 6th iteration.

Finally,

The sorted list is 1 2 3 4 5 6 7

EXPERIMENT NO: 2

Aim: Study of Divide and Conquer: Write a program to implement Quick Sort.

Objective: To learn & implement divide & conquer algorithm.

Theory:

Divide and conquer approach, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

Generally, divide-and-conquer algorithms have three parts −

- **Divide the problem** into a number of sub-problems that are smaller instances of the same problem.
- **Conquer the sub-problems** by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- **Combine the solutions** to the sub-problems into the solution for the original problem.

Pros and cons of divide and conquer Approach

Divide and conquer approach supports parallelism as sub-problems are independent. Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously.

In this approach, most of the algorithms are designed using recursion, hence memory management is very high. For recursive function stack is used, where function state needs to be stored.

Quick Sort:

QuickSort 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 quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort 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
    }
}
```

PartitionAlgorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. 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)
}
```

Illustration of partition() :

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0  1   2   3   4   5   6
```

low = 0, high = 6, pivot = arr[h] = 70
Initialize index of smaller element, **i = -1**

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing

// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

<u>EXPERIMENT NO: 3</u>

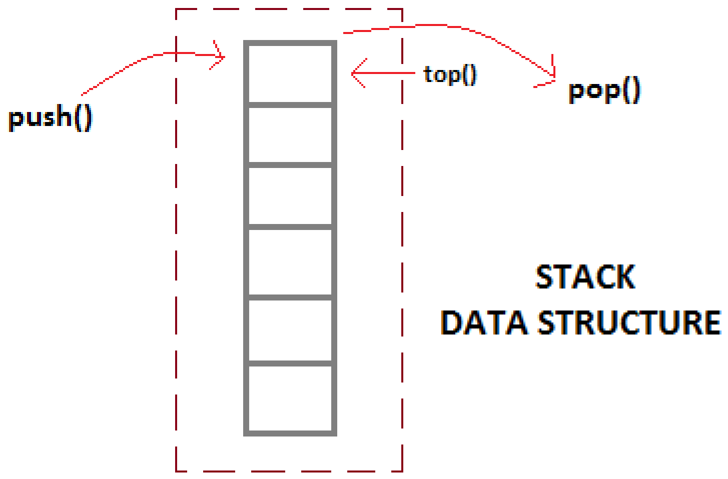
<u>Aim:</u> Implement basic algorithms and perform analysis of complexities (stack, queue, tree, and graph)

<u>Objective</u>: Revise the algorithms of basic data structures and analyze them.

<u>Theory:</u>

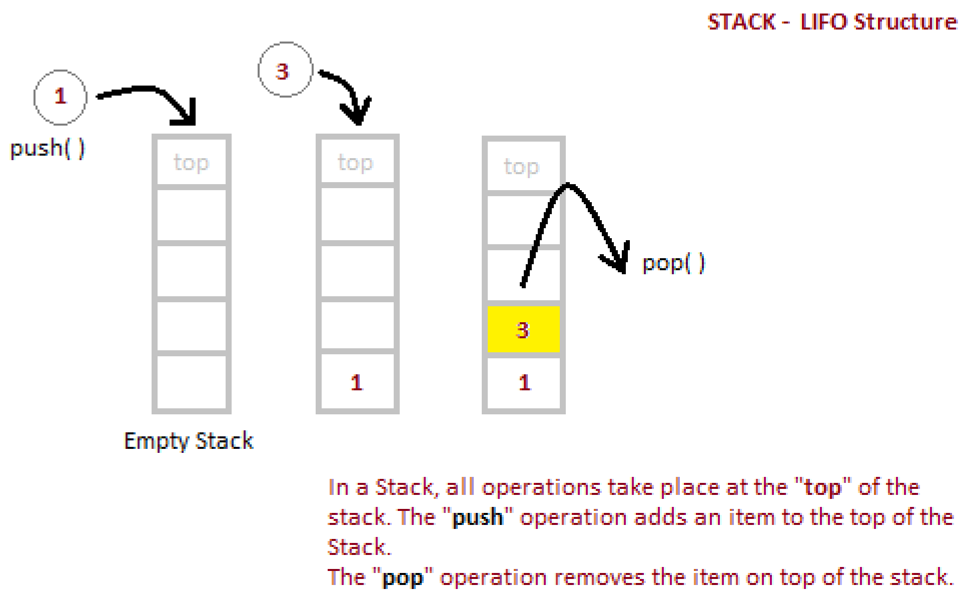
<u>Stack Data Structure</u>

Stack is an abstract data type with a bounded (predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack; the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.



Implementation of Stack

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.


STACK - LIFO Structure
1
push( )
3
top
top
top
pop( )
3
1
1
Empty Stack
In a Stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the Stack.
The "pop" operation removes the item on top of the stack.

Algorithm for PUSH operation

1. Check if the stack is full or not.
2. If the stack is full,then print error of overflow and exit the program.
3. If the stack is not full,then increment the top and add the element .

Algorithm for POP operation

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.

/* Below program is written in C++ language */

```
Class Stack
{
              int top;
             public:
            int a[10];        //Maximum size of Stack Stack()
             Stack()
             {
                         top = -1;
             }
};

  void Stack::push(int x)
    {
            if( top >= 10)
            {
             cout << "Stack Overflow";
            }
            else {
                        a[++top] = x;
            cout << "Element Inserted";
                        }
            }

 int Stack::pop()
 {
  if(top < 0)
{
 cout << "Stack Underflow";
return 0;
}
 else {
  int d = a[top--];
  return d;
 }

 }

    void Stack::isEmpty()
 {
   if(top < 0)
             {
             cout << "Stack is empty";
             }
             else
            {
            cout << "Stack is not empty";

            }
```

Position of Top	**Status of Stack**
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

Analysis of Stacks

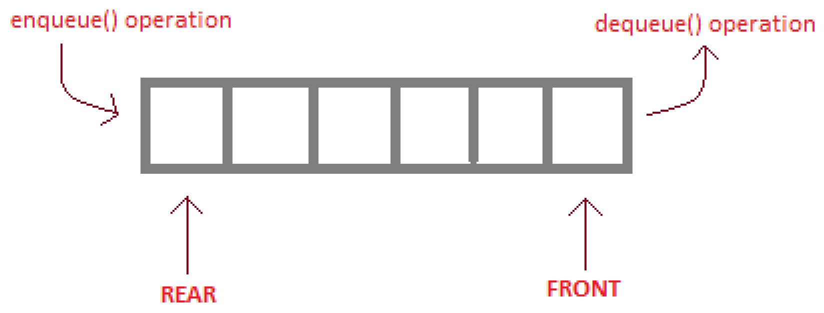
Below mentioned are the time complexities for various operations that can be performed on the Stack data structure

- **Push Operation** : O(1)
- **Pop Operation** : O(1)
- **Top Operation** : O(1)
- **Search Operation** : O(n)

<u>**Queue Data Structure**</u>

Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called **REAR**(also called tail), and the deletion of existing element takes place from the other end called as **FRONT**(also called head). This makes queue as FIFO(First in First Out) data structure, which means that element inserted first will also be removed first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



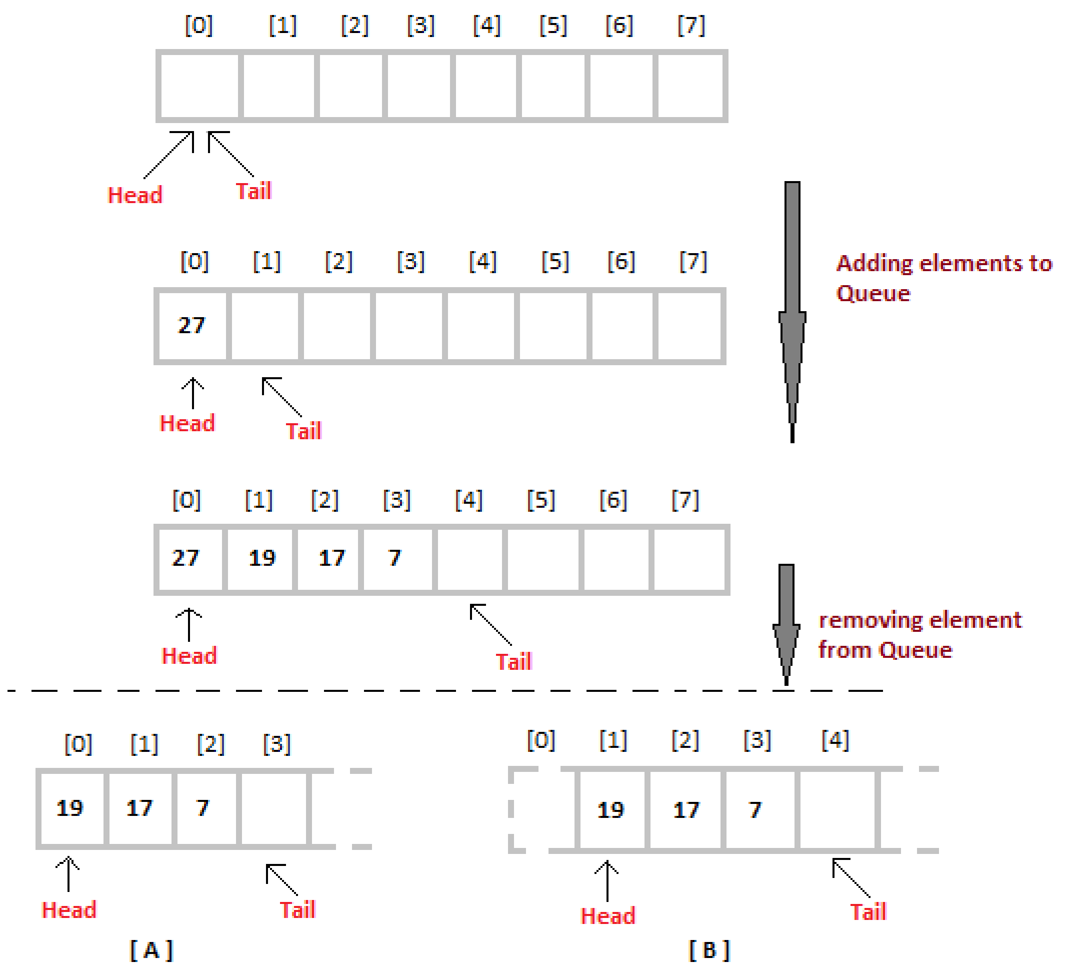
enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Implementation of Queue

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array. Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.



When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position. In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an overhead of shifting the elements one position forward every time we remove the first element. In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the size on Queue is reduced by one space each time.

Algorithm for ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then print overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then print underflow error and exit the program.
3. If the queue is not empty, then print the element at the head and increment the head.

```
/* Below program is written in C++ language */
#define SIZE 100 class Queue
{
int a[100];
int rear;        //same as tail int front;           //same as head

public:
Queue()
{
rear = front = -1;
}
void enqueue(int x); //declaring enqueue, dequeue and display functions int dequeue();
void display();
}


void Queue :: enqueue(int x)
{
if( rear = SIZE-1)
{
cout << "Queue is full";
}
else
{
a[++rear] = x;
}
}


int queue :: dequeue()
{
return a[++front];     //following approach [B], explained above
}
```

```
void queue :: display()
{
int i;
for( i = front; i <= rear; i++)
{
cout << a[i];
}
}
```

//To implement approach [A], you simply need to change the dequeue method, and include a forloop which will shift all the remaining elements one position.

```
return a[0]; //returning first element

for (i = 0; i < tail-1; i++)        //shifting all other elements

{

a[i]= a[i+1];

tail--;

}
```

Analysis of Queue

- Enqueue : **O(1)**
- Dequeue : **O(1)**
- Size : **O(1)**

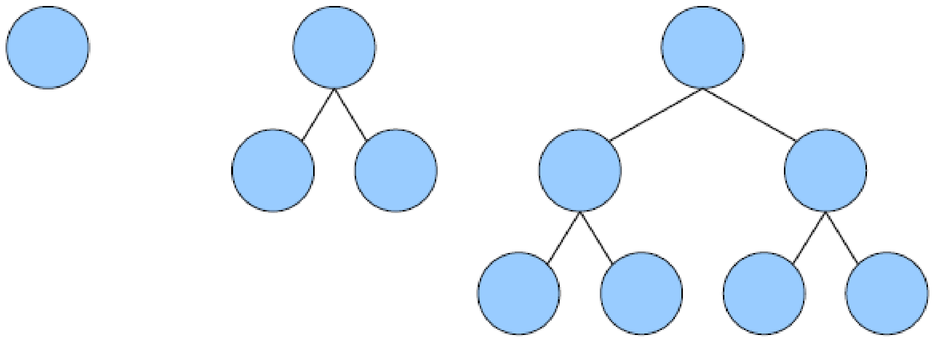
<u>Tree Data Structure</u>

Fastest Running Time

The find, insert and delete algorithms start at the tree root and a follow path down to, at worst case, the leaf at the very lowest level. The total number of steps of these algorithms is, therefore, the largest level of the tree, which is called the ***depth*** of the tree.

The best (fastest) running time occurs when the binary search tree is ***full*** – in which case the search processes used by the algorithms perform like a binary search.

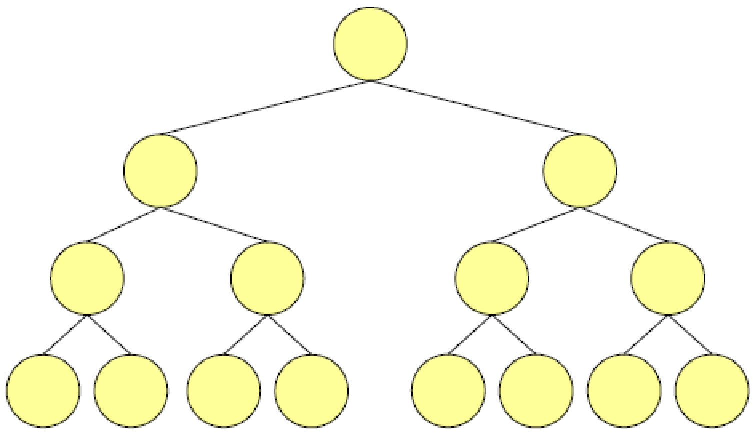
Let's verify this. A ***full binary tree*** is one in which nodes completely fill every level. For example, here are the unique full binary trees of which have tree depths of 0, 1 and 2:



As you can verify by looking at the examples, a full tree has 1 node at level 0, 2 nodes at level 1, 4 nodes at level 2 and so on. Thus, it will have 2*d* nodes at level *d*. Adding these quantities, the total number of nodes *n* for a full binary tree with depth *d* is:

n = 20 + 21 + 22 + … + 2*d* = 2*d*+1 – 1

For example, the full binary tree of depth 2 above has 23 – 1 = 7 nodes. The binary tree below isa full tree of depth 3 and has 24 – 1 = 15 nodes.



Now, considering the formula above for the number of nodes in a full binary search tree:
$n = 2_{d+1} - 1$

Solving for *d*, we get:

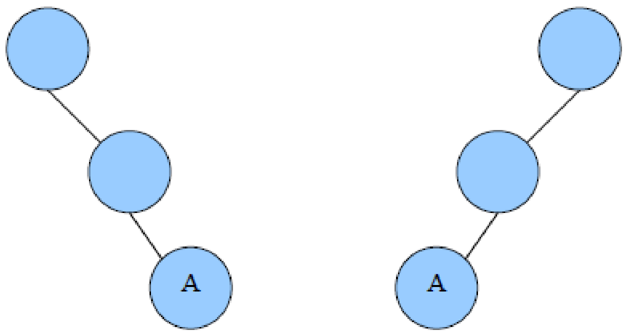
$$
\begin{aligned}
&n+1 = 2^{d+1} \\
&\lg(n+1) = \lg(2^{d+1}) \\
&\lg(n+1) = (d+1)\lg(2) \\
&\lg(n+1) = d+1 \\
&d = \lg(n+1) - 1 \\
&d = \lfloor \lg(n) \rfloor
\end{aligned}
$$

For example, the depth of a full binary search tree with 15 nodes is 3.

In other words, the depth of a binary search tree with n nodes can be no less than lg(n) and so the running time of the find, insert and delete algorithms can be no less than lg(n). A full binary search tree is said to be balanced because every node's proper descendants are divided evenly between its left and right subtrees. Thus, the search algorithm employed by the find, insert and delete operations perform like a binary search.

Slowest Running Time

As a binary search tree becomes more and more unbalanced, the performance of the find, insert and delete algorithms degrades until reaching the worst case of O(*n*), where *n* is the number of nodes in the tree. For example, he number of comparisons needed to find the node marked *A* in the binary search trees below is 3, which is equal to the number of nodes.



Binary search trees, such as those above, in which the nodes are in order so that all links are to right children (or all are to left children), are called ***skewed trees***.

Binary Tree

Best Time	**Average Time**	**Worst Time**
O(lg *n*)	O(lg n)	O(*n*)

<u>Graph Data Structure</u>

Initially it allows visiting vertices of the graph only, but there are hundreds of algorithms for graphs, which are based on DFS. Therefore, understanding the principles of depth-first search is quite important to move ahead into the graph theory. The principle of the algorithm is quite simple: to go forward (in depth) while there is such possibility, otherwise to backtrack.

Algorithm

In DFS, each vertex has three possible colors representing its state:

○ white: vertex is unvisited;

◍ gray: vertex is in progress;

● black: DFS has finished processing the vertex.

NB. For most algorithms boolean classification *unvisited / visited* is quite enough, but we show general case here.

Initially all vertices are white (unvisited). DFS starts in arbitrary vertex and runs as follows:

1. Mark vertex **u** as gray (visited).
2. For each edge **(u, v)**, where **u** is white, run depth-first search for **u** recursively.
3. Mark vertex **u** as black and backtrack to the parent.

DFS doesn't go through all edges. The vertices and edges, which depth-first search has visited is a **tree**. This tree contains all vertices of the graph (if it is connected) and is called *graph spanning tree*. This tree exactly corresponds to the recursive calls of DFS.

If a graph is disconnected, DFS won't visit all of its vertices. For details, see *finding connected components algorithm***.**

Complexity analysis

Assume that graph is connected. Depth-first search visits every vertex in the graph and checks every edge its edge. Therefore, DFS complexity is O(V + E). As it was mentioned before, if an adjacency matrix is used for a graph representation, then all edges, adjacent to a vertex can't be found efficiently, that results in O(V^2)complexity. You can find strong proof of the DFS complexity issues in [1].

Code snippets

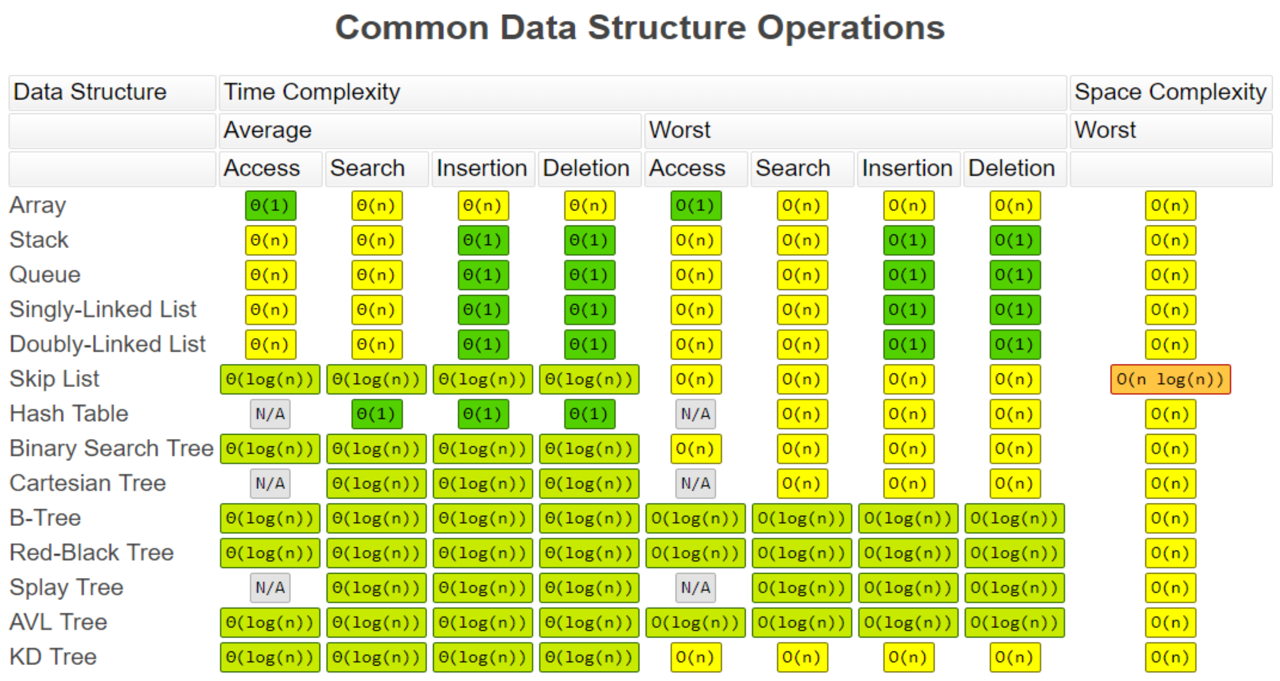
In truth the implementation stated below gives no yields. You will fill an actual use of DFS in further tutorials.

```
public class Graph {
        enum VertexState {
       White, Gray, Black
    }
    public void DFS()

    {
        VertexState state[] = new VertexState[vertexCount];

        for (int i = 0; i < vertexCount; i++)
            state[i] = VertexState.White;
        runDFS(0, state);
    }
    public void runDFS(int u, VertexState[] state)
    {
        state[u] = VertexState.Gray;
        for (int v = 0; v < vertexCount; v++)
            if (isEdge(u, v) && state[v] == VertexState.White)
                runDFS(v, state);
        state[u] = VertexState.Black;
    }
}
```

Summary



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

EXPERIMENT NO: 4

Aim: Greedy method: Design, Implement and analyse Prim's and Kruskal's algorithm for minimum cost spanning Tree.

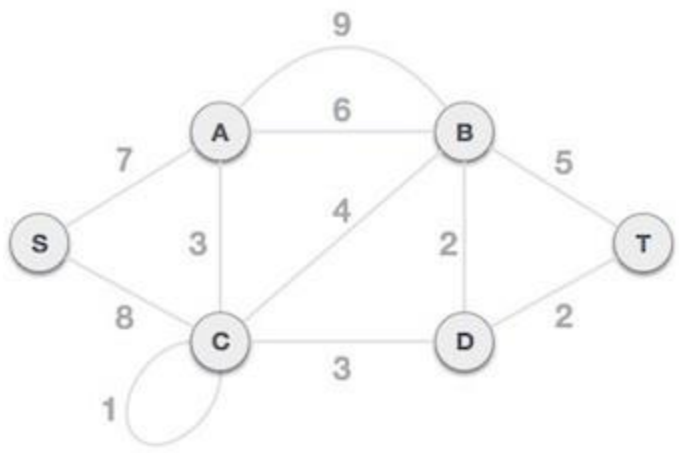
Objective: Implement Prim's and Kruskal's Algorithm.

Theory: **Prim's algorithm** is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

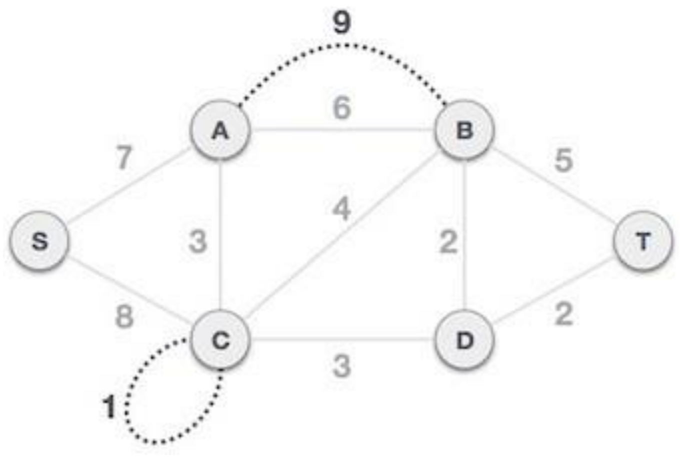
Prim's algorithm is also a **Greedy algorithm**. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect 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 Minimum Spanning Tree.

A group of edges that connects two set of vertices in a graph is called cut in graph theory. So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from thecut and include this vertex to MST Set (the set that contains already included vertices).

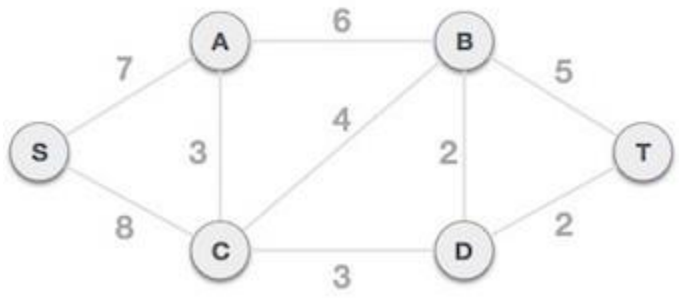
To understand Prim's Algorithm let us consider following example:



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

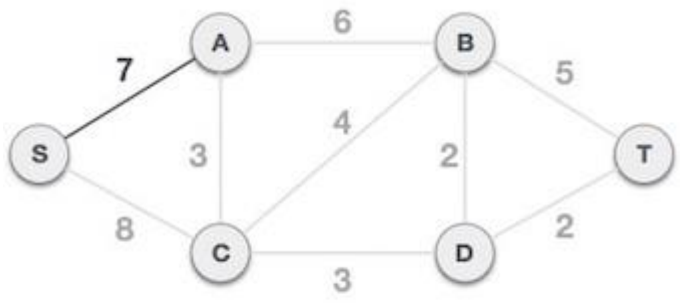


Step 2 - Choose any arbitrary node as root node

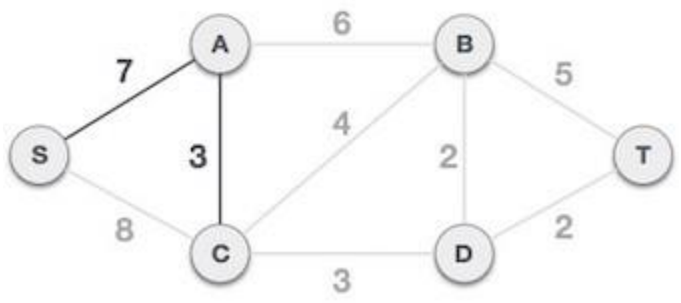
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

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

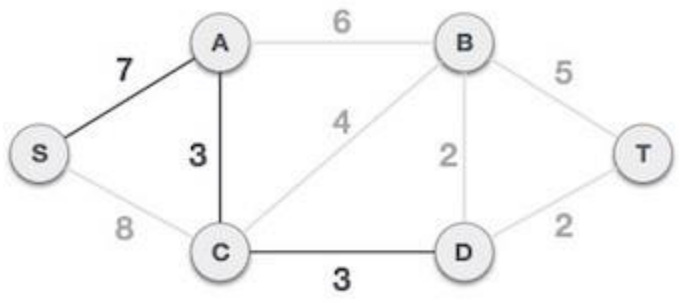
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



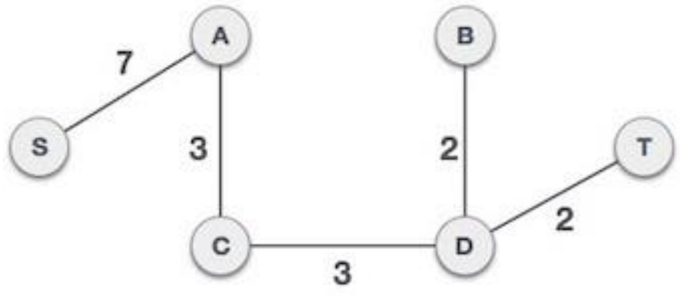
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.

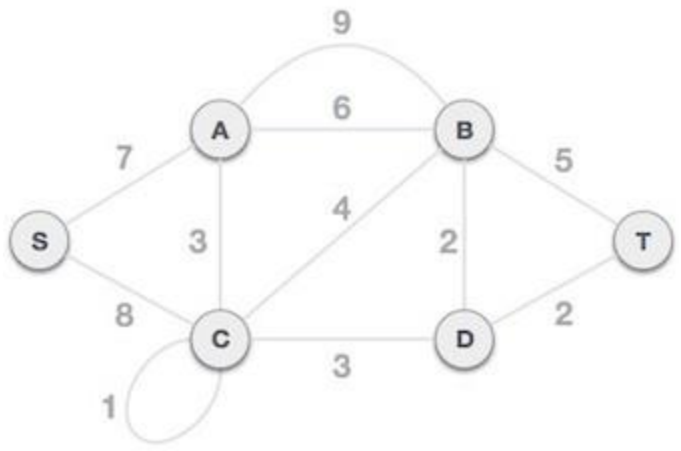


We may find that the output spanning tree of the same graph using two different algorithms is same.

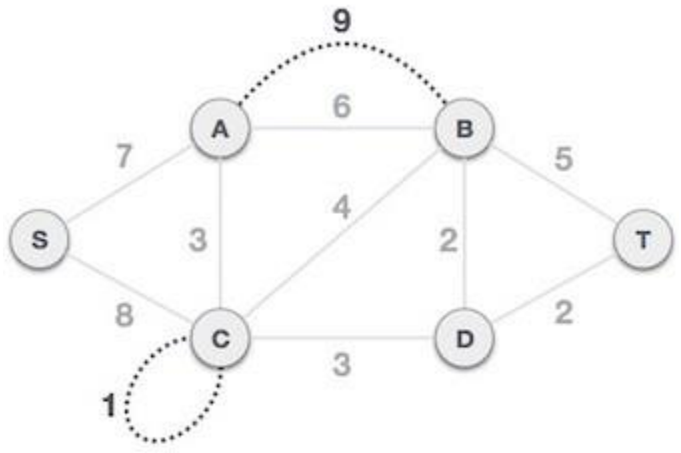
Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

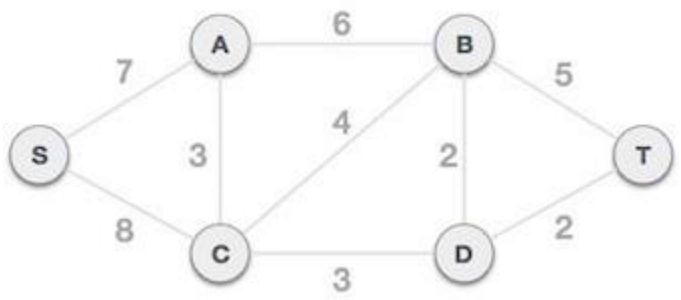
To contrast with Prim's algorithm and to understand Kruskal's algorithm better, we shall use the same example :



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

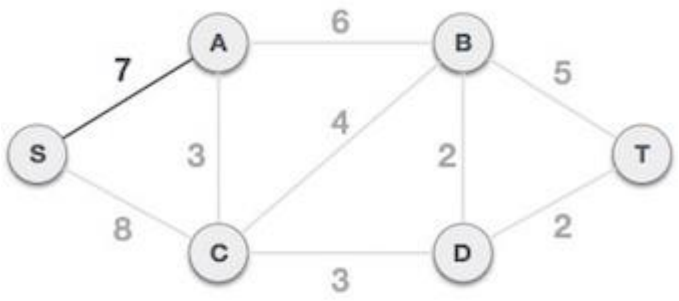


Step 2 - Choose any arbitrary node as root node

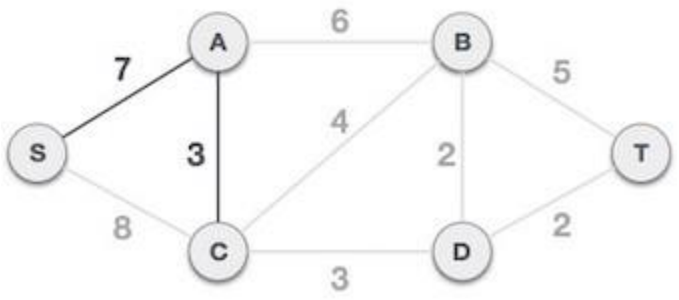
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

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

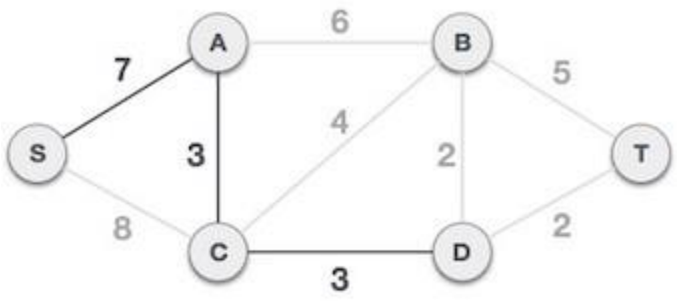
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



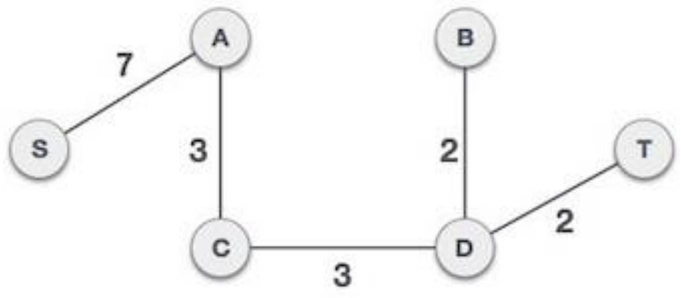
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



<u>Algorithm:</u>

Prim’s Algorithm

```
MST-PRIM(G, w, r)

1.     for each u  V [G]

2.     do key[u] ← ∞

3.     π[u] ← NIL

4.     key[r] ← 0

5.     Q ← V [G]

6.     while Q ≠ ∅

7.             do u ← EXTRACT-MIN(Q)

8.                     for each v  Adj[u]

9.                             do if v  Q and w(u, v) < key[v]

10.                                    then π[v] ← u

11.                                            key[v] ← w(u, v)
```

Kruskal’s Algorithm

```
MST-KRUSKAL(G, w)
1.      A ← ∅
2.      for each vertex v   V[G]
3.              do MAKE-SET(v)
4.      sort the edges of E into nondecreasing order by weight w
5.      for each edge (u, v)  E, taken in nondecreasing order by weight
6.              do if FIND-SET(u) ≠ FIND-SET(v)
7.                      then A ← A  {(u, v)}
8.                              UNION(u, v)
9.      return A
```

<u>Program:</u>

Prim’s Algorithm

```c
#include<stdio.h>

#include<conio.h>

int a,b,u,v,n,i,j,ne=1;

int visited[10]={0},min,mincost=0,cost[10][10];

void main()

{

	clrscr();

	printf("\nEnter the number of nodes:");

	scanf("%d",&n);

	printf("\nEnter the adjacency matrix:\n");

	for(i=1;i<=n;i++)

	for(j=1;j<=n;j++)

	{

		scanf("%d",&cost[i][j]);

		if(cost[i][j]==0)

			cost[i][j]=999;

```

```c
    }

    visited[1]=1;

    printf("\n");

    while(ne < n)

    {

        for(i=1,min=999;i<=n;i++)

        for(j=1;j<=n;j++)

        if(cost[i][j]< min)

        if(visited[i]!=0)

        {

            min=cost[i][j];

            a=u=i;

            b=v=j;

        }

        if(visited[u]==0 || visited[v]==0)

        {

            printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);

            mincost+=min;

            visited[b]=1;

        }

        cost[a][b]=cost[b][a]=999;

    }

    printf("\n Minimun cost=%d",mincost);

    getch();

}
```

Kruskal's Algorithm

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
	clrscr();
	printf("\n\tImplementation of Kruskal's algorithm\n");
	printf("\nEnter the no. of vertices:");
	scanf("%d",&n);
	printf("\nEnter the cost adjacency matrix:\n");
	for(i=1;i<=n;i++)
	{
		for(j=1;j<=n;j++)
		{
			scanf("%d",&cost[i][j]);
			if(cost[i][j]==0)
				cost[i][j]=999;
		}
	}
	printf("The edges of Minimum Cost Spanning Tree are\n");
	while(ne < n)
	{
		for(i=1,min=999;i<=n;i++)
		{
			for(j=1;j <= n;j++)
			{
				if(cost[i][j] < min)
				{
					min=cost[i][j];
					a=u=i;
					b=v=j;
				}
			}
		}
		u=find(u);
		v=find(v);
		if(uni(u,v))
		{
			printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
			mincost +=min;
		}
		cost[a][b]=cost[b][a]=999;
	}
	printf("\n\tMinimum cost = %d\n",mincost);
	getch();
}
```

```
                int find(int i)
    {
                while(parent[i])
                  i=parent[i];
                return i;
    }
                int uni(int i,int j)
                  {
                        if(i!=j)
        {
                    parent[j]=i;
                    return 1;
        }
        return 0;
}

```

<u>Time Complexity :</u>

Time Complexity of Prim's Algorithm 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.

In case of Kruskal's Algorithm time complexity is 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 atmost O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be atmost O(V^2), so O(LogV) are O(LogE) same. Therefore, overall time complexity is O(ElogE) or O(ElogV)

EXPERIMENT NO: 5

Aim: Implement Job Sequencing Problem and find its complexity.

Objective: Find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

Theory: **Job sequencing** consists of n jobs each associated with a deadline and profit and our objective is to earn maximum profit. We will earn profit only when job is completed on or before deadline. We assume that each job will take unit time to complete.

- In this problem we have n jobs j1, j2, … jn each has an associated deadline d1, d2, … dnand profit p1, p2, ... pn.
- Profit will only be awarded or earned if the job is completed on or before the deadline.
- We assume that each job takes unit time to complete.
- The objective is to earn maximum profit when only one job can be scheduled orprocessed at any given time.

Algorithm:

Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)

```
D(0) := J(0) := 0
k := 1
J(1) := 1 // means first job is selected
for i = 2 … n do
  r := k
  while D(J(r)) > D(i) and D(J(r)) ≠ r dor
    := r – 1
  if D(J(r)) ≤ D(i) and D(i) > r then
    for l = k … r + 1 by -1 do
      J(l + 1) := J(l)
      J(r + 1) := i
      k := k + 1
```

Let us consider, a set of n given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

It may happen that all of the given jobs may not be completed within their deadlines.

Assume, deadline of ith job Ji is di and the profit received from this job is pi. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Thus, $D(i) > 0$ for $1 \leqslant i \leqslant n$.

Initially, these jobs are ordered according to profit, i.e. $p_{1} \geqslant p_{2} \geqslant p_{3} \geqslant \:... \: \geqslant p_{n}$.

Program:

```
#include <stdio.h>

#define MAX 100

struct Job {
        char id[5];
        int deadline;
        int profit;
};

void jobSequencingWithDeadline(Job jobs[], int n);

int minValue(int x, int y) {
        if(x < y) return x;
        return y;
}

int main(void) {
        //variables
        int i, j;

        //jobs with deadline and profit
        Job jobs[5] = {
                {"j1", 2,  60},
                {"j2", 1, 100},
                {"j3", 3,  20},
                {"j4", 2,  40},
                {"j5", 1,  20},
        };
```

```
	//temp
	Job temp;

	//number of jobs
	int n = 5;

	//sort the jobs profit wise in descending order
	for(i = 1; i < n; i++) {
		for(j = 0; j < n - i; j++) {
			if(jobs[j+1].profit > jobs[j].profit) {
				temp = jobs[j+1];
				jobs[j+1] = jobs[j];
				jobs[j] = temp;
			}
		}
	}

	printf("%10s %10s %10s\n", "Job", "Deadline", "Profit");
	for(i = 0; i < n; i++) {
		printf("%10s %10i %10i\n", jobs[i].id, jobs[i].deadline, jobs[i].profit);
	}

	jobSequencingWithDeadline(jobs, n);

	return 0;
}

void jobSequencingWithDeadline(Job jobs[], int n) {
	//variables
	int i, j, k, maxprofit;

	//free time slots
	int timeslot[MAX];

	//filled time slots
	int filledTimeSlot = 0;

	//find max deadline value
	int dmax = 0;
	for(i = 0; i < n; i++) {
		if(jobs[i].deadline > dmax) {
			dmax = jobs[i].deadline;
		}
	}

	//free time slots initially set to -1 [-1 denotes EMPTY]
```

```
    for(i = 1; i <= dmax; i++) {
        timeslot[i] = -1;
    }

    printf("dmax: %d\n", dmax);

    for(i = 1; i <= n; i++) {
        k = minValue(dmax, jobs[i - 1].deadline);
        while(k >= 1) {
            if(timeslot[k] == -1) {
                timeslot[k] = i-1;
                filledTimeSlot++;
                break;
            }
            k--;
        }

        //if all time slots are filled then stop
        if(filledTimeSlot == dmax) {
            break;
        }
    }

    //required jobs
    printf("\nRequired Jobs: ");
    for(i = 1; i <= dmax; i++) {
        printf("%s", jobs[timeslot[i]].id);

        if(i < dmax) {
            printf(" --> ");
        }
    }

    //required profit
    maxprofit = 0;
    for(i = 1; i <= dmax; i++) {
        maxprofit += jobs[timeslot[i]].profit;
    }
    printf("\nMax Profit: %d\n", maxprofit);
}
```

Analysis:

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is $O(n^2).

EXPERIMENT NO: 6

Aim: Dynamic programming I: Design, Implement and analyze shortest path algorithm.

Objective: To find shortest path between nodes with dynamic programming.

Theory:

Dynamic programming:

Dynamic programming (usually referred to as **DP**) is a very powerful technique to solve a particular class of problems. It demands very elegant formulation of the approach and simple thinking and the coding part is very easy. The idea is very simple, If you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again.. Shortly *'Remember your Past'* :) . If the given problem can be broken up in to smaller sub-problems and these smaller sub problems are in turn divided in to still-smaller ones, and in this process, if you observe some over-lapping sub problems, then its a big hint for DP. Also, the optimal solutions to the sub problems contribute to the optimal solution of the given problem (referred to as the Optimal Substructure Property).

There are two ways of doing this.

1.) Top-Down : Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as ***Memorization***.

2.) Bottom-Up: Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial sub problem, up towards the given problem. In this process, it is guaranteed that the sub problems are solved before solving the problem. This is referred to as ***Dynamic Programming***.

Types of Shortest Path Algorithms

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

There are two main types of shortest path algorithms, single-source and all-pairs. Both types have algorithms that perform best in their own way. All-pairs algorithms take longer to run because of the added complexity. All shortest path algorithms return values that can be used to find the shortest path, even if those return values vary in type or form from algorithm toalgorithm.

Single-source

Single-source shortest path algorithms operate under the following principle:

Definition:

Given a graph **G**, with vertices **V**, edges **E** with weight function $\mathbf{w(u,v)=w_{u,v}}$ and a single source vertex, **s**, return the shortest paths from to all other vertices in **V**.

If the goal of the algorithm is to find the shortest path between only two given vertices, **s** and **t**, then the algorithm can simply be stopped when that shortest path is found. Because there is no way to decide which vertices to "finish" first, all algorithms that solve for the shortest path between two given vertices have the same worst-case asymptotic complexity as single-source shortest path algorithms.

This paradigm also works for the *single-destination shortest path* problem. By reversing all of the edges in a graph, the single-destination problem can be reduced to the single-source problem. So, given a destination vertex, **t**, this algorithm will find the shortest paths *starting* at all other vertices and ending at **t**.

All-pairs

All-pairs shortest path algorithms follow this definition:

Definition:

Given a graph **G**, with vertices **V**, edges **E** with weight function $\mathbf{w(u,v)=w_{u,v}}$ return the shortest path from **u** to **v** for all **(u,v)** in **V**.

The most common algorithm for the all-pairs problem is the floyd-warshall algorithm. This algorithm returns a matrix of values **M**, where each cell $\mathbf{M_{i,j}}$ is the distance of the shortest path from vertex **i** to vertex **j** . Path reconstruction is possible to find the actual path taken to achieve that shortest path, but it is not part of the fundamental algorithm.

Bellman-Ford algorithm

The Bellman-Ford algorithm solves the single-source problem in the general case, where edges can have negative weights and the graph is directed. If the graph is undirected, it will have to modified by including two edges in each direction to make it directed.

Bellman-Ford has the property that it can detect negative weight cycles reachable from the source, which would mean that no shortest path exists. If a negative weight cycle existed, a path could run infinitely on that cycle, decreasing the path cost to $-\infty$.

If there is no negative weight cycle, then Bellman-Ford returns the weight of the shortest path along with the path itself.

Dijkstra's algorithm

Dijkstra's algorithm makes use of breadth-first search (which is not a single source shortest path algorithm) to solve the single-source problem. It does place one constraint on the graph: there can be no negative weight edges. However, for this one constraint, Dijkstra greatly improves on the runtime of Bellman-Ford.

Dijkstra's algorithm is also sometimes used to solve the all-pairs shortest path problem by simply running it on all vertices in **V**. Again, this requires all edge weights to be positive.

Topological Sort

For graphs that are directed acyclic graphs (DAGs), a very useful tool emerges for finding shortest paths. By performing a topological sort on the vertices in the graph, the shortest path problem becomes solvable in linear time.

A topological sort is an ordering all of the vertices such that for each edge (u,v) in **E**, comes before in the ordering. In a DAG, shortest paths are always well defined because even if there arenegative weight edges, there can be no negative weight cycles.

Floyd-Warshall algorithm

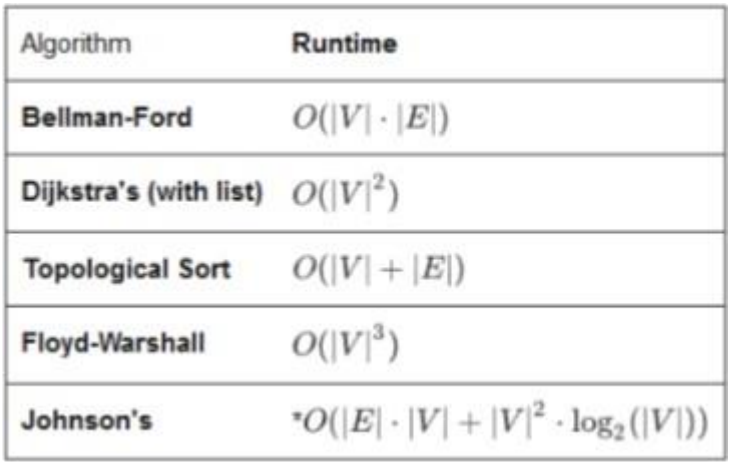
The Floyd-Warshall algorithm solves the all-pairs shortest path problem. It uses a dynamic programming approach to do so. Negative edge weight may be present for Floyd-Warshall.

Floyd-Warshall takes advantage of the following observation: the shortest path from A to C is either the shortest path from A to B plus the shortest path from B to C *or* it's the shortest path from A to C that's already been found. This may seem trivial, but it's what allows Floyd- Warshall to build shortest paths from smaller shortest paths, in the classic dynamic programmingway.

Johnson's Algorithm

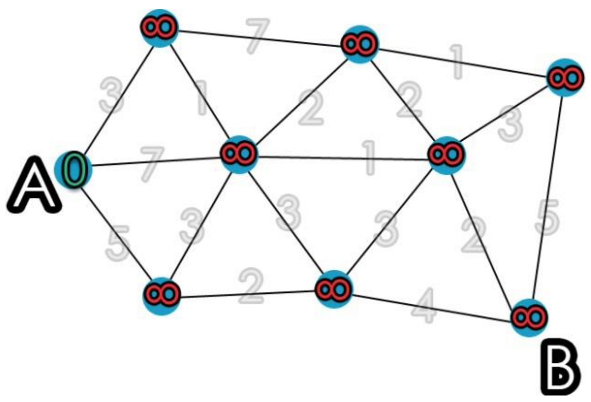
While Floyd-Warshall works well for dense graphs (meaning many edges), Johnson's algorithm works best for sparse graphs (meaning few edges). In sparse graphs, Johnson's algorithm has a lower asymptotic running time compared to Floyd-Warshall.

Johnson's algorithm takes advantage of the concept of reweighting, and it uses Dijkstra's algorithm on many vertices to find the shortest path once it has finished reweighting the edges.



Algorithm	Runtime
Bellman-Ford	$O(\lvert V\rvert \cdot \lvert E\rvert)$
Dijkstra's (with list)	$O(\lvert V\rvert^2)$
Topological Sort	$O(\lvert V\rvert + \lvert E\rvert)$
Floyd-Warshall	$O(\lvert V\rvert^3)$
Johnson's	${}^*O(\lvert E\rvert \cdot \lvert V\rvert + \lvert V\rvert^2 \cdot \log_2(\lvert V\rvert))$

Dijkstra’s algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.



The graph has the following:

- vertices, or nodes, denoted in the algorithm by **v** or **u**;
- weighted edges that connect two nodes: **(u,v)** denotes an edge, and **w(u,v)** denotes its weight. In the diagram on the right, the weight for each edge is written in gray.

This is done by initializing three values:

- **dist**, an array of distances from the source node to each node in the graph, initialized the following way: **dist(s) = 0**; and for all other nodes **v, dist(v)** =∞ . This is done at the beginning because as the algorithm proceeds, the from the source to each node in the graph will be recalculated and finalized when the shortest distance to is found

- **Q**, a queue of all nodes in the graph. At the end of the algorithm's progress, **Q** will be empty.
- **S**, an empty set, to indicate which nodes the algorithm has visited. At the end of the algorithm's run, **S** will contain all the nodes of the graph.

<u>Algorithm:</u>

The algorithm proceeds as follows:

1. While **Q** is not empty, pop the node **v**, that is not already in **S**, from **Q** with the smallest **dist(v)**. In the first run, source node **s** will be chosen because **dist(s)** was initialized to 0. In the next run, the next node with the smallest **dist** value is chosen.
2. Add node **v** to **S**, to indicate that **v** has been visited
3. Update **dist** values of adjacent nodes of the current node **v** as follows: for each new adjacent node **u**,

- if **dist(v) + < weight(u,v) < dist(u)** , there is a new minimal distance found for **u**, so update **dist(u)** to the new minimal distance value;
- otherwise, no updates are made to **dist(u)**.

The algorithm has visited all nodes in the graph and found the smallest distance to each node. **dist** now contains the shortest path tree from source **s**.

Note: The weight of an edge **(u,v)** is taken from the value associated with **(u,v)** on the graph.

Implementation:

This is pseudocode for Dijkstra's algorithm, mirroring Python syntax. It can be used in order to implement the algorithm in any language.

```
function Dijkstra(Graph, source):

    dist[source]  := 0                    // Distance from source to source is set to 0

    for each vertex v in Graph:           // Initializations

        if v ≠ source

            dist[v]  := infinity          // Unknown distance function from source to each node set to infinity

        add v to Q                        // All nodes initially in Q
    while Q is not empty:                  // The main loop
        v := vertex in Q with min dist[v] // In the first run-through, this vertex is the source node
        remove v from Q

        for each neighbor u of v:           // where neighbor u has not yet been removed from Q.
            alt := dist[v] + length(v, u)
            if alt < dist[u]:               // A shorter path to u has been found
                dist[u]  := alt            // Update distance of u

    return dist[]
  end function
}
```

Analysis

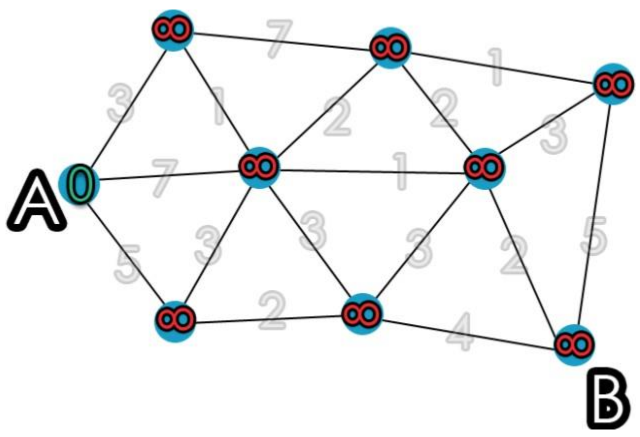
These algorithms have been improved upon over time. Dijkstra's algorithm, for example, was initally implemented using a list, and had a runtime of **$O(|V|^2)$**. However, when a binary heap is used, a runtime of **$O((|E| + |V|). \log_2(|V|))$** has been achieved. When a fibonacci heap is used, one implementation can achieve **$O(|E| + |V|. \log_2(|V|))$** while another can do **$O(|E|.\log_2(\log_2(|C|)))$**where **$|C|$** is a bounded constant for edge weight.

Bellman-Ford has been implemented in **$O(|V|^2. \log_2(|V|))$**. This implementation can be efficient if used on the right kind of graph (sparse).

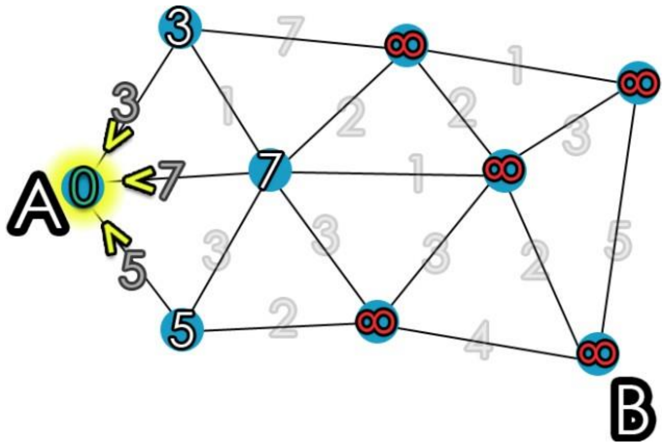
Example:

We step through Dijkstra's algorithm on the graph used in the algorithm above:

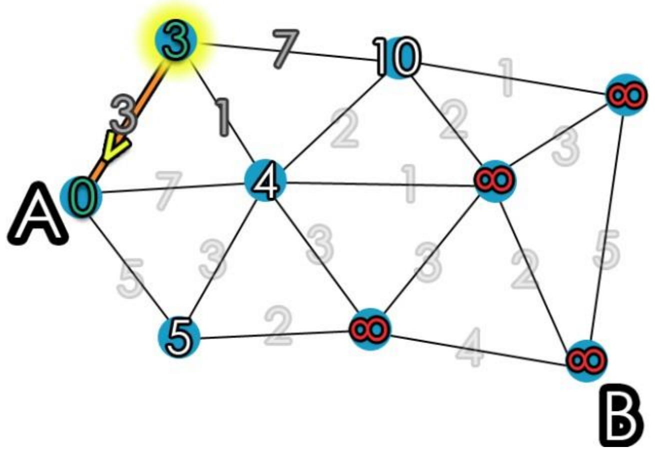
1. Initialize distances according to the algorithm.



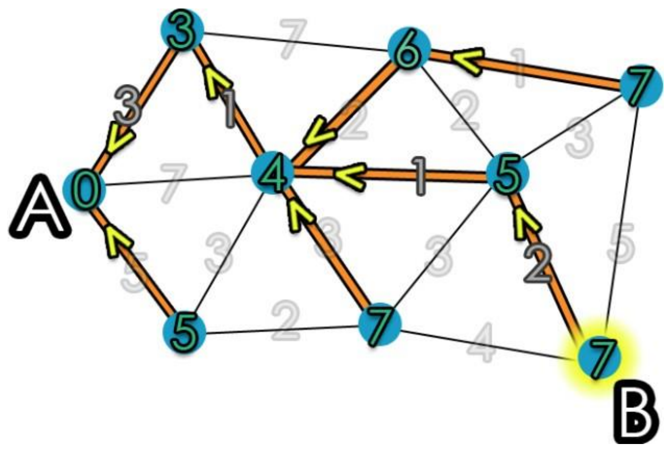
2. Pick first node and calculate distances to adjacent nodes.



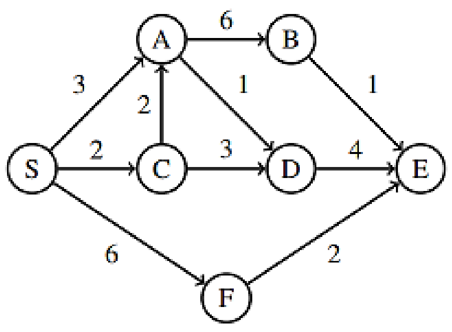
3. Pick next node with minimal distance; repeat adjacent node distance calculations.



4. Final result of shortest-path tree



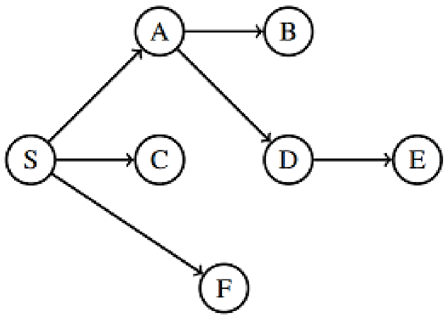
Example:



Question

Run Dijkstra's on the following graph and determine the resulting shortest path tree.

Dijkstra will visit the vertices in the following order: **S, C, A, D, F, E, B**. The closer edges will be relaxed first. As a result, the parent of each node is as follows:



Solution

C Implemetation of Dijkstra's Algorithm

```
#include<stdio.h>
#include<conio.h>
#define INFINITY 9999
#define MAX 10

void dijikstra(int G[MAX][MAX], int n, int startnode);

void main(){
	int G[MAX][MAX], i, j, n, u;
	clrscr();
	printf("\nEnter the no. of vertices:: ");
	scanf("%d", &n);
	printf("\nEnter the adjacency matrix::\n");
	for(i=0;i < n;i++)
		for(j=0;j < n;j++)
			scanf("%d", &G[i][j]);
	printf("\nEnter the starting node:: ");
	scanf("%d", &u);
	dijikstra(G,n,u);
	getch();
}

void dijikstra(int G[MAX][MAX], int n, int startnode)
{
	int cost[MAX][MAX], distance[MAX], pred[MAX];
	int visited[MAX], count, mindistance, nextnode, i,j;
	for(i=0;i < n;i++)
		for(j=0;j < n;j++)
			if(G[i][j]==0)
				cost[i][j]=INFINITY;
			else
				cost[i][j]=G[i][j];
```

```
        for(i=0;i< n;i++)
        {
                distance[i]=cost[startnode][i];
                pred[i]=startnode;
                visited[i]=0;
        }
        distance[startnode]=0;
        visited[startnode]=1;
        count=1;
        while(count < n-1){
                mindistance=INFINITY;
                for(i=0;i < n;i++)
                        if(distance[i] < mindistance&&!visited[i])
                        {
                                mindistance=distance[i];
                                nextnode=i;
                        }
                visited[nextnode]=1;
                for(i=0;i < n;i++)
                        if(!visited[i])
                                if(mindistance+cost[nextnode][i] < distance[i])
                                {
                                        distance[i]=mindistance+cost[nextnode][i];
                                        pred[i]=nextnode;
                                }
                        count++;
        }


        for(i=0;i < n;i++)
                if(i!=startnode)
                {
                        printf("\nDistance of %d = %d", i, distance[i]);
                        printf("\nPath = %d", i);
                        j=i;
                        do
                        {
                                j=pred[j];
                                printf(" <-%d", j);
                        }
                        while(j!=startnode);
                }
}
```

<u>EXPERIMENT NO: 7</u>

<u>Aim:</u> Design, Implement and analyze Optimal Binary Search Trees.

<u>Objective:</u> To study Overlapping Sub problems with the help of dynamic programming.

<u>Theory:</u>
Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into sub problems and stores the results of sub problems to avoid computing the same results again. Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.
1) Overlapping subproblems
2) Optimal Substructure

Overlapping Subproblems:
Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. 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.

Optimal Substructure: A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

Binary tree: Binary Tress is a structure defined on a finite set of nodes that either contains no nodes, or is composed of three disjoint sets of nodes: a root node, a binary tree called its left subtree, and a binary tree called its right subtree.

Binary Search Tree: A binary search tree (BST) is a binary tree whose nodes are organized according to the binary search tree property: keys in the left subtree are all less than the key at the root; keys in the right subtree are all greater than the key at the root; and both subtrees are themselves BSTs.

Optimal binary search tree : An optimal binary search tree (Optimal BST), sometimes calleda weight-balanced binary tree, is a binary search tree which provides the smallest possible search time (or expected search time) for a given sequence of accesses (or access probabilities). Optimal BSTs are generally divided into two types: Static and Dynamic.

In the **static optimality** problem, the tree cannot be modified after it has been constructed. In this case, there exists some particular layout of the nodes of the tree which provides the smallest expected search time for the given access probabilities. Various algorithms exist to construct or approximate the statically optimal tree given the information on the access probabilities of the elements.

In the **dynamic optimality** problem, the tree can be modified at any time, typically by permitting tree rotations. The tree is considered to have a cursor starting at the root which it can move or use to perform modifications. In this case, there exists some minimal-cost sequence of these operations which causes the cursor to visit every node in the target access sequence in order. The splay tree is conjectured to have a constant competitive ratio compared to the dynamically optimal tree in all cases, though this has not yet been proven.

Program: Optimal Binary Search Tree using Dynamic Method in C

```
#include<stdio.h>
#include<conio.h>
#define MAX 10

void main()
{
char ele[MAX][MAX];
int w[MAX][MAX], c[MAX][MAX], r[MAX][MAX], p[MAX], q[MAX];
int temp=0, root, min, min1, n;
int i,j,k,b;
clrscr();
printf("Enter the number of elements:");
scanf("%d",&n);
printf("\n");
for(i=1; i <= n; i++)
{
printf("Enter the Element of %d:",i);
scanf("%d",&p[i]);
}
printf("\n");
for(i=0; i <= n; i++)
{
printf("Enter the Probability of %d:",i);
scanf("%d",&q[i]);
}
printf("W\t\tC\t\tR\n");
for(i=0; i <= n; i++)
{
for(j=0; j <= n; j++)
{
```

```
if(i == j)
{
w[i][j] = q[i];
c[i][j] = 0;
r[i][j] = 0;
printf("W[%d][%d]: %d\tC[%d][%d]: %d\tR[%d][%d]: %d\n",i,j,w[i][j],i,j,c[i][j],i,j,r[i][j]);
}
}
}
printf("\n");
for(b=0; b < n; b++)
{
for(i=0,j=b+1; j < n+1 && i < n+1; j++,i++)
{
if(i!=j && i < j)
{
w[i][j] = p[j] + q[j] + w[i][j-1];

min = 30000;
for(k = i+1; k <= j; k++)
{
min1 = c[i][k-1] + c[k][j] + w[i][j];
if(min > min1)
{
min = min1;
temp = k;
}
}
c[i][j] = min;
r[i][j] = temp;
}
printf("W[%d][%d]: %d\tC[%d][%d]: %d\tR[%d][%d]: %d\n",i,j,w[i][j],i,j,c[i][j],i,j,r[i][j]);
}
printf("\n");
}
printf("Minimum cost = %d\n",c[0][n]);
root = r[0][n];
printf("Root = %d \n",root);
getch();
}
```

EXPERIMENT NO: 8

Aim: Backtracking I: Write a program to implement 8 queens problem

Objective: To study backtracking algorithm.
Theory:

The Eight Queen’s problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal). More generally, the n queens problem places n queens on an n×n chessboard.

There are different solutions for the problem.

Backtracking | Set 3 (N Queen Problem)

History

Chess composer Max Bezzel published the eight queens puzzle in 1848. Franz Nauck published the first solutions in 1850.[2] Nauck also extended the puzzle to the *n* queens problem,with *n* queens on a chessboard of $n \times n$ squares.

Since then, many mathematicians, including Carl Friedrich Gauss, have worked on both the eight queens puzzle and its generalized *n*-queens version. In 1874, S. Gunther proposed a method using determinants to find solutions.[2] J.W.L. Glaisher refined Gunther's approach.

In 1972, Edsger Dijkstra used this problem to illustrate the power of what he called structured programming. He published a highly detailed description of a depth-first backtracking algorithm

Solutions

The eight queen’s puzzle has 92 distinct solutions. If solutions that differ only bythe symmetry operations of rotation and reflection of the board are counted as one, the puzzlehas 12 solutions. These are called *fundamental* solutions; representatives of each are shownbelow.

A fundamental solution usually has eight variants (including its original form) obtained by rotating 90, 180, or 270° and then reflecting each of the four rotational variants in a mirror in a fixed position. However, should a solution be equivalent to its own 90° rotation (as happens to one solution with five queens on a 5×5 board), that fundamental solution will have only two variants (itself and its reflection). Should a solution be equivalent to its own 180° rotation (butnot to its 90° rotation), it will have four variants (itself and its reflection, its 90° rotation and the reflection of that). If $n > 1$, it is not possible for a solution to be equivalent to its own reflection because that would require two queens to be facing each other. Of the 12 fundamental solutions to the problem with eight queens on an 8×8 board, exactly one (solution 12 below) is equal to its own 180° rotation, and none is equal to its 90° rotation; thus, the number of distinct solutions is 11×8 + 1×4 = 92 (where the 8 is derived from four 90° rotational positions and their reflections, and the 4 is derived from two 180° rotational positions and their reflections).

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.

```
1) Start in the leftmost column
2) If all queens are placed
    return true
3) 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.
3) If all rows have been tried and nothing worked, return false to trigger
    backtracking.
```

Implementation of Backtracking solution

```
/* C/C++ program to solve N Queen Problem using    backtracking */
#define N 4
#include<stdio.h>

/* A utility function to print solution */
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int 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 */
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    /* Check this row on left side */
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

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

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

    return true;
}
```

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

    /* Consider this column and try placing
       this queen in all rows one by one */
    for (int i = 0; i < N; i++)
    {
        /* Check if queen can be placed on
          board[i][col] */
        if ( isSafe(board, i, col) )
        {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

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

            /* 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 false;
}

/* 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.*/
bool 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) == false )
    {
      printf("Solution does not exist");
      return false;
    }

    printSolution(board);
    return true;
}

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

<u>Output: The 1 values indicate placements of queens</u>

```
0  0  1  0

1  0  0  0

0  0  0  1

0  1  0  0
```

EXPERIMENT NO: 9

Aim: Backtracking II: Implement Hamiltonian Cycle.

Objective: To learn backtracking with the help of Hamiltonian cycle.

Theory:

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

Input:

A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] isadjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edgefrom i to j, otherwise graph[i][j] is 0.

Output:

An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

```
(0)--(1)--(2)
 |   / \   |
 |  /   \  |
 | /     \ |
(3)-------(4)
```

And the following graph doesn’t contain any Hamiltonian Cycle.

```
(0)--(1)--(2)
 |   / \   |
 |  /   \  |
 | /     \ |
(3)      (4)
```

History

The Hamiltonian cycle was named after Sir William Rowan Hamilton who, in 1857, invented a puzzle-game which involved hunting for a Hamiltonian cycle. The game, called the Icosian game, was distributed as a dodecahedron graph with a hole at each vertex. To solve the puzzle or win the game one had to use pegs and string to find the Hamiltonian cycle — a closed loop that visited every hole exactly once.

Examples of Hamiltonian Graphs

Every complete graph with more than two vertices is a Hamiltonian graph. This follows from the definition of a complete graph: an undirected, simple graph such that every pair of nodes is connected by a unique edge.

The graph of every **platonic solid** is a Hamiltonian graph. So the graph of a cube, a tetrahedron, an octahedron, or an icosahedron are all Hamiltonian graphs with Hamiltonian cycles.
A graph with *n* vertices (where *n* > 3) is Hamiltonian if the sum of the degrees of every pair of non-adjacent vertices is *n* or greater. This is known as **Ore's theorem**.

Applications of Hamiltonian cycles and Graphs

A search for Hamiltonian cycles isn't just a fun game for the afternoon off. It has real applications in such diverse fields as computer graphics, electronic circuit design, mapping genomes, and operations research.

For instance, when mapping genomes scientists must combine many tiny fragments of genetic code ("reads", they are called), into one single genomic sequence (a 'superstring'). This can be done by finding a Hamiltonian cycle or Hamiltonian path, where each of the reads are considered nodes in a graph and each overlap (place where the end of one read matches the beginning of another) is considered to be an edge.

In a much less complex application of exactly the same math, school districts use Hamiltonian cycles to plan the best route to pick up students from across the district. Here students may be considered nodes, the paths between them edges, and the bus wishes to travel a route that will pass each students house exactly once.

Naive Algorithm

Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be n! (n factorial) configurations.

```
while there are untried conflagrations
{
  generate the next configuration
  if ( there are edges between two consecutive vertices of this
     configuration and there is an edge from the last vertex to the
     first ).
  {
     print this configuration;
     break;
  }

}
```

Backtracking Algorithm
Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

Implementation of Backtracking solution

```
/* C/C++ program for solution of Hamiltonian Cycle problem
   using backtracking */
#include<stdio.h>

// Number of vertices in the graph
#define V 5

void printSolution(int path[]);

/* A utility function to check if the vertex v can be added at
   index 'pos' in the Hamiltonian Cycle constructed so far (stored
   in 'path[]') */
bool isSafe(int v, bool graph[V][V], int path[], int pos)
{
    /* Check if this vertex is an adjacent vertex of the previously
       added vertex. */
    if (graph [ path[pos-1] ][ v ] == 0)
        return false;
    /* Check if the vertex has already been included.
      This step can be optimized by creating an array of size V */
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;

    return true;
}

/* A recursive utility function to solve hamiltonian cycle problem */
bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    /* base case: If all vertices are included in Hamiltonian Cycle */
    if (pos == V)
    {
        // And if there is an edge from the last included vertex to the
        // first vertex
        if ( graph[ path[pos-1] ][ path[0] ] == 1 )
           return true;
        else
          return false;
    }

    // Try different vertices as a next candidate in Hamiltonian Cycle.
    // We don't try for 0 as we included 0 as starting point in in hamCycle()
    for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian Cycle */
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;

            /* recur to construct rest of the path */
            if (hamCycleUtil (graph, path, pos+1) == true)
                return true;

            /* If adding vertex v doesn't lead to a solution,
               then remove it */
            path[pos] = -1;
        }
    }

    /* If no vertex can be added to Hamiltonian Cycle constructed so far,
       then return false */
    return false;
}
```

```
/* This function solves the Hamiltonian Cycle problem using Backtracking.
  It mainly uses hamCycleUtil() to solve the problem. It returns false
  if there is no Hamiltonian Cycle possible, otherwise return true and
  prints the path. Please note that there may be more than one solutions,
  this function prints one of the feasible solutions. */
bool hamCycle(bool graph[V][V])
{
    int *path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    /* Let us put vertex 0 as the first vertex in the path. If there is
       a Hamiltonian Cycle, then the path can be started from any point
       of the cycle as the graph is undirected */


    path[0] = 0;
    if ( hamCycleUtil(graph, path, 1) == false )
    {
        printf("\nSolution does not exist");
        return false;
    }

    printSolution(path);
    return true;
}

/* A utility function to print solution */
void printSolution(int path[])
{
    printf ("Solution Exists:"
            " Following is one Hamiltonian Cycle \n");
    for (int i = 0; i < V; i++)
        printf(" %d ", path[i]);

    // Let us print the first vertex again to show the complete cycle
    printf(" %d ", path[0]);
    printf("\n");
}

// driver program to test above function
int main()
{
   /* Let us create the following graph
      (0)--(1)--(2)
       |   / \   |
       | /   \ |
       |/     \|
      (3)-------(4)    */
   bool graph1[V][V] = {{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},
                       };

    // Print the solution
    hamCycle(graph1);

   /* Let us create the following graph
      (0)--(1)--(2)
       |  / \   |
       | /   \ |
       |/     \|
      (3)       (4)    */
    bool graph2[V][V] = {{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},
                       };

    // Print the solution
    hamCycle(graph2);

    return 0;
}
```

Output:

```
Solution Exists: Following is one Hamiltonian Cycle0 1 2
 4 3 0


Solution does not exist
```

EXPERIMENT NO: 10

Aim: Study NP Hard Graph , NP Hard Scheduling problem.

Objective: To Learn whether all the computational problems be solved by a computer.

Theory:

There are computational problems that can not be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer andprogram(SourceHaltingproblem).

Status of NP Complete problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

What are NP, P, NP-complete and NP-Hard problems?

P is set of problems that can be solved by a deterministic Turing machine in **P**olynomial time.

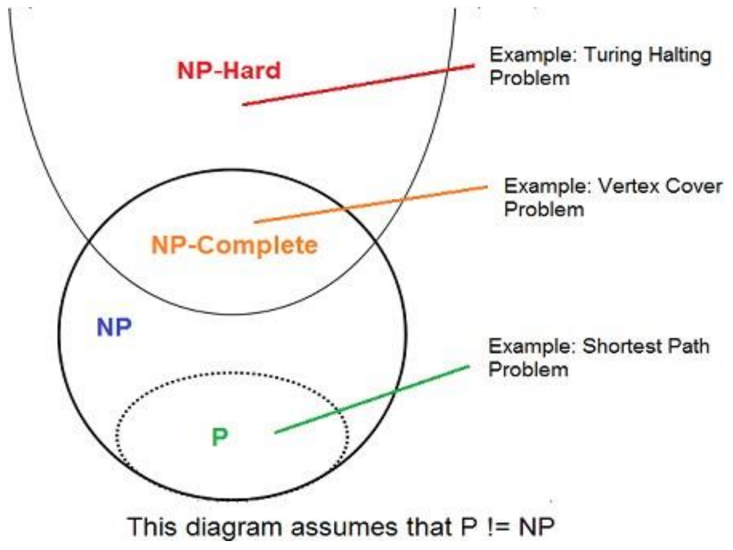
NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time). Informally, NP is set of decision problems which can be solved by a polynomial time via a “Lucky Algorithm”, a magical algorithm that always makes a right guess among the given set of choices (Source Ref 1).

NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:

1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).

2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is NP-Hard if it follows property 2 mentioned above, doesn’t need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.

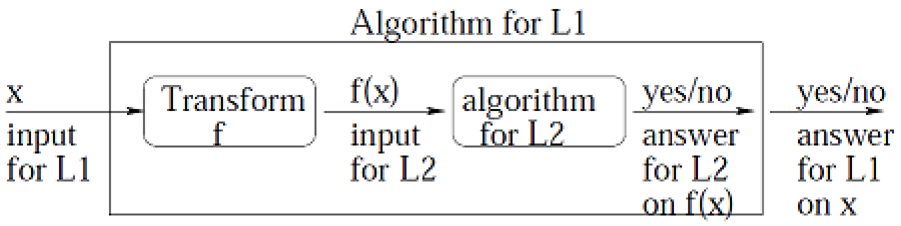

NP-Hard
Example: Turing Halting Problem
Example: Vertex Cover Problem
NP-Complete
NP
Example: Shortest Path Problem
P
This diagram assumes that P != NP

Decision vs Optimization Problems

NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number ofcalls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source Ref 2).For example, consider the vertex cover problem (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

What is Reduction?

Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether y belongs to L_2 or not. The idea is to find a transformation from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.



Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for Dijkstra’s algorithm to find shortest path, we can take log of allweights and use Dijkstra’s algorithm to find the minimum product path rather than writing afresh code for this new problem.

How to prove that a given problem is NP complete?

From the definition of NP-complete, it appears impossible to prove that a problem L is NP- Complete. By definition, it requires us to that show every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L. If polynomial time reduction is possible, we can provethat L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

What was the first problem proved as NP-Complete?

There must be some first NP-Complete problem proved by definition of NP-Complete problems. SAT (Boolean satisfiability problem) is the first NP-Complete problem proved by Cook (See CLRS book for proof).

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked towrite an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don’t know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem as NP-complete, youcan proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science manyscientists have been trying for years.