Summary I

SWE2016-44

Dynamic Programming

Dynamic Programming

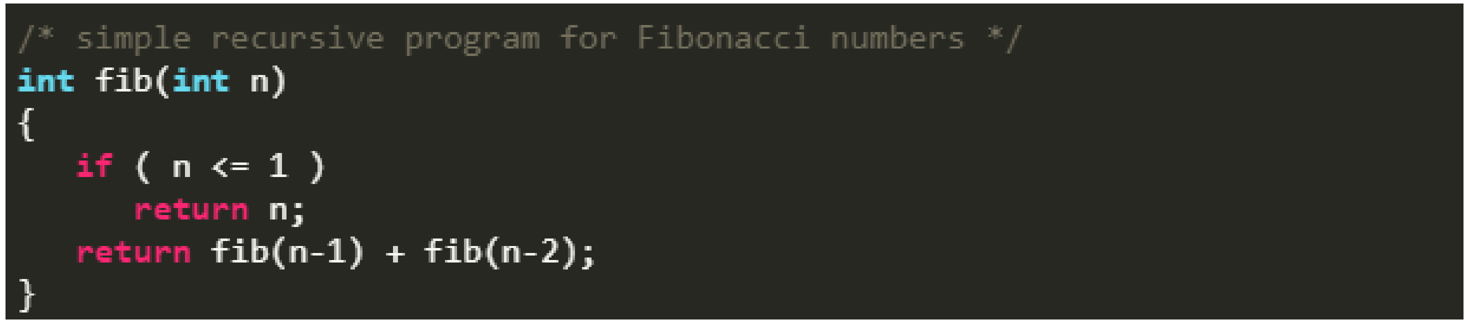
Definition

- Solves a given complex problem by breaking it into subproblems
- Stores the results of subproblems to avoid computing the same results again.

Properties

- Optimal Substructure
- Overlapping Subproblems

Optimal Substructure



```
/* simple recursive program for Fibonacci numbers */
int fib(int n)
{
   if ( n <= 1 )
      return n;
   return fib(n-1) + fib(n-2);
}
```

Optimal Substructure

A given problem is said to have the Optimal Substructure property if an optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

Optimal Substructure

For Example,

- **The Shortest Path problem has following Optimal Substructure property:**
  - **If a node x lies in the shortest path from source node u to destination node v then, the shortest path from u to v is the combination of shortest path from u to x and shortest path from x to v.**

- **All Pair Shortest Path**
  - **Floyd-Warshall**
  - **Bellman-Ford**

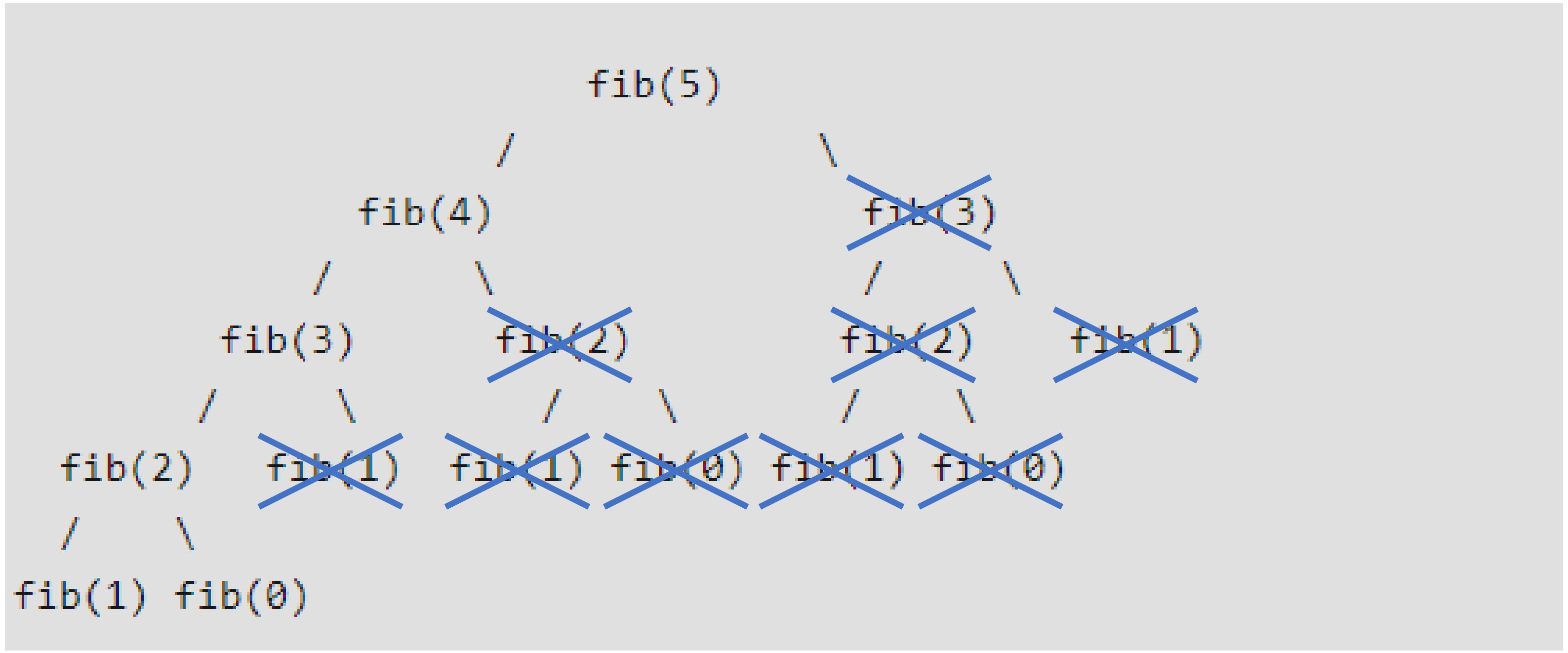
Optimal Substructure

On the other hand,

- The Longest Path problem doesn’t have the Optimal Substructure property.

- Here, by Longest Path we mean longest simple path (path without cycle) between two nodes.

Overlapping Subproblems



Dynamic Programming

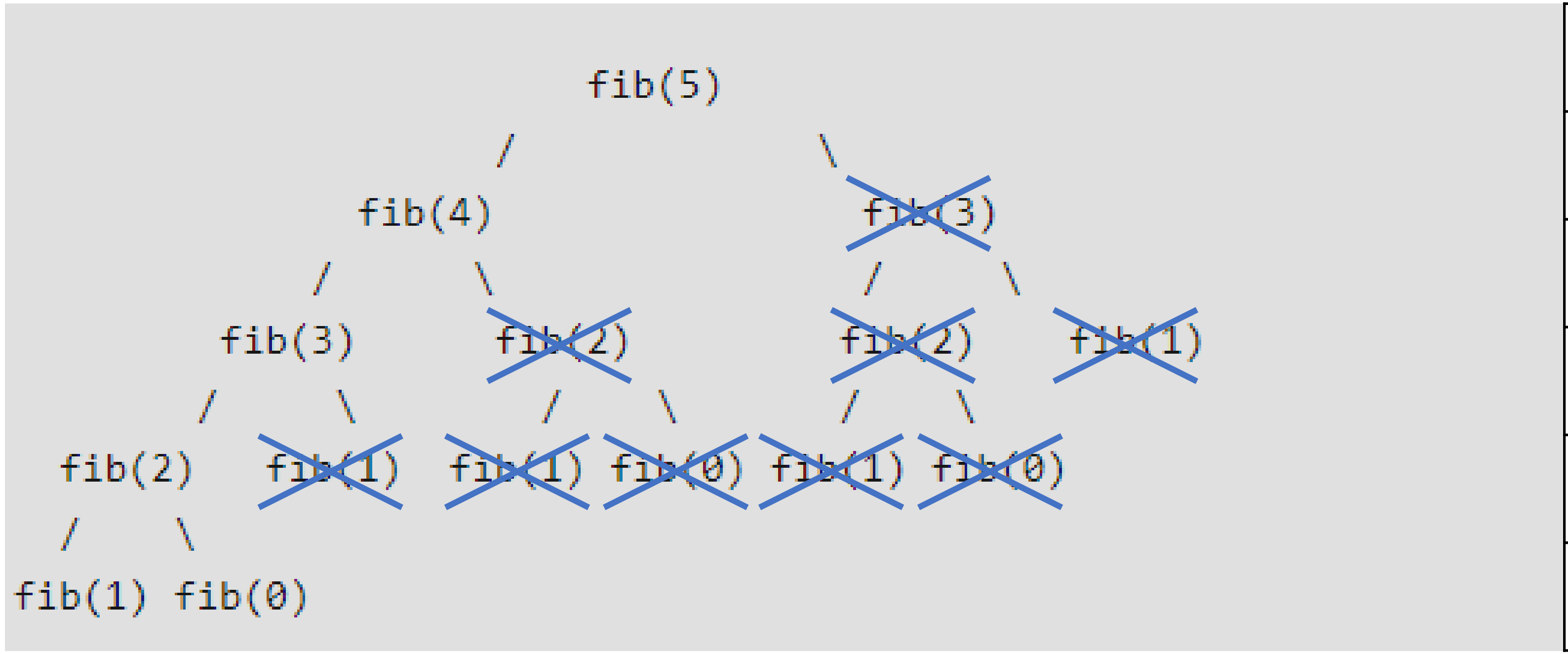
Memoization

- **Initialize a lookup array/table with all its elements as NIL**
- **Call the recursive function f(n) to solve for ‘n’ using memoization**

Tabulation

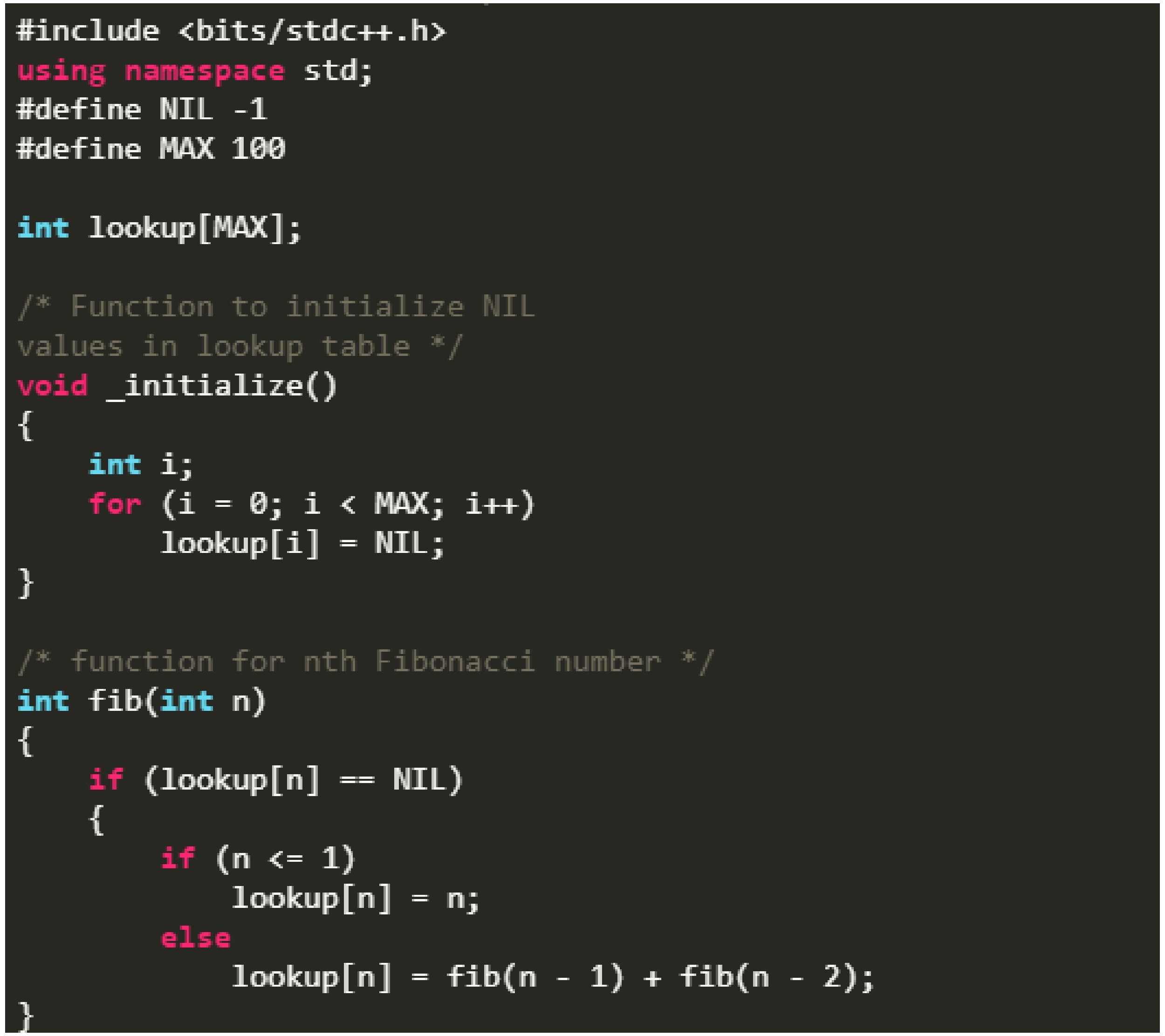
- **Build the lookup table in bottom up fashion**
- **After the table is built, simply return table[n]**

Memoization



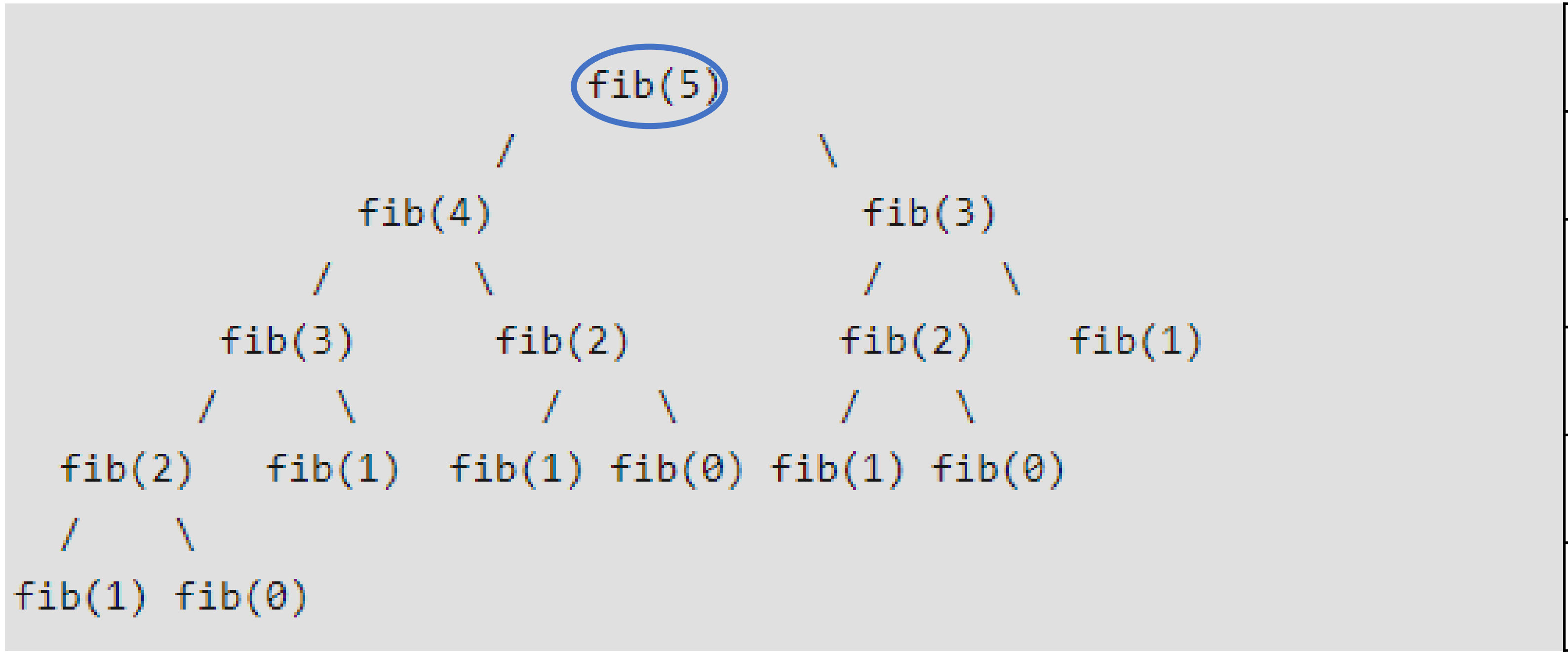
table[5]	5
table[4]	3
table[3]	2
table[2]	1
table[1]	1
table[0]	0

Memoization



```
#include <bits/stdc++.h>
using namespace std;
#define NIL -1
#define MAX 100

int lookup[MAX];

/* Function to initialize NIL
values in lookup table */
void _initialize()
{
    int i;
    for (i = 0; i < MAX; i++)
        lookup[i] = NIL;
}

/* function for nth Fibonacci number */
int fib(int n)
{
    if (lookup[n] == NIL)
    {
        if (n <= 1)
            lookup[n] = n;
        else
            lookup[n] = fib(n - 1) + fib(n - 2);
}
```

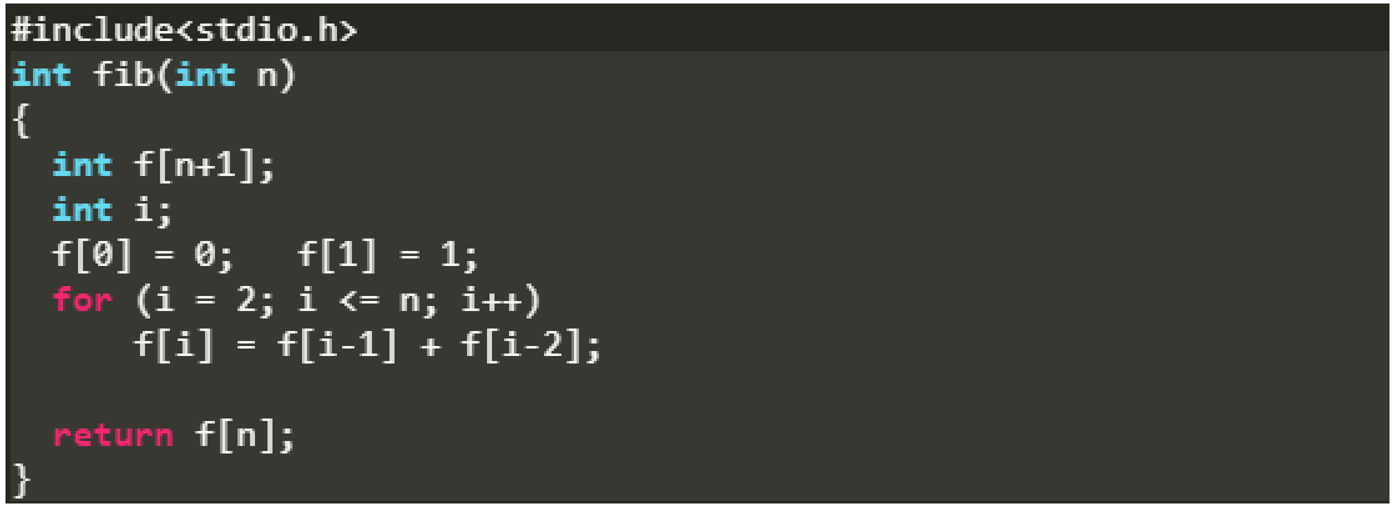
Tabulation



```
                         fib(5)
                     /             \
               fib(4)                fib(3)
             /      \                /     \
         fib(3)      fib(2)        fib(2)    fib(1)
        /     \        /    \       /    \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

table[5]	5
table[4]	3
table[3]	2
table[2]	1
table[1]	1
table[0]	0

Tabulation



```
#include<stdio.h>
int fib(int n)
{
  int f[n+1];
  int i;
  f[0] = 0;   f[1] = 1;
  for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];

  return f[n];
}
```

Tabulation or Memoization

- **Tabulation**
  - **Works in bottom up fashion**
  - **Avoids multiple lookups, thus, saves function call overhead time**

- **Memoization**
  - **Works in top down fashion**
  - **Sometimes, avoids computing solutions to subproblems that are not needed, e.g., Longest Common Subsequence**
  - **Sometimes, more intuitive to write, e.g., Matrix Chain Multiplication**

How to solve DP?

Steps to solve a DP

1. **Identify if it is a DP problem**
2. **Decide a state expression with least parameters**
3. **Formulate state relationship**
4. **Do tabulation (or add memoization)**

How to solve DP?

Step 1: How to classify a problem as a DP Problem?

- **Typically, 1) all the problems that require to maximize or minimize certain quantity or 2) counting problems that say to count the arrangements under certain condition or certain probability problems can be solved by using DP.**

- **All DP problems satisfy the overlapping subproblems property and most of the classic dynamic problems also satisfy the optimal substructure property.**

How to solve DP?

Step 2: Deciding the state

- **DP problems are all about <u>state</u> and their <u>transition</u>. This is the most basic step which must be done very carefully because the state transition depends on the choice of state definition you make.**

- **State: A state can be defined as the set of parameters that can uniquely identify a certain position or standing in the given problem. This set of parameters should be as small as possible to reduce state space.**

How to solve DP?

Step 2: Deciding the state

```
                         fib(5)
                     /            \
               fib(4)                   fib(3)
             /      \                 /      \
         fib(3)      fib(2)         fib(2)     fib(1)
        /     \       /    \       /    \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

table[5]	5
table[4]	3
table[3]	2
table[2]	1
table[1]	1
table[0]	0

How to solve a Dynamic Programming Problem?

Step 3: Formulating a relation among the states

Given 3 numbers {1, 3, 5}, we wish to know the result of the state (n = 7). See, we can only add 1, 3 and 5.

Now we can get a sum total of 7 by the following 3 ways:

1) **Adding 1 to all possible combinations of state (n = 6)**
2) **Adding 3 to all possible combinations of state (n = 4)**
3) **Adding 5 to all possible combinations of state(n = 2)**

How to solve a Dynamic Programming Problem?

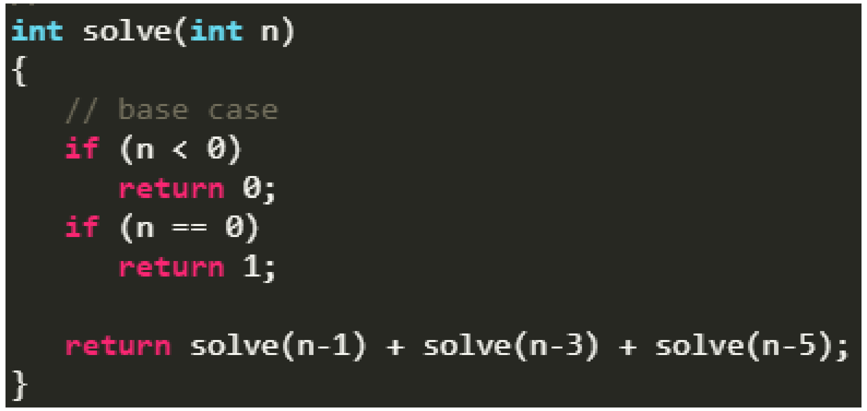
Step 3: Formulating a relation among the states

We can say that result for
state(7) = state (6) + state (4) + state (2)

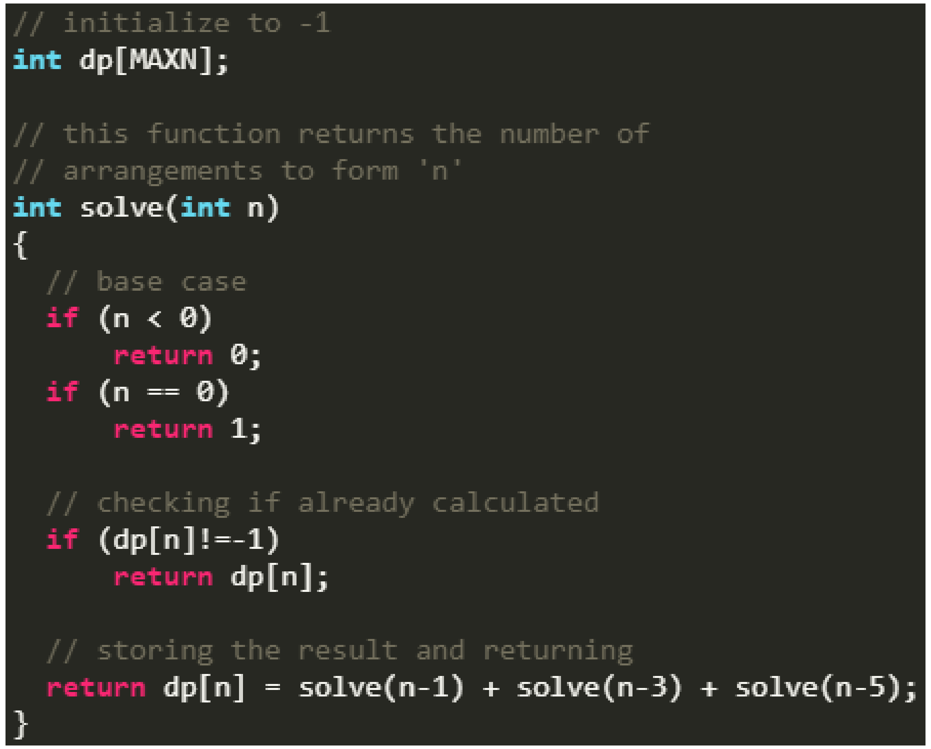
In general,
state(n) = state(n-1) + state(n-3) + state(n-5)

How to solve a Dynamic Programming Problem?

Step 4: Adding memoization or tabulation for the state



```
int solve(int n)
{
   // base case
   if (n < 0)
      return 0;
   if (n == 0)
      return 1;

   return solve(n-1) + solve(n-3) + solve(n-5);
}
```



```
// initialize to -1
int dp[MAXN];

// this function returns the number of
// arrangements to form 'n'
int solve(int n)
{
  // base case
  if (n < 0)
      return 0;
  if (n == 0)
      return 1;

  // checking if already calculated
  if (dp[n]!=-1)
      return dp[n];

  // storing the result and returning
  return dp[n] = solve(n-1) + solve(n-3) + solve(n-5);
}
```

Adding memoization

Example (Longest Increasing Subsequence)

Find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

For example,

Given sequence LIS = {10, 22, 9, 33, 21, 50, 41, 60}

Subsequences: {10}, {10, 22}, {10, 9, 33}, {9, 21, 60}, {50, 60}, …

Increasing Subsequences: {10}, {9, 33, 41}, {33, 41, 60}, {41}, …

Longest Increasing Subsequences: {10, 22, 33, 50, 60} or {10, 22, 33, 41, 60}

So, Length of LIS = 5

Example (Longest Increasing Subsequence)

Let arr[0..n-1] be the input array; and L(i) be the length of the LIS ending at index i such that arr[i] is the last element of the LIS.

L(i) can be recursively written as:

L(i) = 1 + max(L(j)) where 0 < j < i and arr[j] < arr[i]; or
L(i) = 1, if no such j exists.

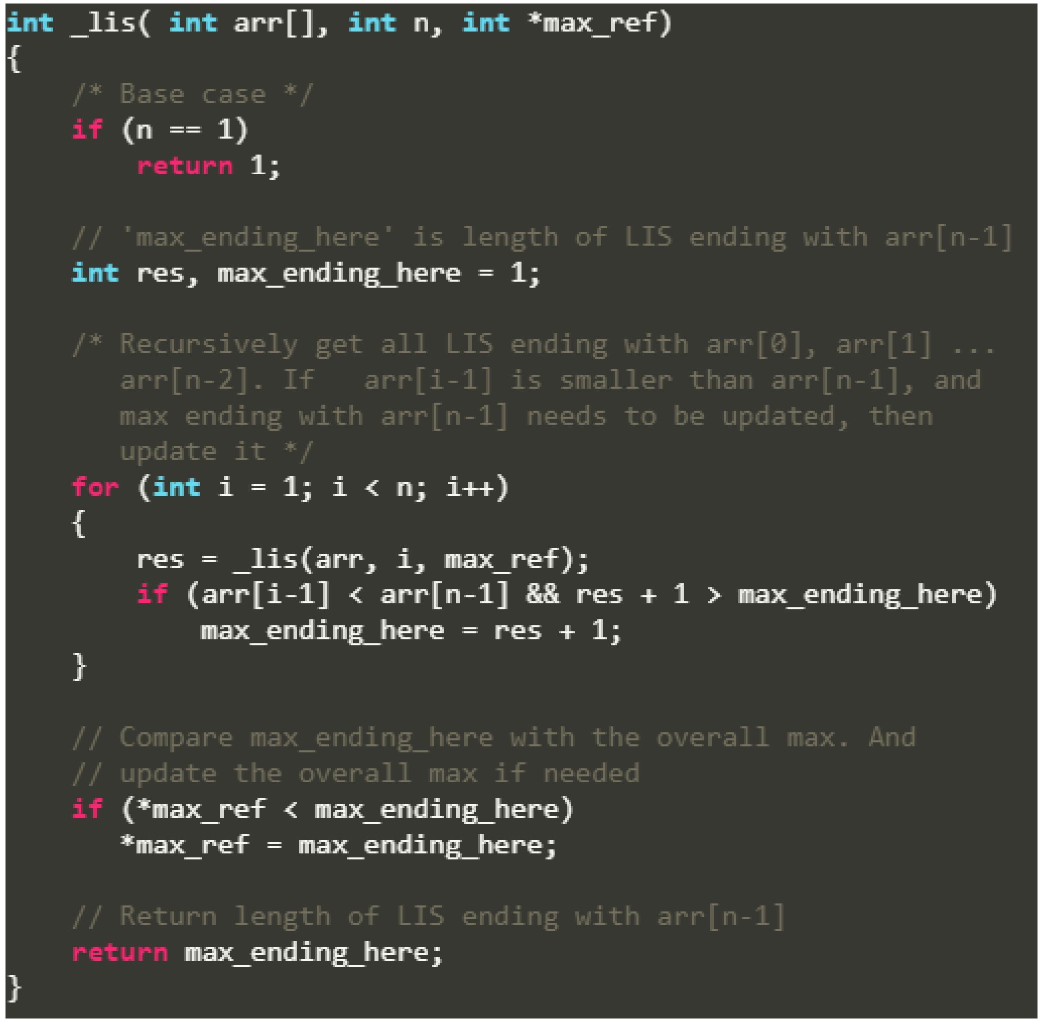
To find the LIS for a given array, return max(L(i)) where 0<i<n.

Example (Longest Increasing Subsequence)

Final values:

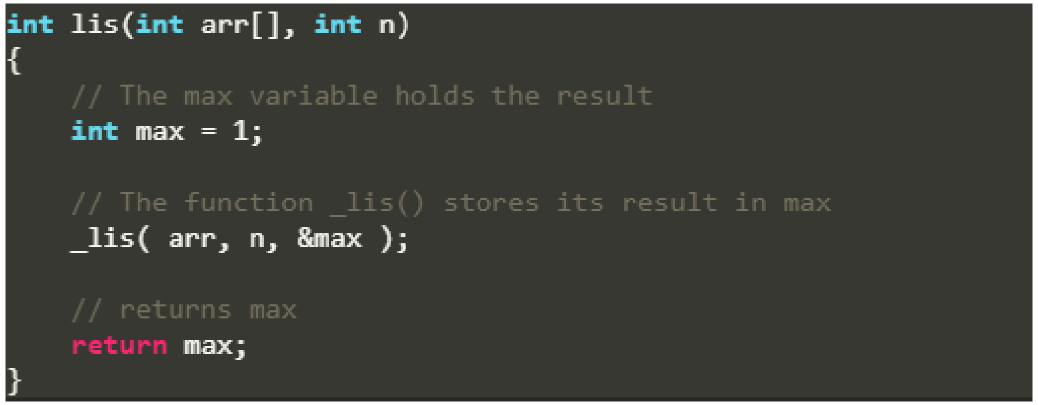
iterator							j	i
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	4	5

Example (Longest Increasing Subsequence)



```
int _lis( int arr[], int n, int *max_ref)
{
    /* Base case */
    if (n == 1)
        return 1;

    // 'max_ending_here' is length of LIS ending with arr[n-1]
    int res, max_ending_here = 1;

    /* Recursively get all LIS ending with arr[0], arr[1] ...
       arr[n-2]. If   arr[i-1] is smaller than arr[n-1], and
       max ending with arr[n-1] needs to be updated, then
       update it */
    for (int i = 1; i < n; i++)
    {
        res = _lis(arr, i, max_ref);
        if (arr[i-1] < arr[n-1] && res + 1 > max_ending_here)
            max_ending_here = res + 1;
    }

    // Compare max_ending_here with the overall max. And
    // update the overall max if needed
    if (*max_ref < max_ending_here)
       *max_ref = max_ending_here;

    // Return length of LIS ending with arr[n-1]
    return max_ending_here;
}
```

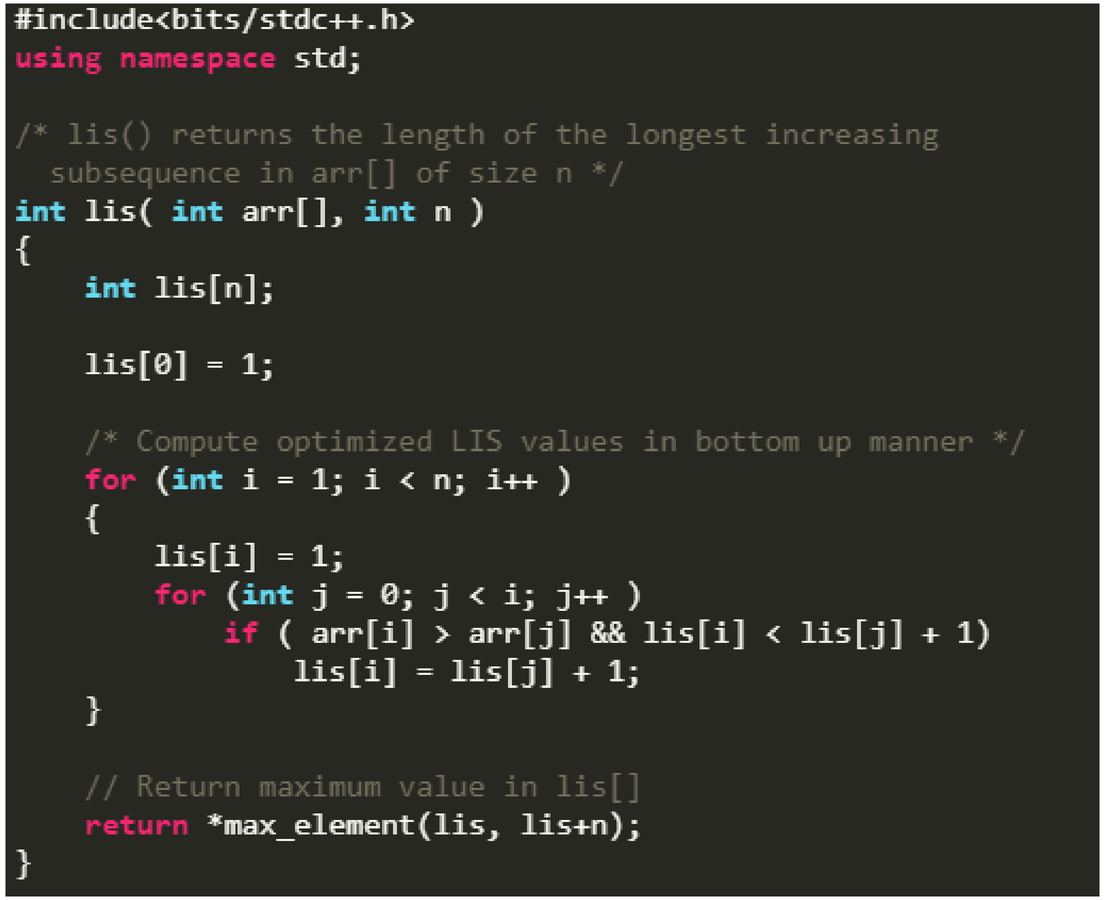
Recursive Implementation



```
int lis(int arr[], int n)
{
    // The max variable holds the result
    int max = 1;

    // The function _lis() stores its result in max
    _lis( arr, n, &max );

    // returns max
    return max;
}
```

→ Time Complextity: O(2^n)

Example (Longest Increasing Subsequence)



```
#include<bits/stdc++.h>
using namespace std;

/* lis() returns the length of the longest increasing
  subsequence in arr[] of size n */
int lis( int arr[], int n )
{
    int lis[n];

    lis[0] = 1;

    /* Compute optimized LIS values in bottom up manner */
    for (int i = 1; i < n; i++ )
    {
        lis[i] = 1;
        for (int j = 0; j < i; j++ )
            if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
                lis[i] = lis[j] + 1;
    }

    // Return maximum value in lis[]
    return *max_element(lis, lis+n);
}
```

Dynamic Programming

Tabulation

→ Time Complextity: O(n^2)

Greedy Algorithm

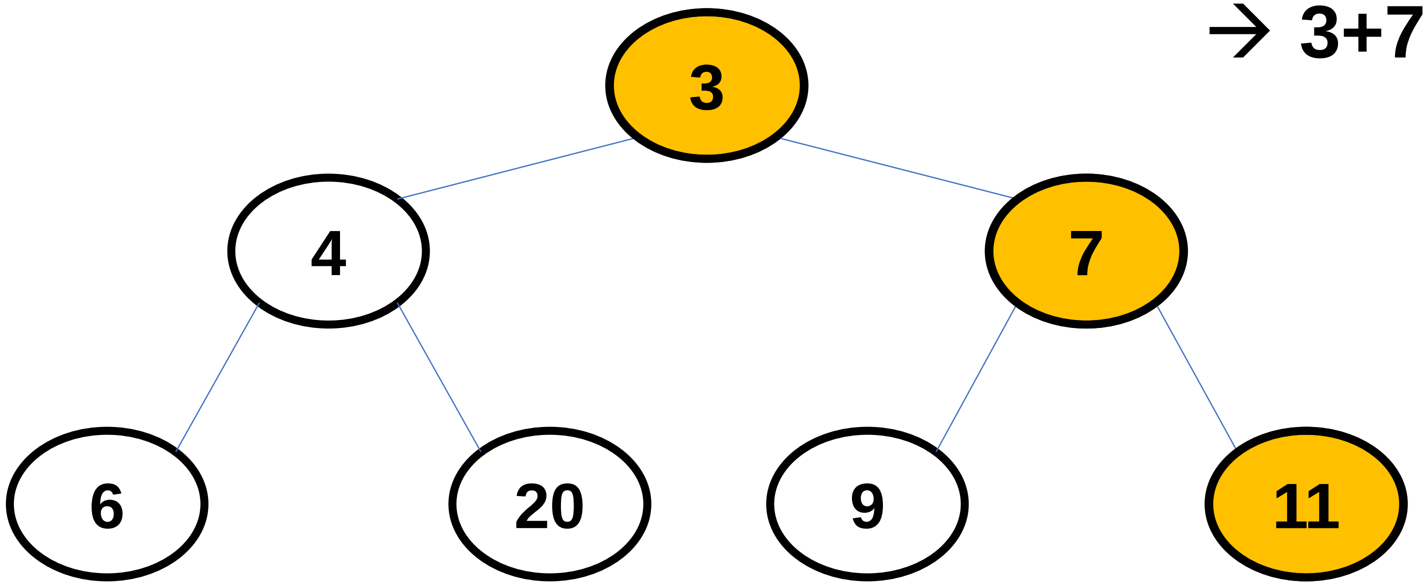
Introduction

Greedy Algorithm – an algorithmic paradigm that follows the problem solving approach of making the locally optimal choice at each stage with the hope of finding a global optimum.

Pros – simple, easy to implement, run fast

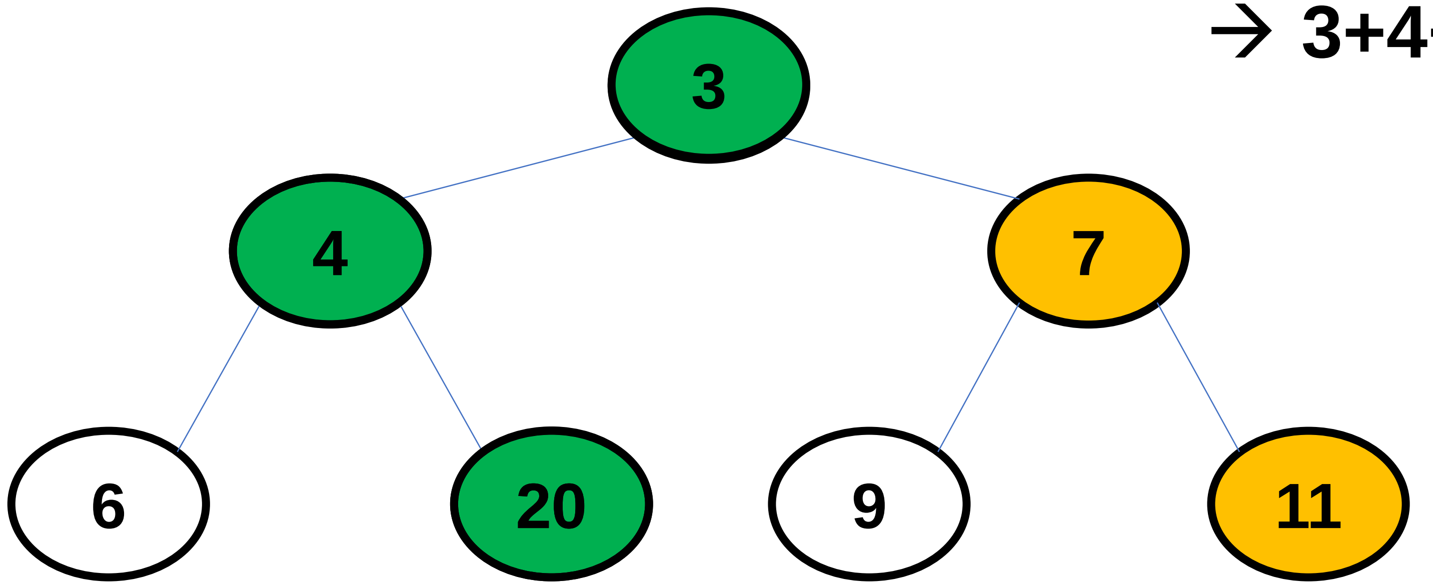
Cons – Very often they don’t provide a globally solution

Problem



Greedy Answer:
→ 3+7+11=21

Problem



Actual Answer:
→ 3+4+20=27

When to use?

Problems on which greedy approach work has two properties.

1. **Greedy – choice property: A global optimum can be arrived at by selecting a local minimum**

2. **Optimal substructure: An optimum solution to the problem contains an optimal solution to subproblem**

Activity Selection Problem

Problem: 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.

Example:

Activity	**A1**	**A2**	**A3**
Start	12	10	20
Finish	25	20	30

A2 → A3 **Total = 2 activities**

Activity Selection Problem

<u>Greedy Approach</u>

1. **Sort the activities according to their finishing time.**

Activity	A1	A2	A3	A4	A5	A6
Start	0	3	1	5	5	8
Finish	6	4	2	9	7	9

Sorted

Activity	A3	A2	A1	A5	A6	A4
Start	1	3	0	5	8	5
Finish	2	4	6	7	9	9

Activity Selection Problem

Greedy Approach

2. **Select the first activity from the sorted array and print it.**

Activity	**A3**	**A2**	**A1**	**A5**	**A6**	**A4**
Start	1	3	0	5	8	5
Finish	2	4	6	7	9	9

Answer = A3

Activity Selection Problem

Greedy Approach

3. **If the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.**

Activity	**A3**	**A2**	**A1**	**A5**	**A6**	**A4**
Start	1	3	0	5	8	5
Finish	2	4	6	7	9	9

Answer = A3 $\rightarrow$ A2 $\rightarrow$ A5 $\rightarrow$ A6 **Total = 4 activities**

Huffman Coding

- It is a loseless data compression
- We assign variable-length codes to input characters, length of which depends on frequency of characters.
- The variable-length codes assigned to input characters are Prefix Codes.

{0, 11}

Prefix Code

{0, 1, 11}

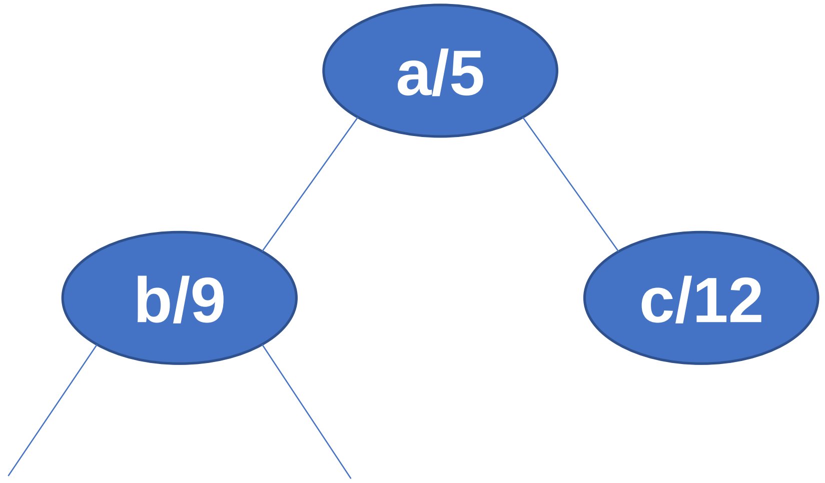
Non Prefix Code

Huffman Coding

Huffman Tree

1. **Create a leaf node for each unique character and build a min heap of all leaf nodes.**

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45



Huffman Coding

Huffman Tree

2. Extract two nodes with the minimum frequency from the min heap.

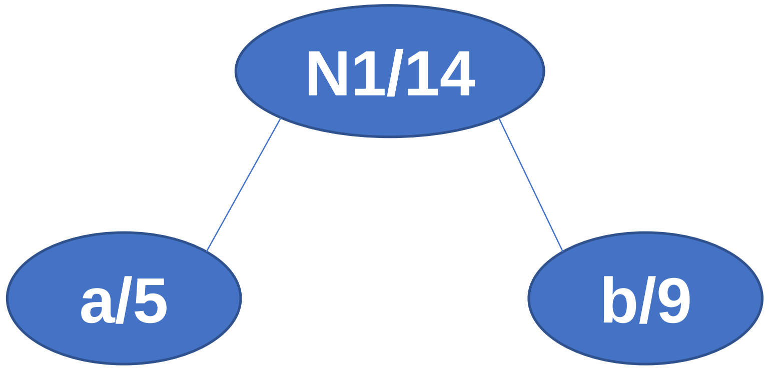
Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Huffman Coding

Huffman Tree

3. **Create a new interval 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.**

Character	Frequency
N1	14
c	12
d	13
e	16
f	45



Huffman Coding

Huffman Tree

4. **Repeat Steps 2 and 3 until the heap contains only one node. The remaining tree is the root node and the tree is complete.**

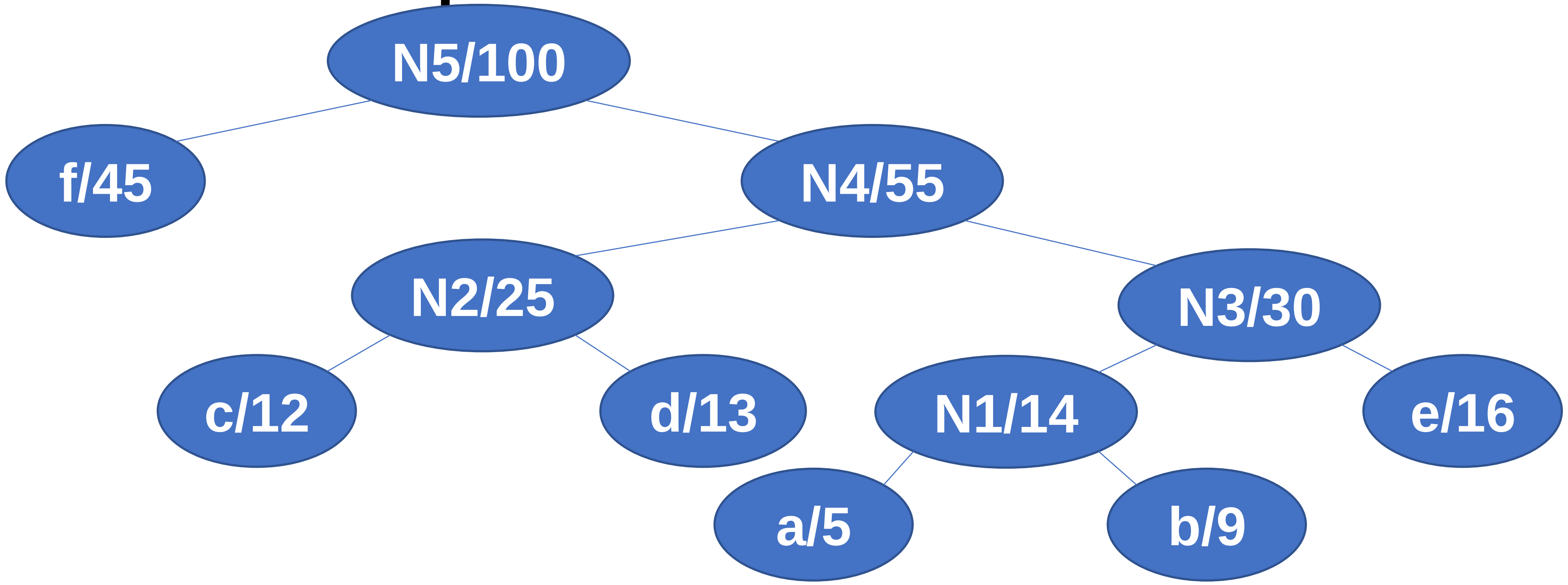
Character	Frequency
N1	14
c	12
d	13
e	16
f	45

Huffman Coding

Huffman Tree

Character	Frequency
N5	100

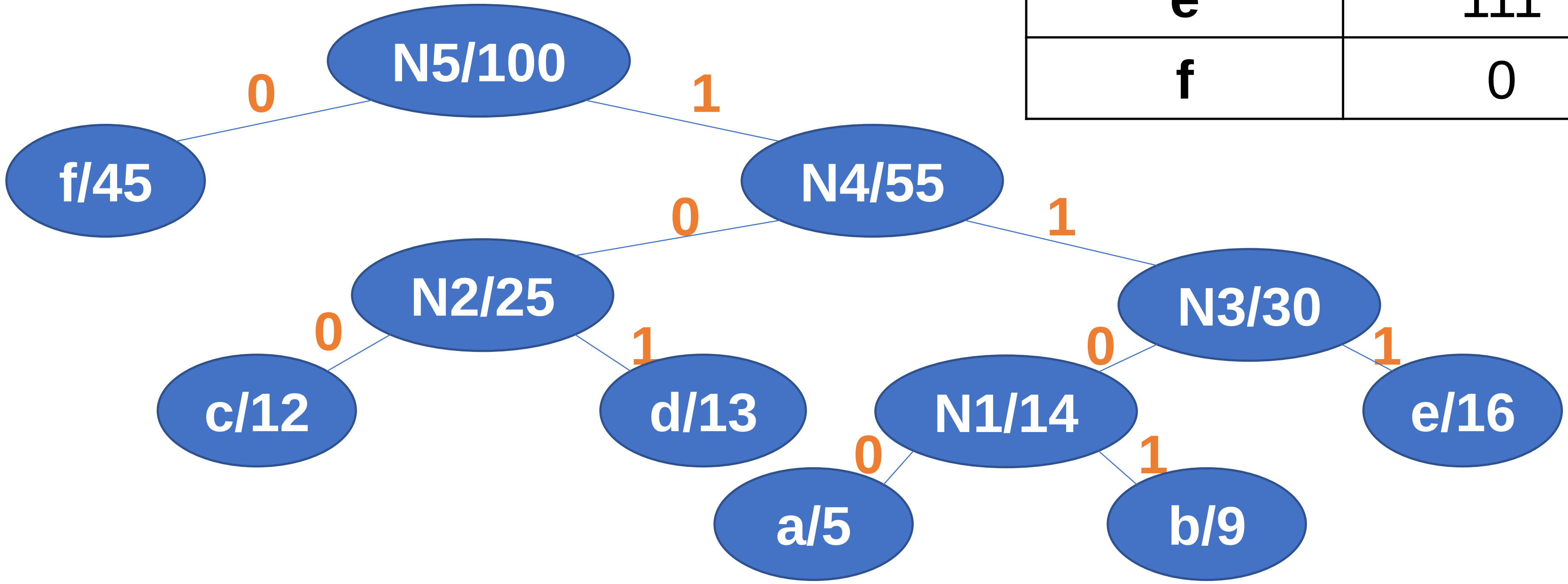
4. **Repeat Steps 2 and 3 until the heap contains only one node. The remaining tree is the root node and the tree is complete.**



Huffman Coding

Huffman Tree

Character	Code
a	1100
b	1101
c	100
d	101
e	111
f	0



Kruskal’s Algorithm

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

2. 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.

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

Greedy Algorithm vs Dynamic Programming

- **Greedy algorithm: Greedy algorithm is one which finds the feasible solution at every stage with the hope of finding global optimum solution.**

- **Dynamic Programming: Dynamic programming is one which breaks up the problem into series of overlapping sub-problems.**

Greedy Algorithm vs Dynamic Programming

1. Greedy algorithm never reconsiders its choices whereas Dynamic programming may consider the previous state.

2. Greedy algorithm have a local choice of the sub-problems whereas Dynamic programming would solve the all sub-problems and then select one that would lead to an optimal solution.

3. Greedy algorithm take decision in one time whereas Dynamic programming take decision at every stage.

Greedy Algorithm vs Dynamic Programming

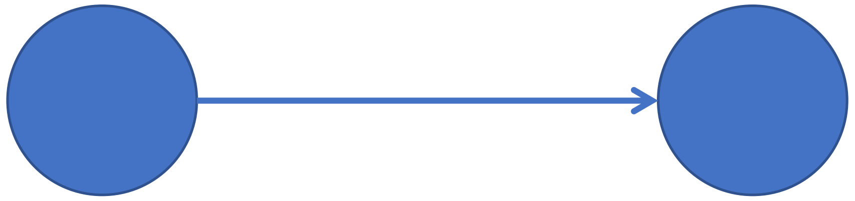
4. Greedy algorithm work based on choice property whereas Dynamic programming work based on principle of optimality.

5. Greedy algorithm follows the top-down strategy whereas Dynamic programming follows the bottom-up strategy.

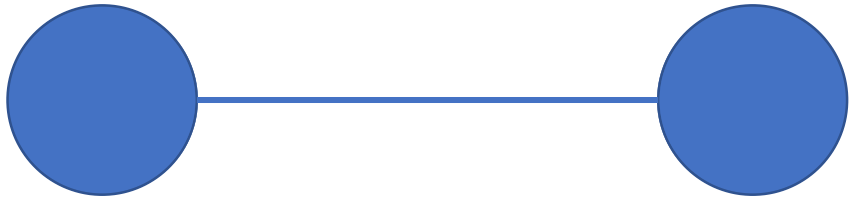
Graph Algorithm

Introduction

Directed Graph (di-graph): have pair of ordered vertices (u,v)



Un-Directed Graph: have pair of unordered vertices, (u,v) and (v,u) are same



Graph Representation

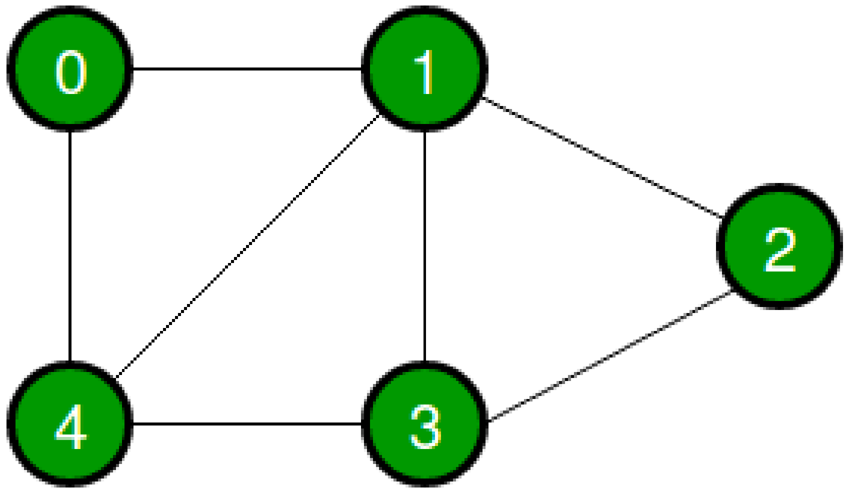
There are generally two ways to represent a graph data structure

- **Adjacency Matrix**
- **Adjacency List**

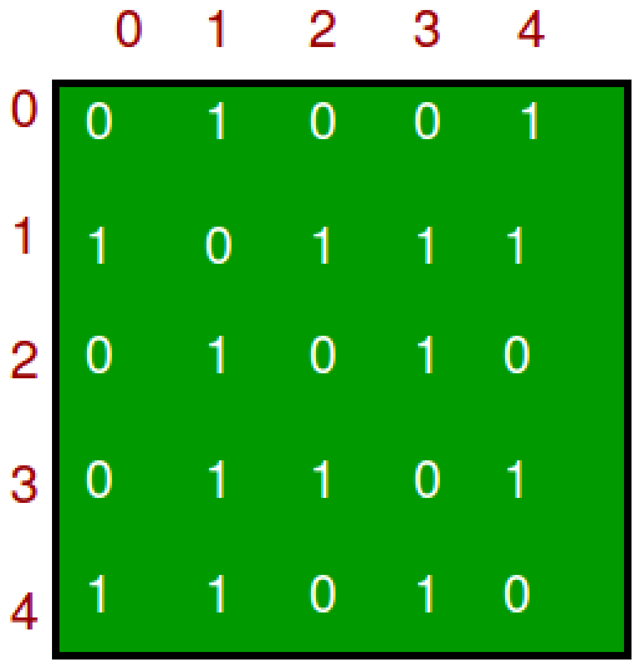
Graph Representation

Adjacency Matrix: Represents graph with ‘V’ nodes into an VxV 0-1 matrix where A_{ij}=1 means that vertex ‘i’ and ‘j’ are connected.

Example



Graph



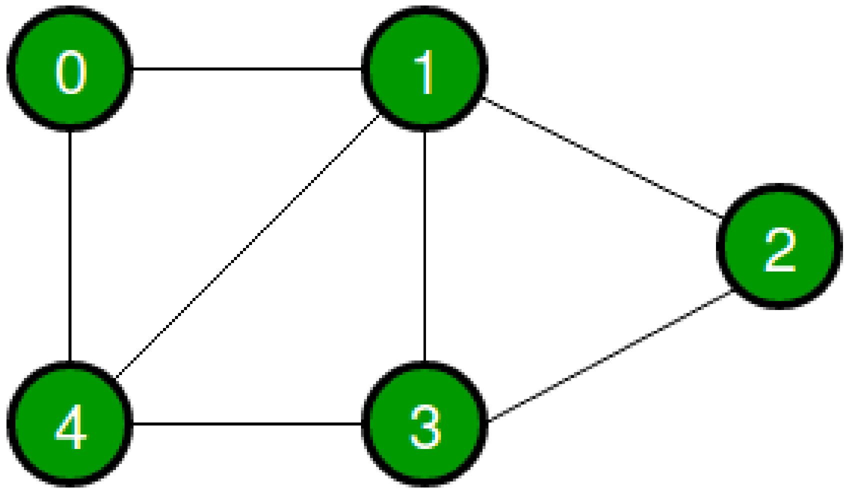
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency Matrix

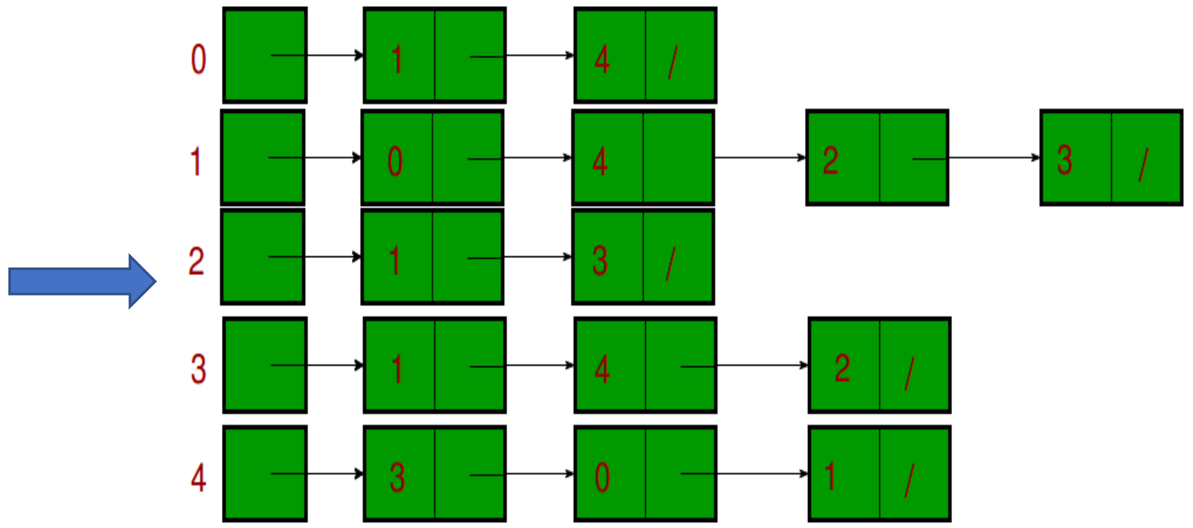
Graph Representation

Adjacency List: An array of linked lists is used. Size of the array is equal to number of vertices and each entry of array corresponds to a linked list of vertices adjacent to the index

Example



Graph



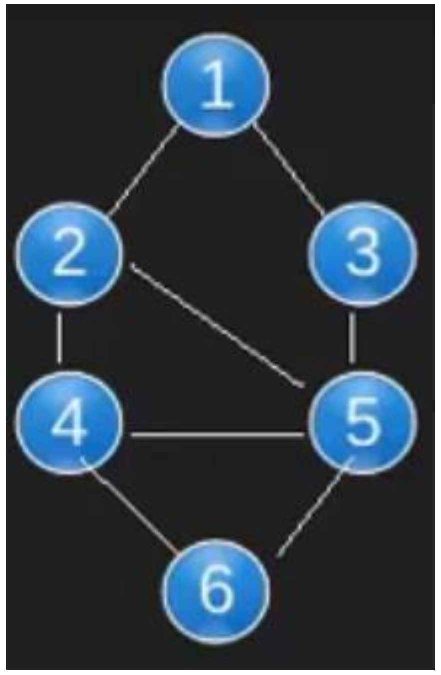
Adjacency List

Breadth First Search

Idea: Traverse nodes in layers

Problem: Since we have cycles, each node will be visited infinite times.

Solution: Use a Boolean visited array.



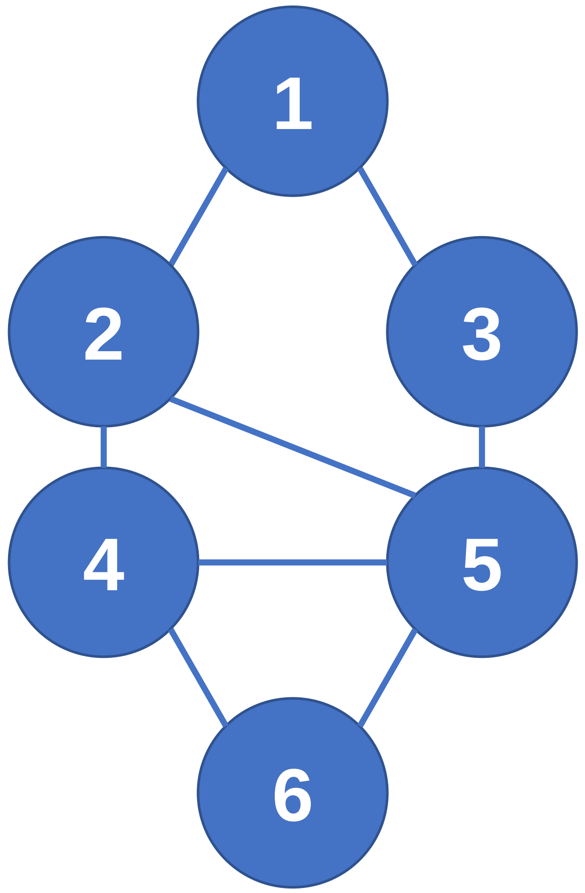
L1

L2

L3

L4

```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

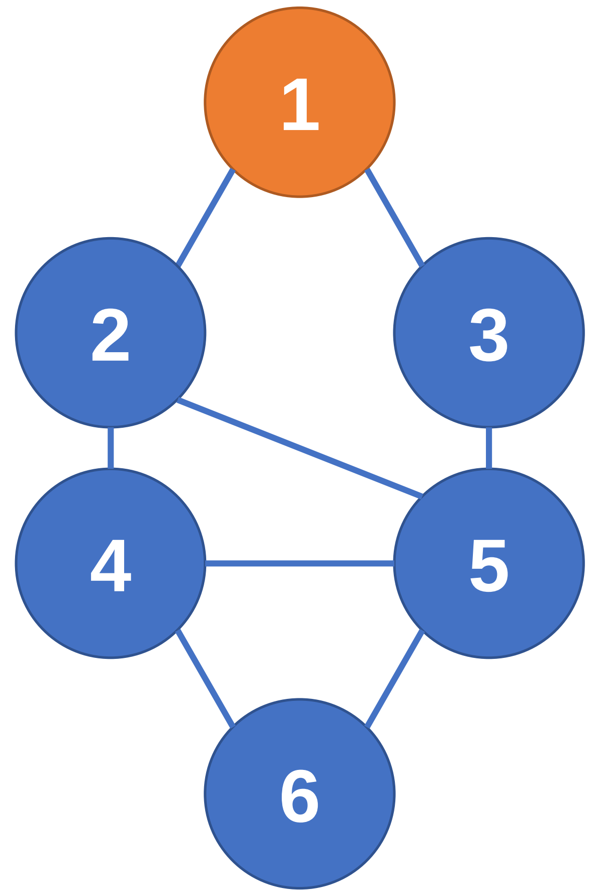


	1	2	3	4	5	6
Visited :	0	0	0	0	0	0

Queue :

Print :

```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

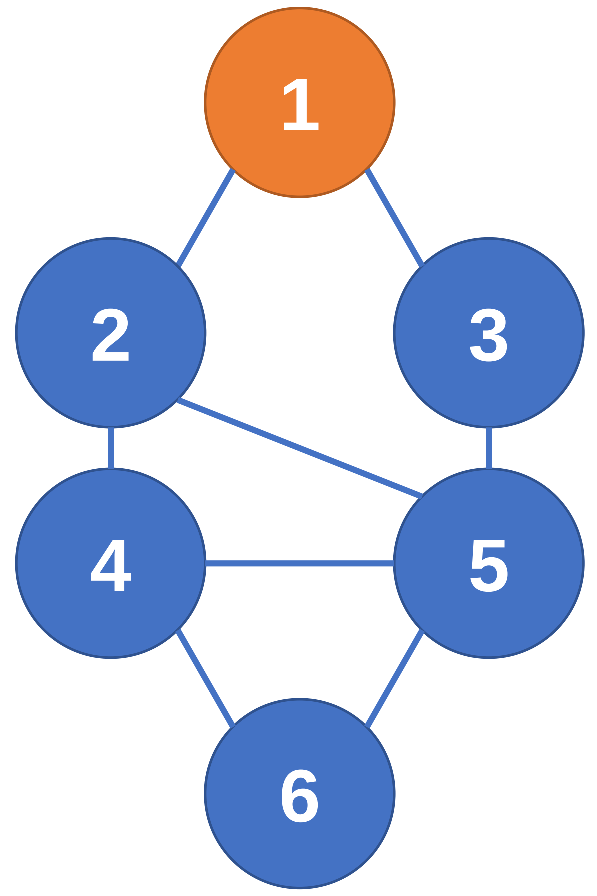


	1	2	3	4	5	6
Visited :	1	0	0	0	0	0

Queue : 1

Print : 1

```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

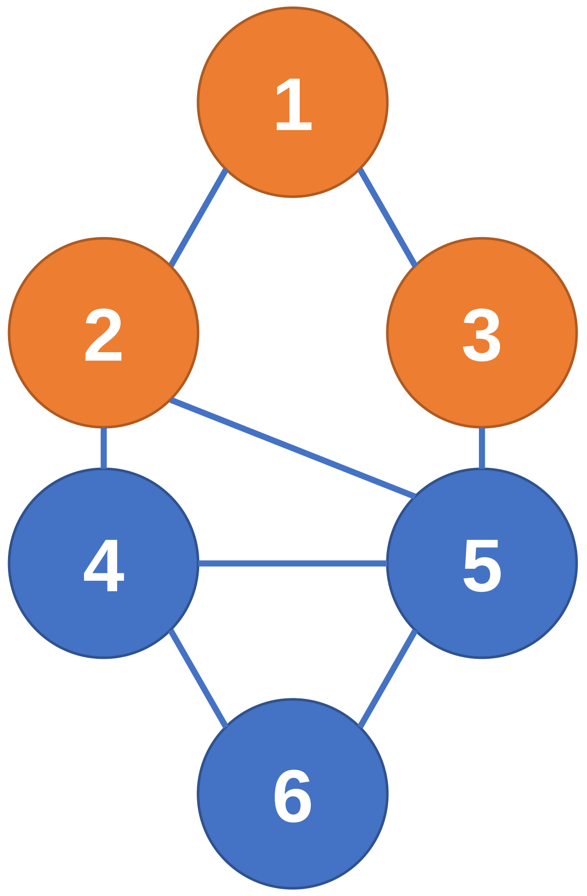


	1	2	3	4	5	6
Visited :	1	0	0	0	0	0

Queue :

Print : 1

```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

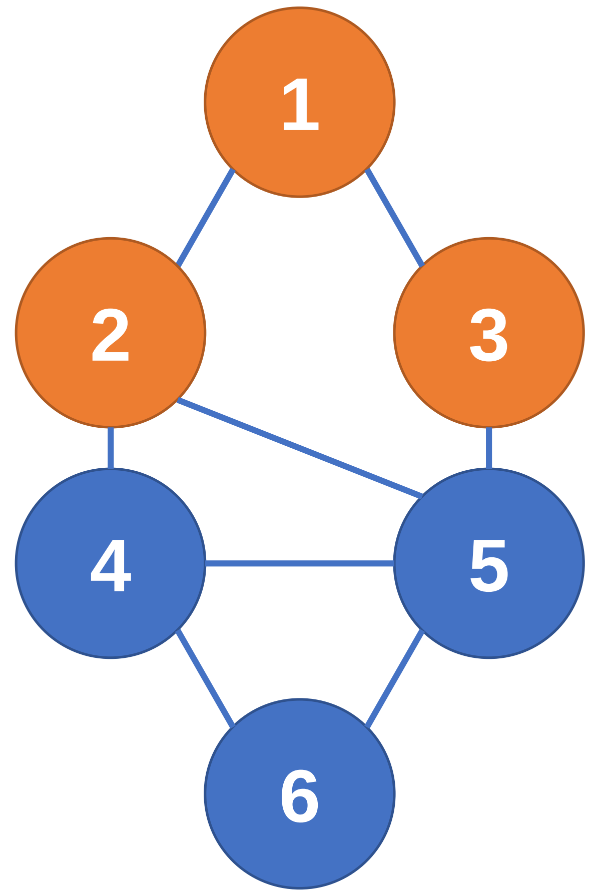


	1	2	3	4	5	6
Visited :	1	1	1	0	0	0

Queue : 2 3

Print : 1

```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

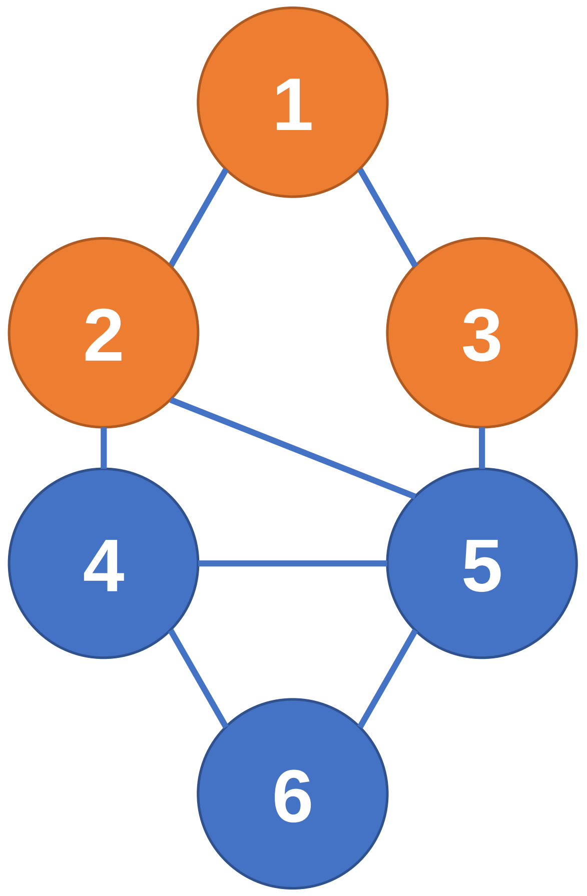


	1	2	3	4	5	6
Visited :	1	1	1	0	0	0

Queue : 2 3

Print : 1 2

```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

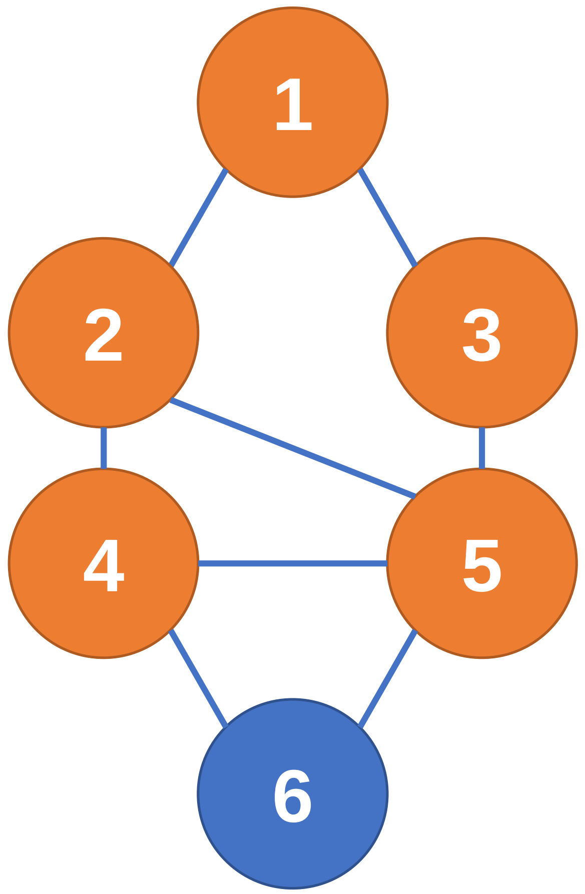


	1	2	3	4	5	6
Visited :	1	1	1	0	0	0

Queue : 3

Print : 1 2

```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

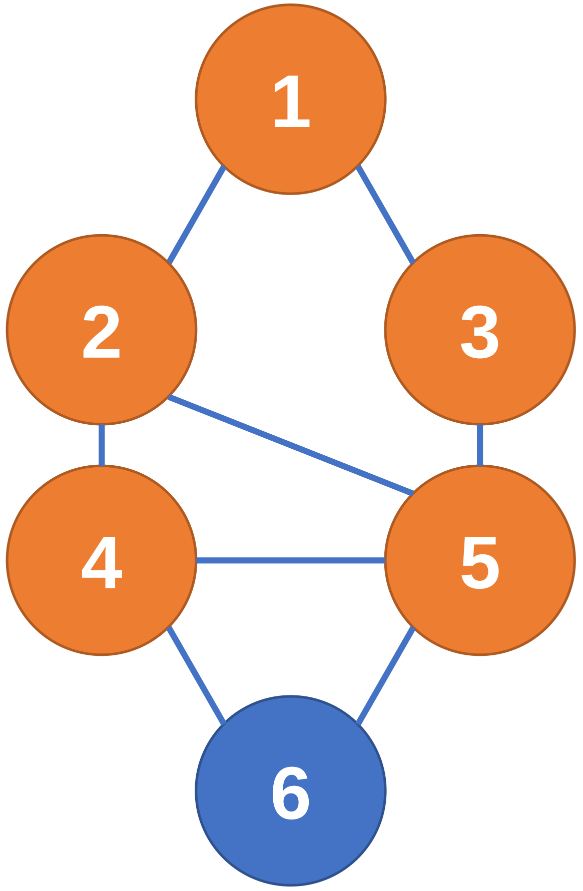


	1	2	3	4	5	6
Visited :	1	1	1	1	1	0

Queue : 3 4 5

Print : 1 2

```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

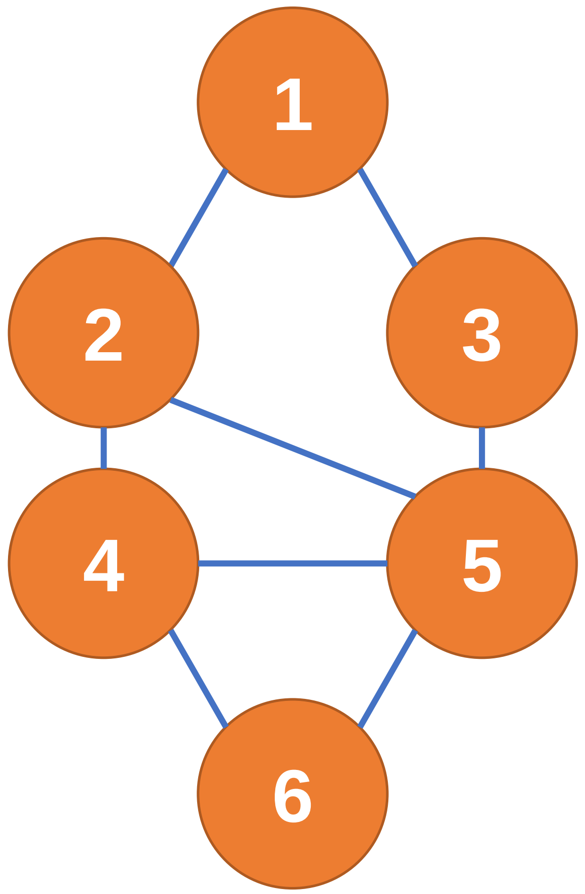


	1	2	3	4	5	6
Visited :	1	1	1	1	1	0

Queue : 4 5

Print : 1 2 3

```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

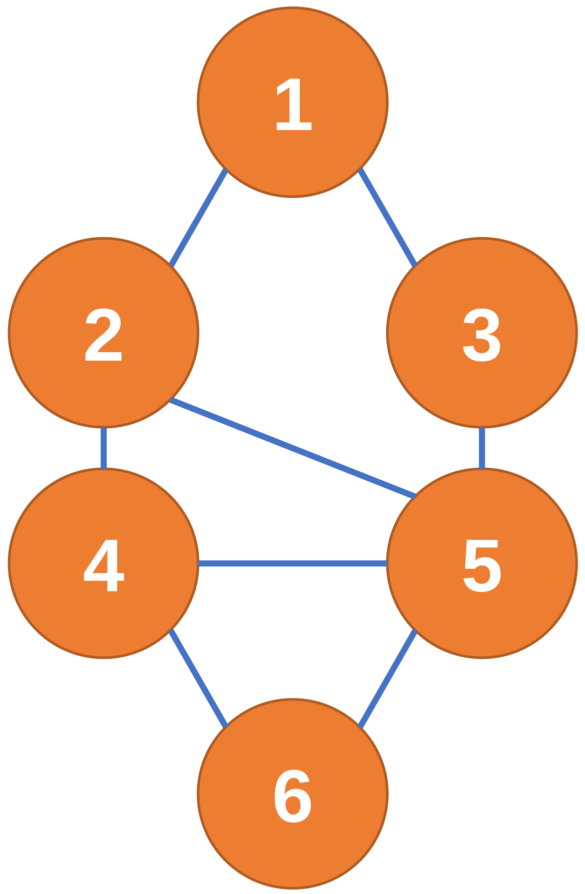


	1	2	3	4	5	6
Visited :	1	1	1	1	1	1

Queue : 5 6

Print : 1 2 3 4

```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

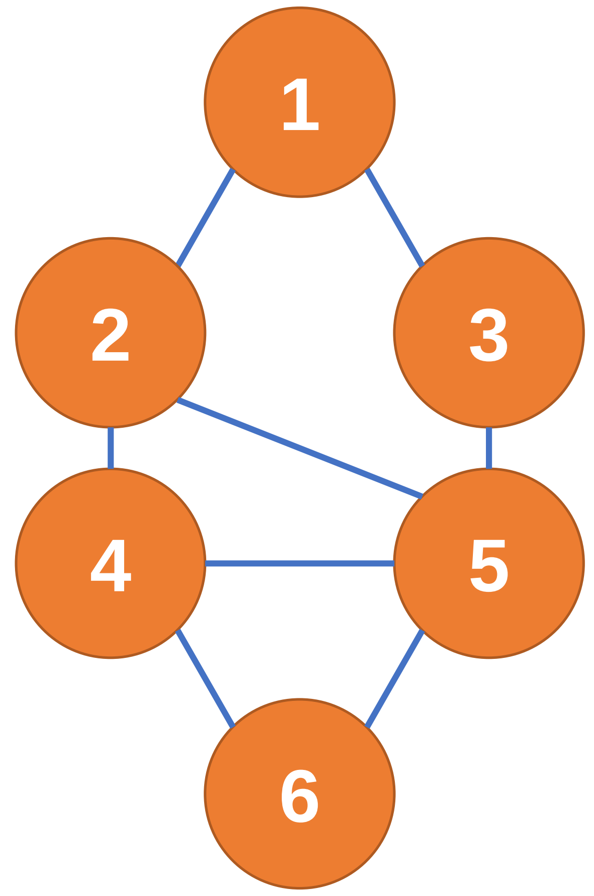


	1	2	3	4	5	6
Visited :	1	1	1	1	1	1

Queue : 6

Print : 1 2 3 4 5

```
void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```



	1	2	3	4	5	6
Visited :	1	1	1	1	1	1

Queue :

Print : 1 2 3 4 5 6

Complexity

Time Complexity: O(V+E)

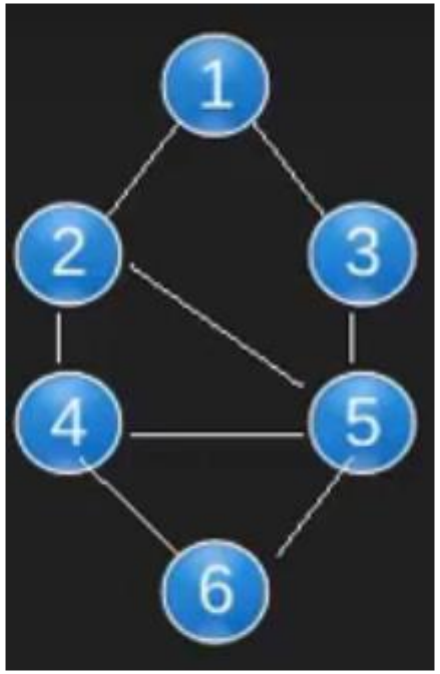
V: Vertices
E: Edges

Depth First Search

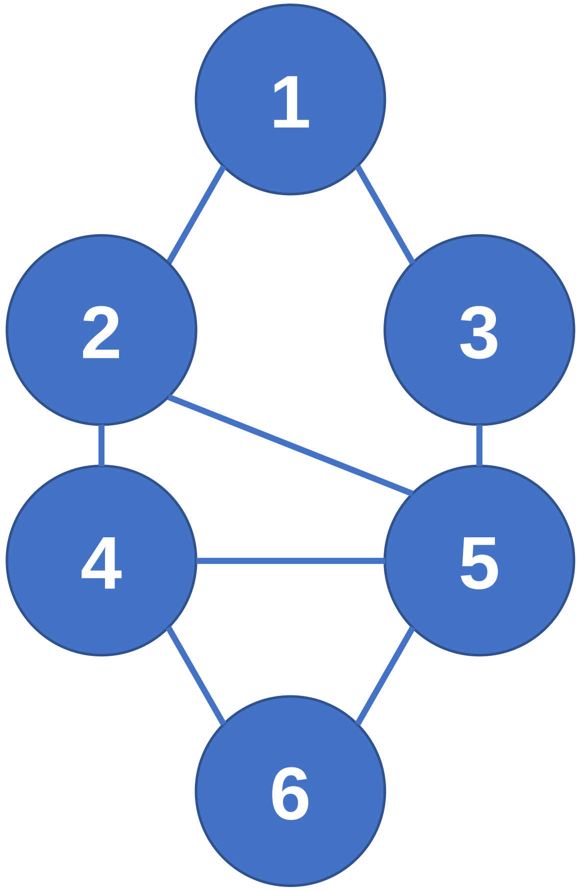
Idea: To go forward (in depth) while there is any such possibility, if not then, backtrack

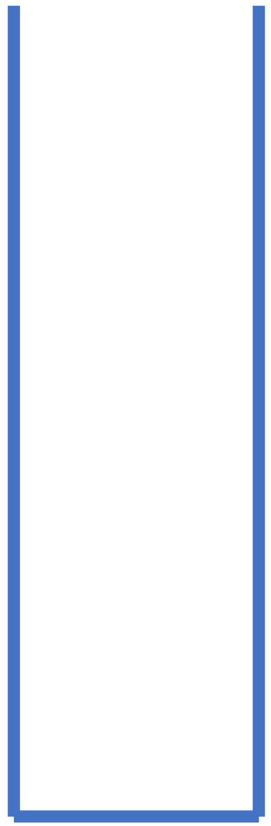
Problem: Since we have cycles, each node may be visited infinite times.

Solution: Use a Boolean visited array.



Depth First Search

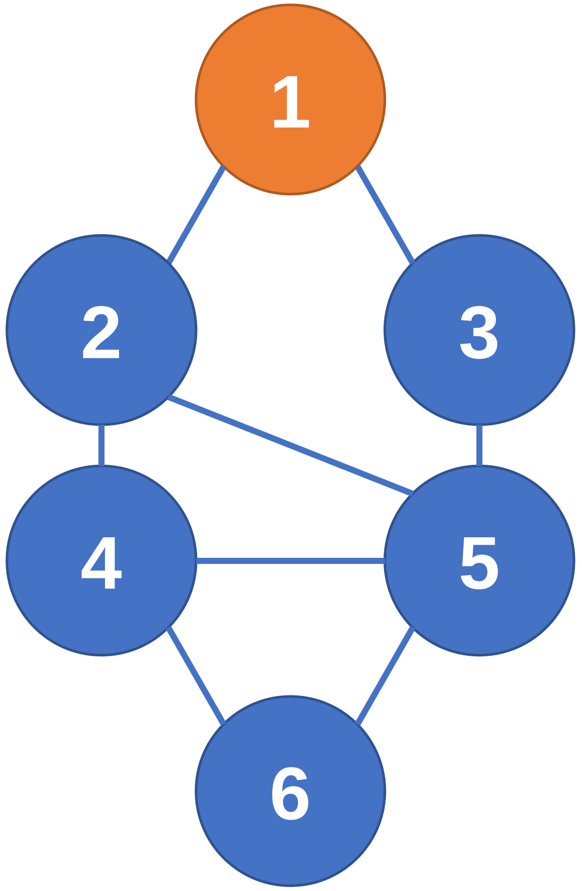


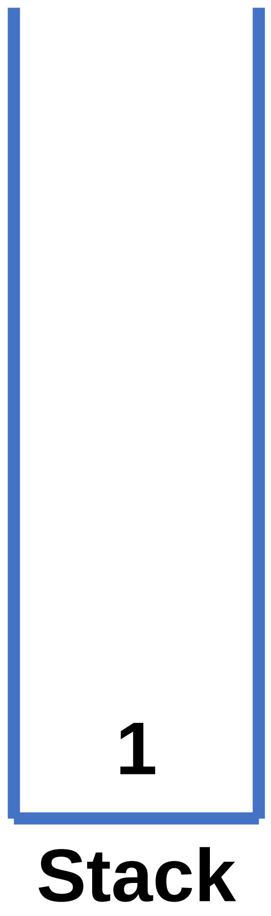


Stack

Output:

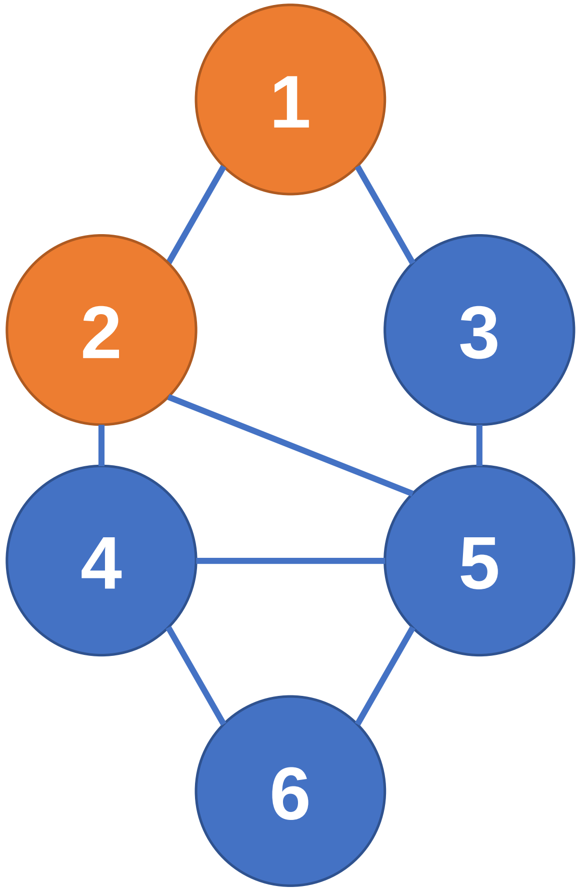
Depth First Search

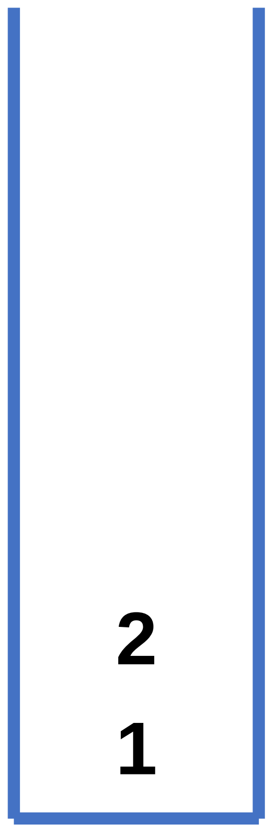




Output: 1

Depth First Search

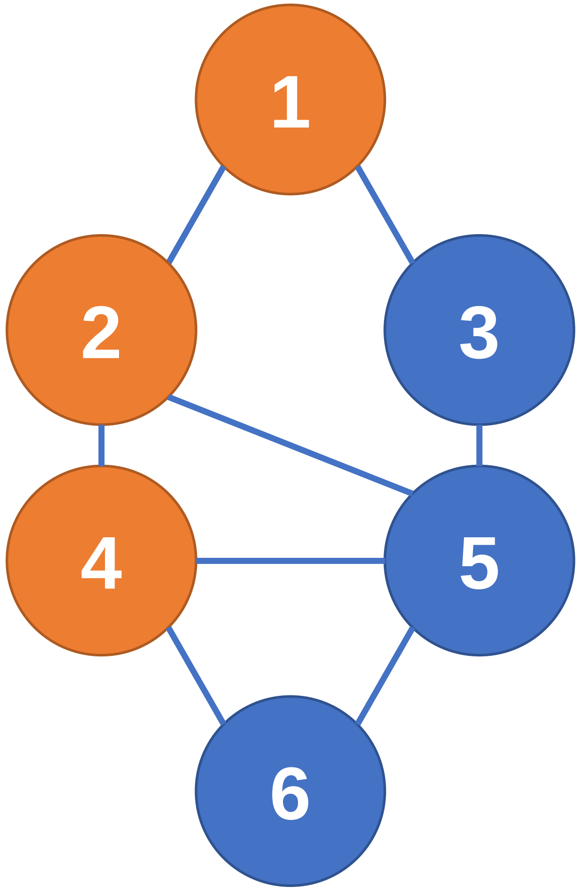


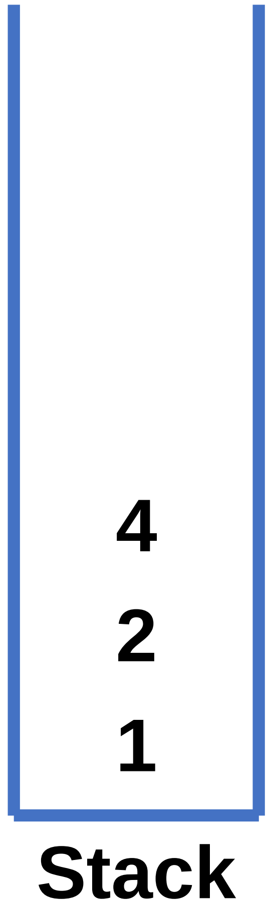


Stack

Output: 1 2

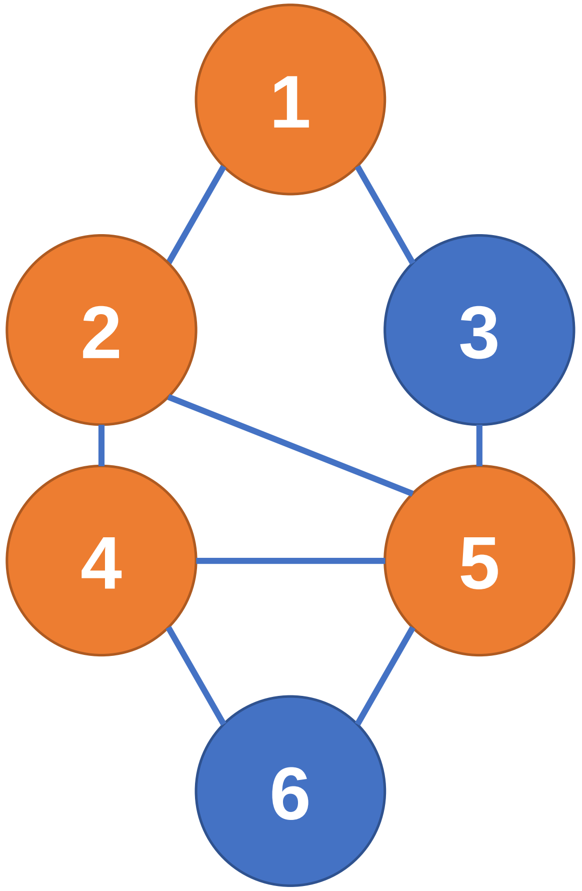
Depth First Search

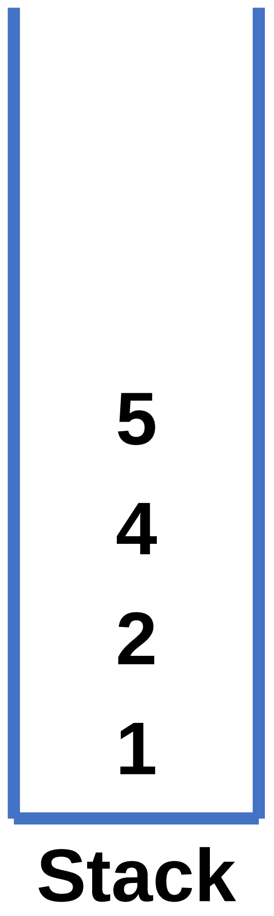




Output: 1 2 4

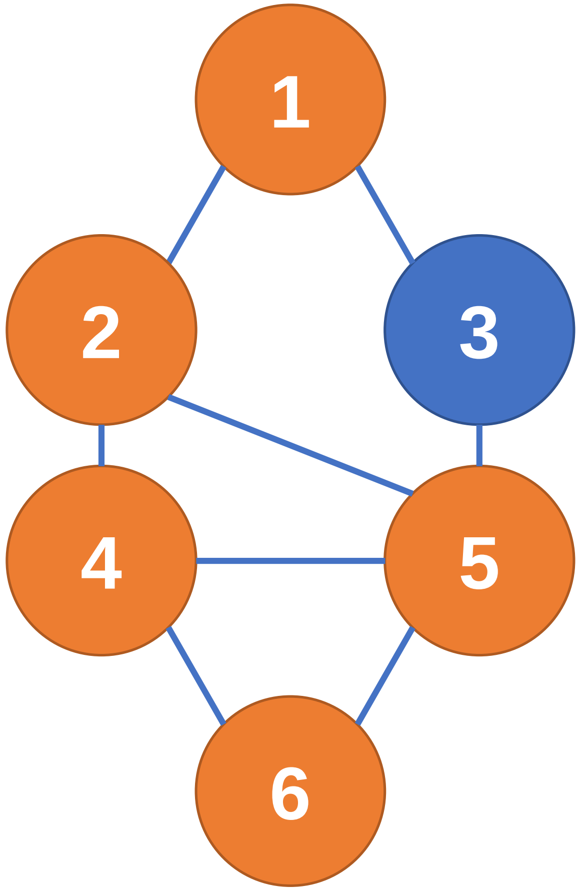
Depth First Search

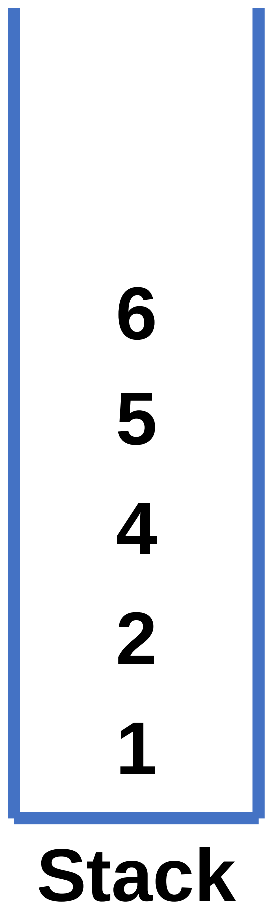




Output: 1 2 4 5

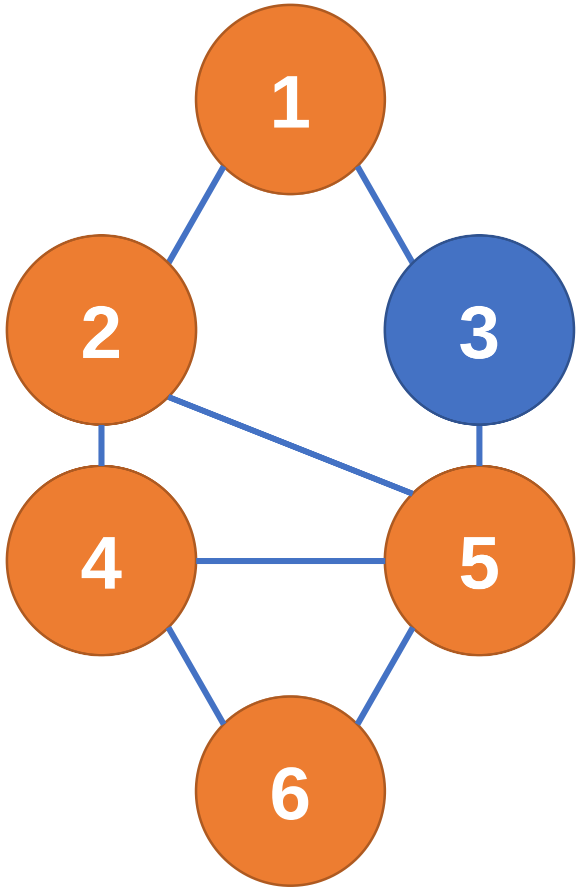
Depth First Search

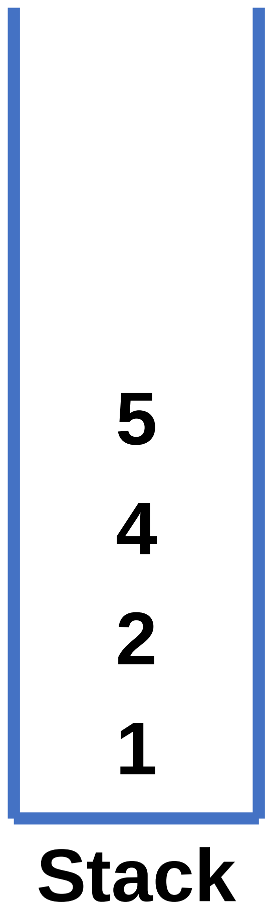




Output: 1 2 4 5 6

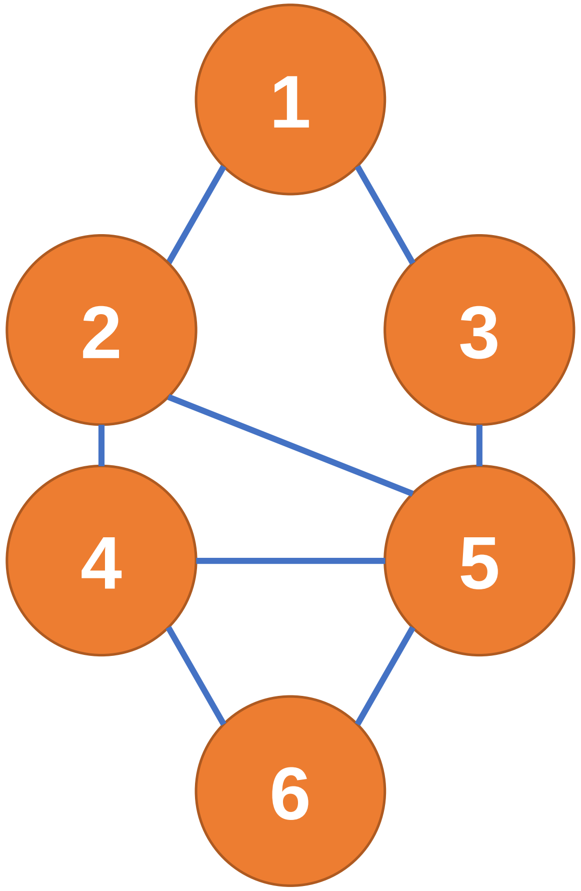
Depth First Search

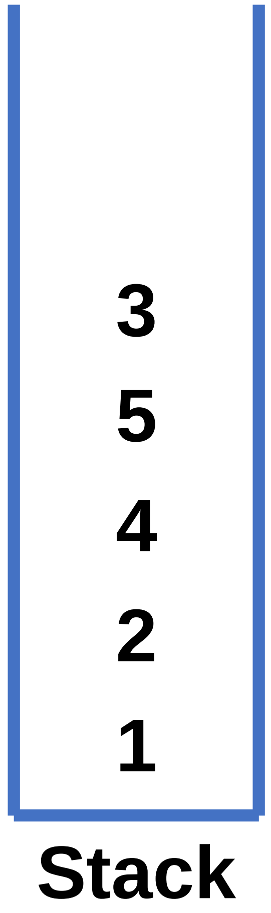




Output: 1 2 4 5 6

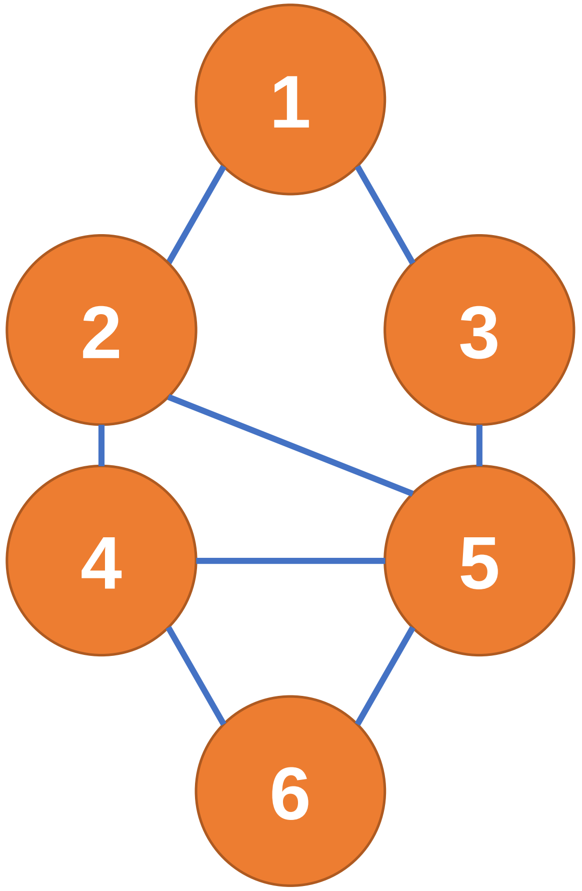
Depth First Search

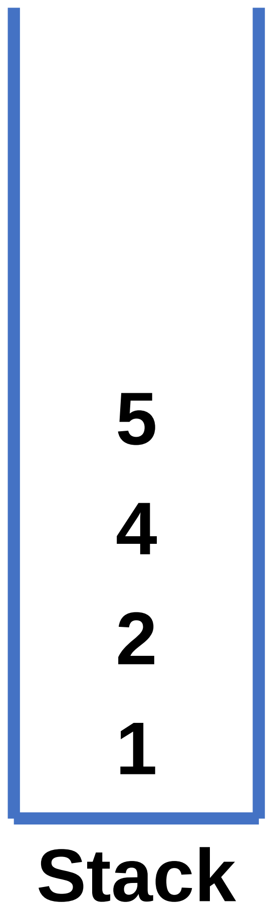




Output: 1 2 4 5 6 3

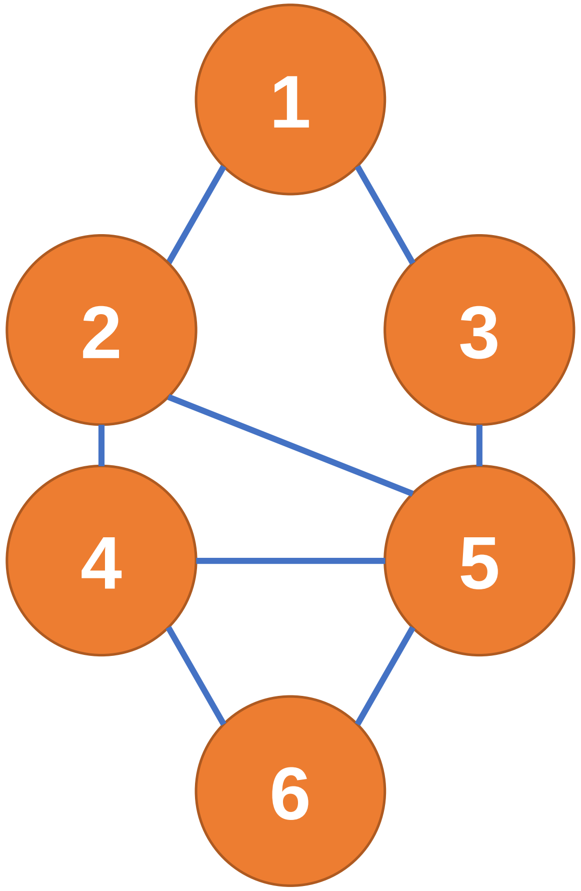
Depth First Search

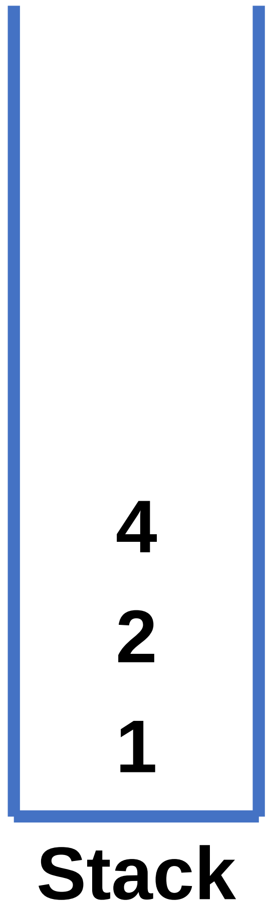




Output: 1 2 4 5 6 3

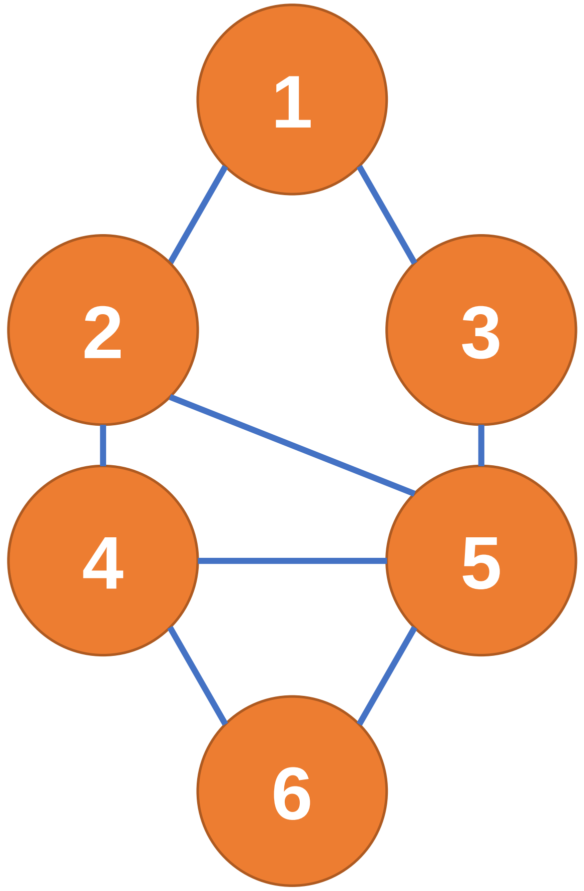
Depth First Search

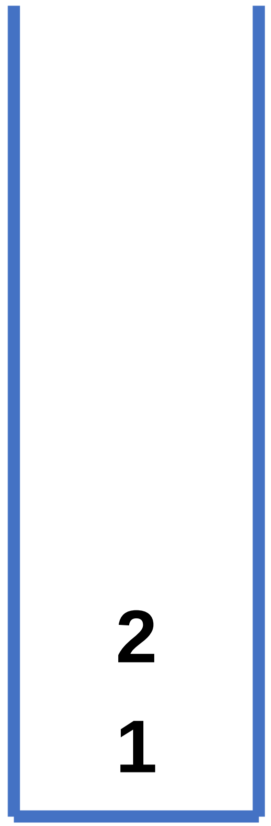




Output: 1 2 4 5 6 3

Depth First Search

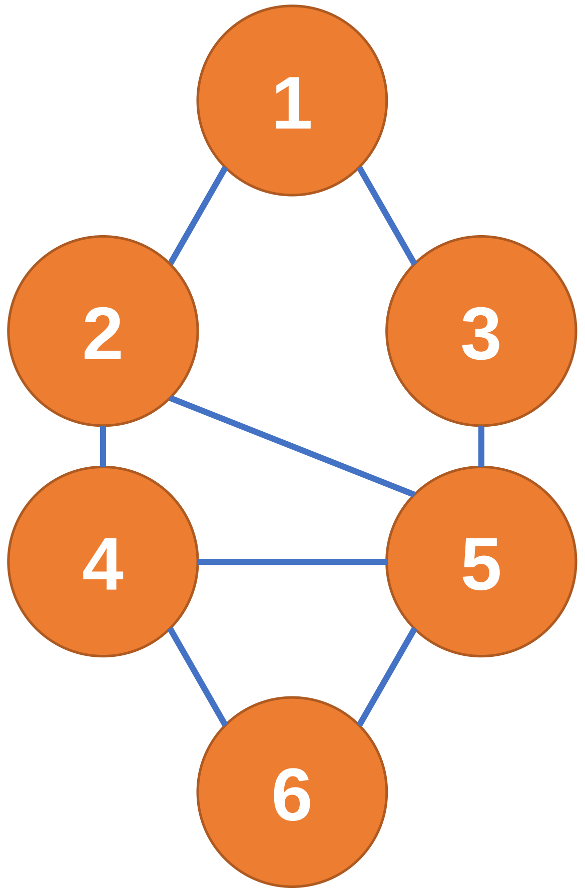


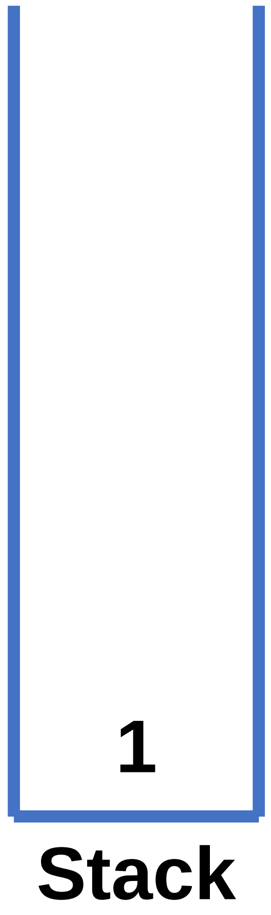


Stack

Output: 1 2 4 5 6 3

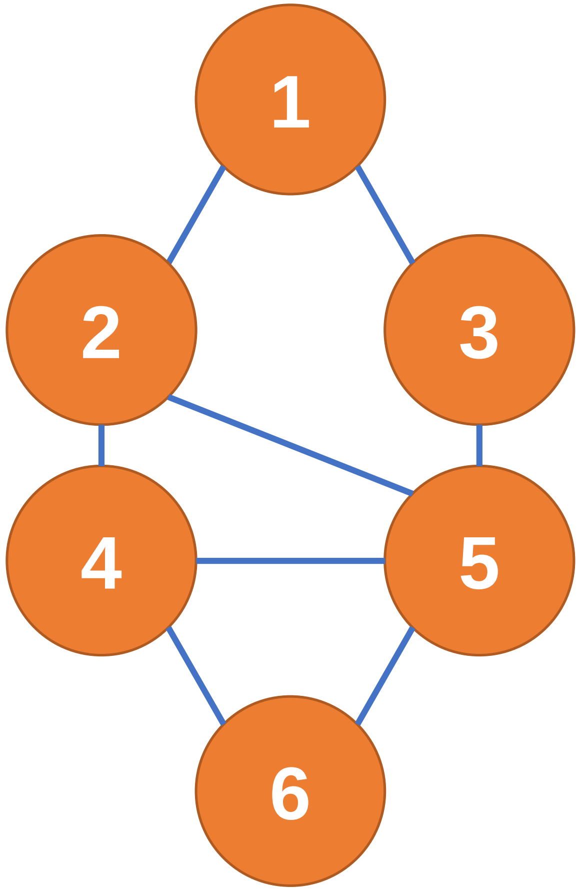
Depth First Search

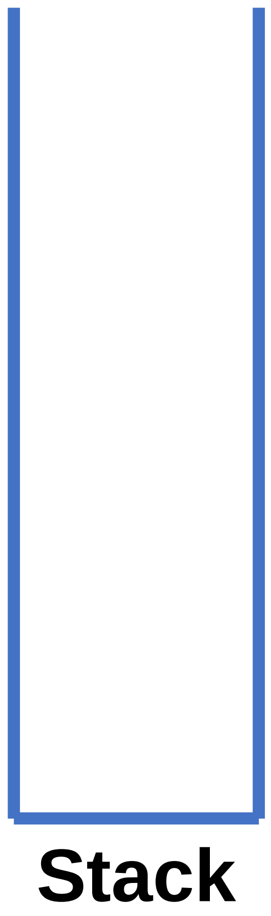




Output: 1 2 4 5 6 3

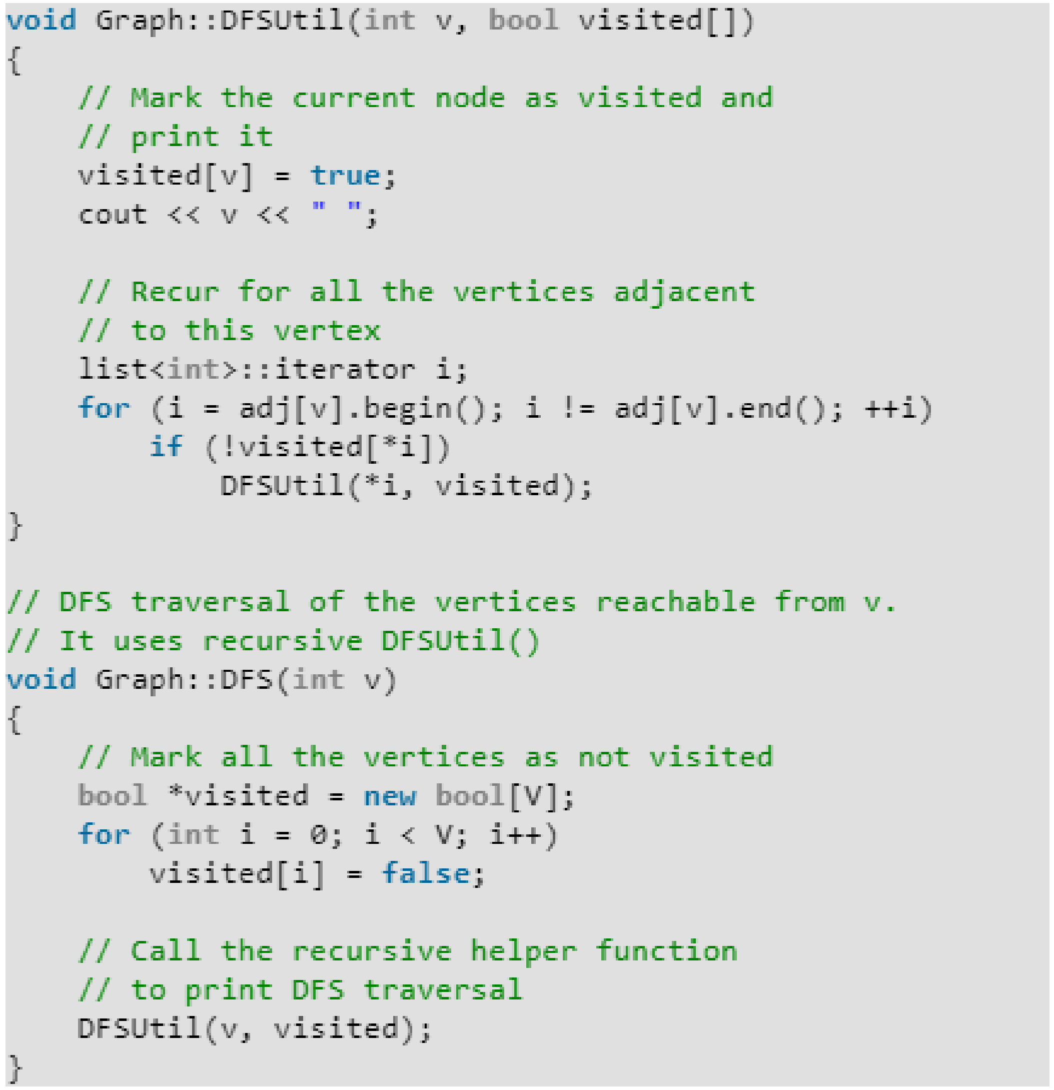
Depth First Search





Output: 1 2 4 5 6 3

Implementation



```
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}
```

Complexity

Time Complexity: O(V+E)

V: Vertices
E: Edges

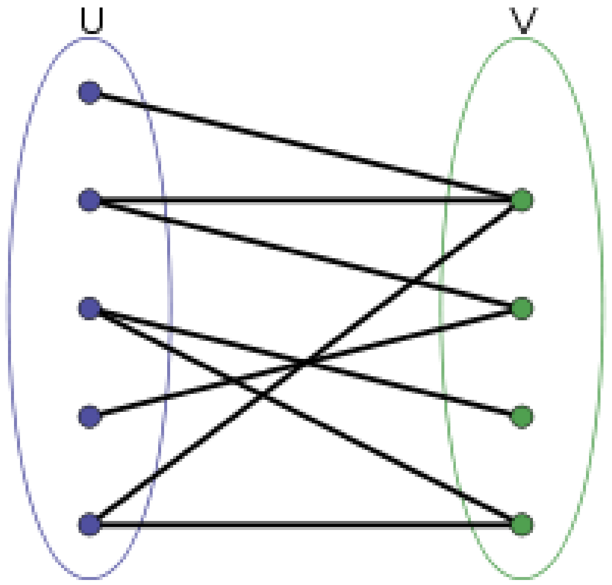
Applications of BFS

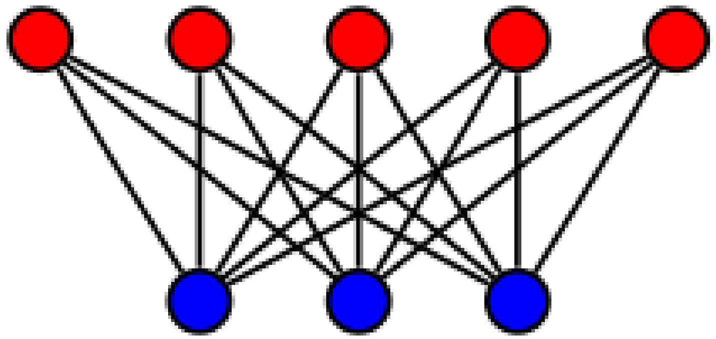
1. Shortest Path in a graph
2. Social Network
3. Cycle Detection in undirected graph
4. <u>To test if a graph is bipartite</u>
5. Broadcasting in a network
6. Path Finding

Applications of BFS

4. <u>Check if graph is bipartite or not</u>

Bipartite Graph





Applications of BFS

4. <u>**Check if graph is bipartite or not**</u>

Use the vertex coloring algorithm:

1) **Start with a vertex and give it a color (RED).**
2) **Run BFS from this vertex. For each new vertex, color it opposite its parents. (p:RED → v:BLUE, p:BLUE → v:RED)**
3) **Check for edges that it doesn’t link two vertices of the same color.**
4) **Repeat steps 2 and 3 until all the vertices are colored RED or BLUE.**

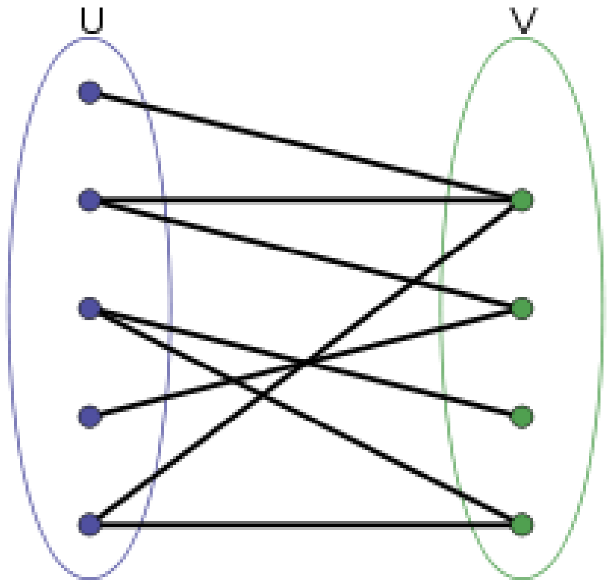
Applications of DFS

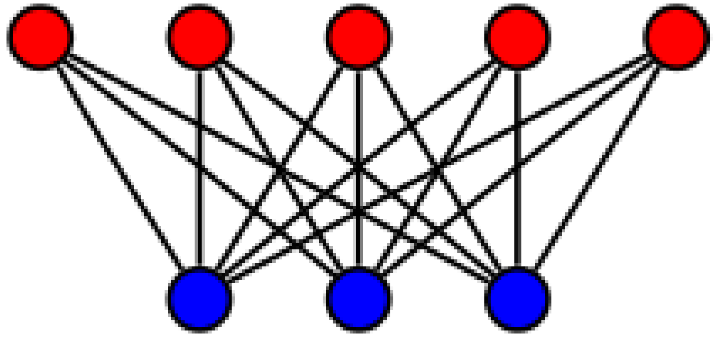
1. **Detecting cycle in a graph**
2. **Path Finding**
3. **<u>To test if a graph is bipartite</u>**
4. **<u>Topological Sort</u>**
5. **Strongly Connected Components**

Applications of DFS

3. <u>**Check if graph is bipartite or not**</u>

Bipartite Graph





Applications of DFS

3. **<u>Check if graph is bipartite or not</u>**

Use the vertex coloring algorithm:

1) **Start with a vertex and give it a color (RED).**
2) **Run DFS from this vertex. For each new vertex, color it opposite its parents. (p:RED → v:BLUE, p:BLUE → v:RED)**
3) **Check for edges that it doesn’t link two vertices of the same color.**
4) **Repeat steps 2 and 3 until all the vertices are colored RED or BLUE.**

Applications of DFS

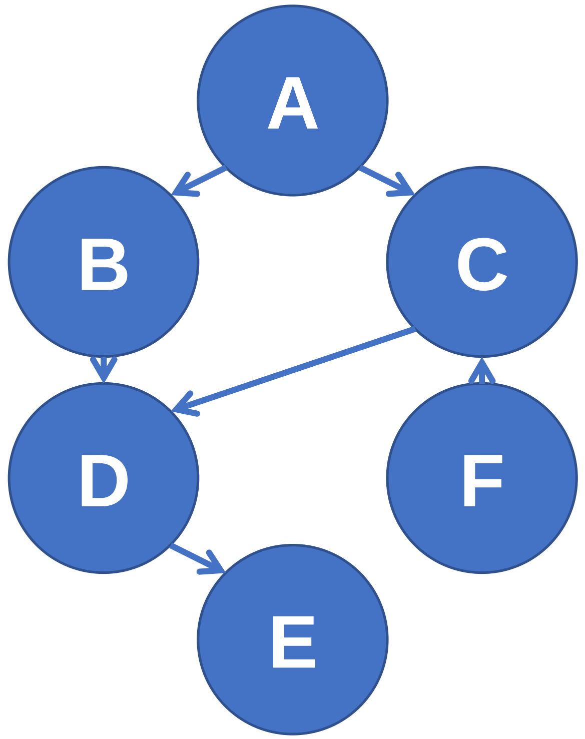
4. **<u>Topological Sort</u>: for a DAG (Directed Acyclic Graph) G=(V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v) then u appears before v in ordering.**

Topological Sort:

F A B C D E

F A C B D E

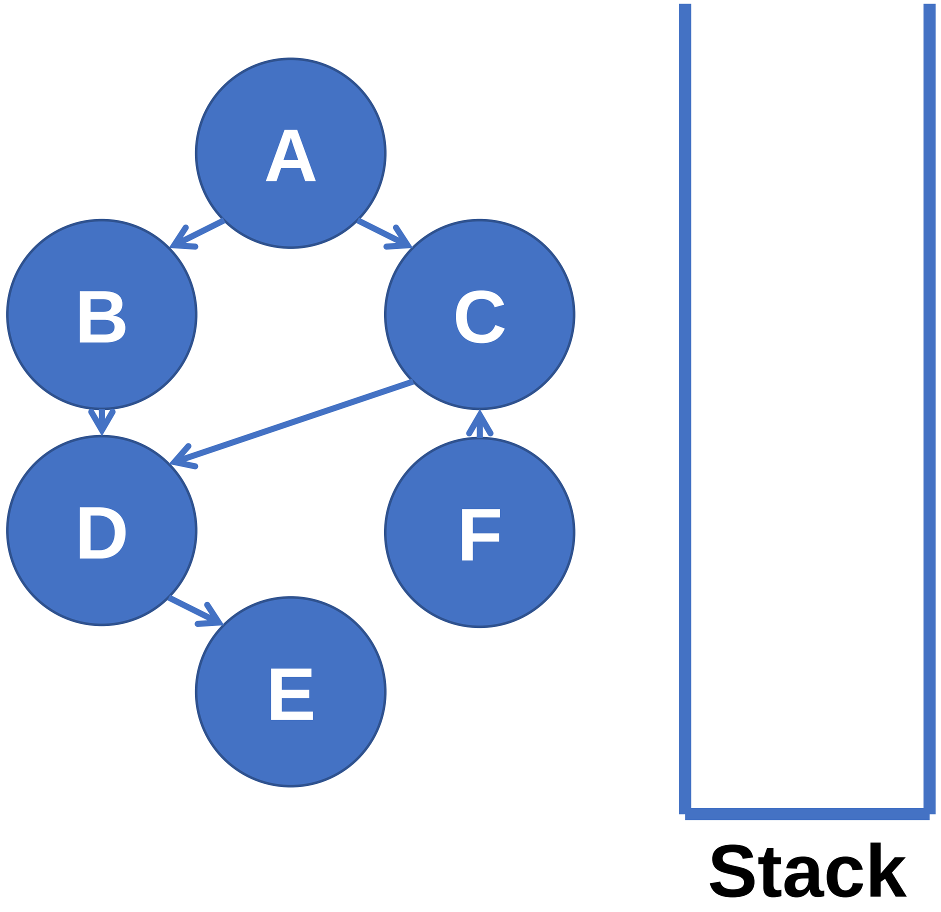
A B F C D E



Applications of DFS

Topological Sort:

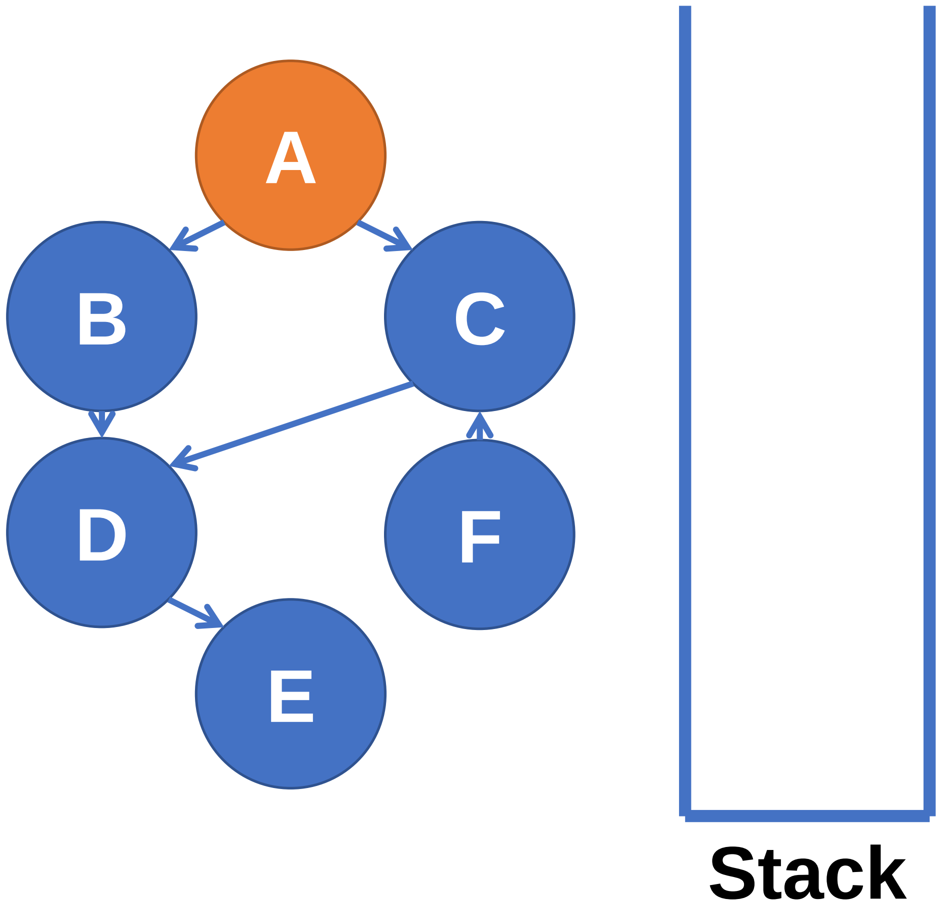
Use Depth First Search using a temporary stack



Applications of DFS

Topological Sort:

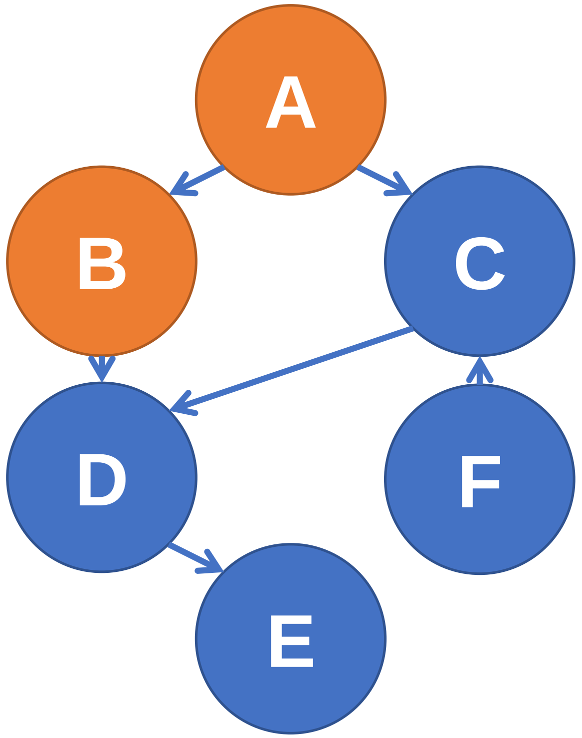
Use Depth First Search using a temporary stack

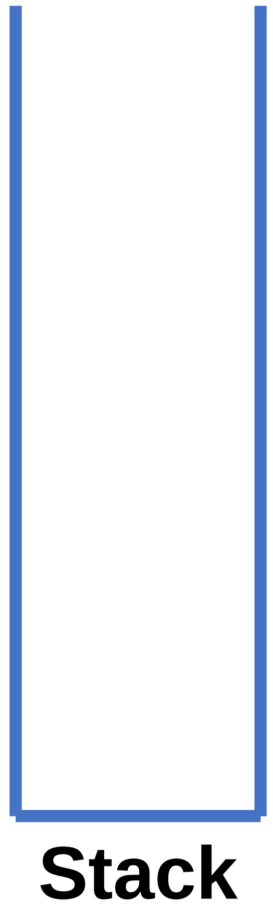


Applications of DFS

Topological Sort:

Use Depth First Search using a temporary stack

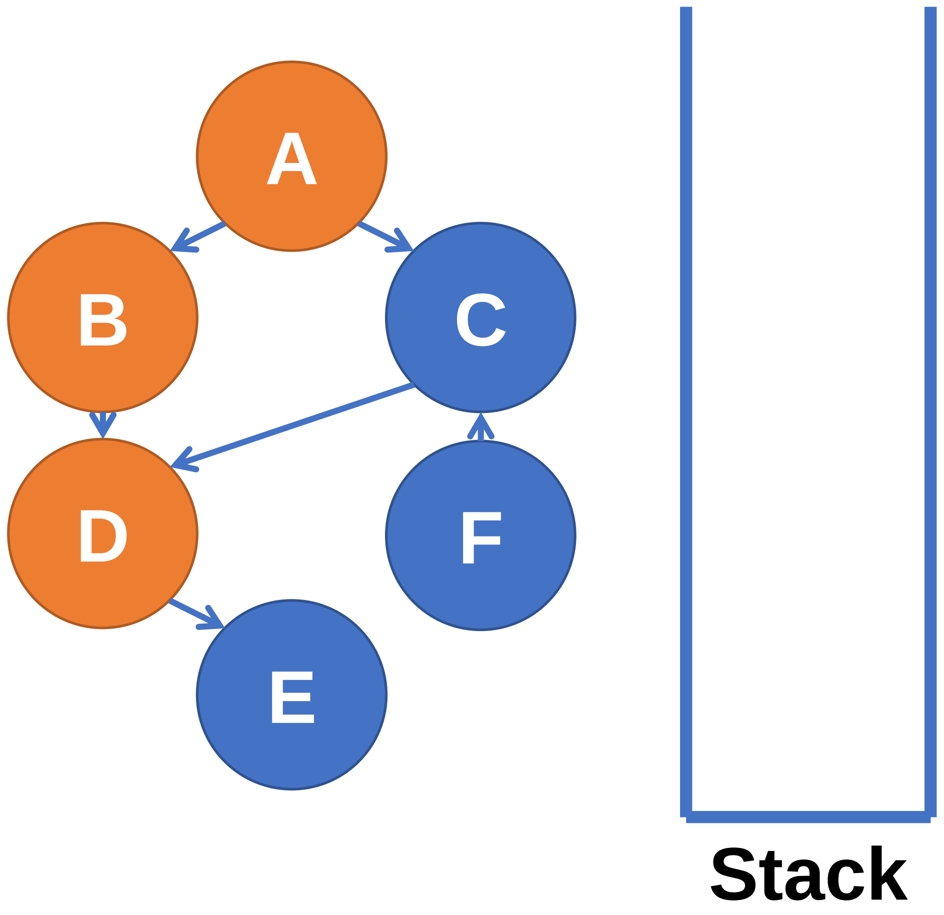




Applications of DFS

Topological Sort:

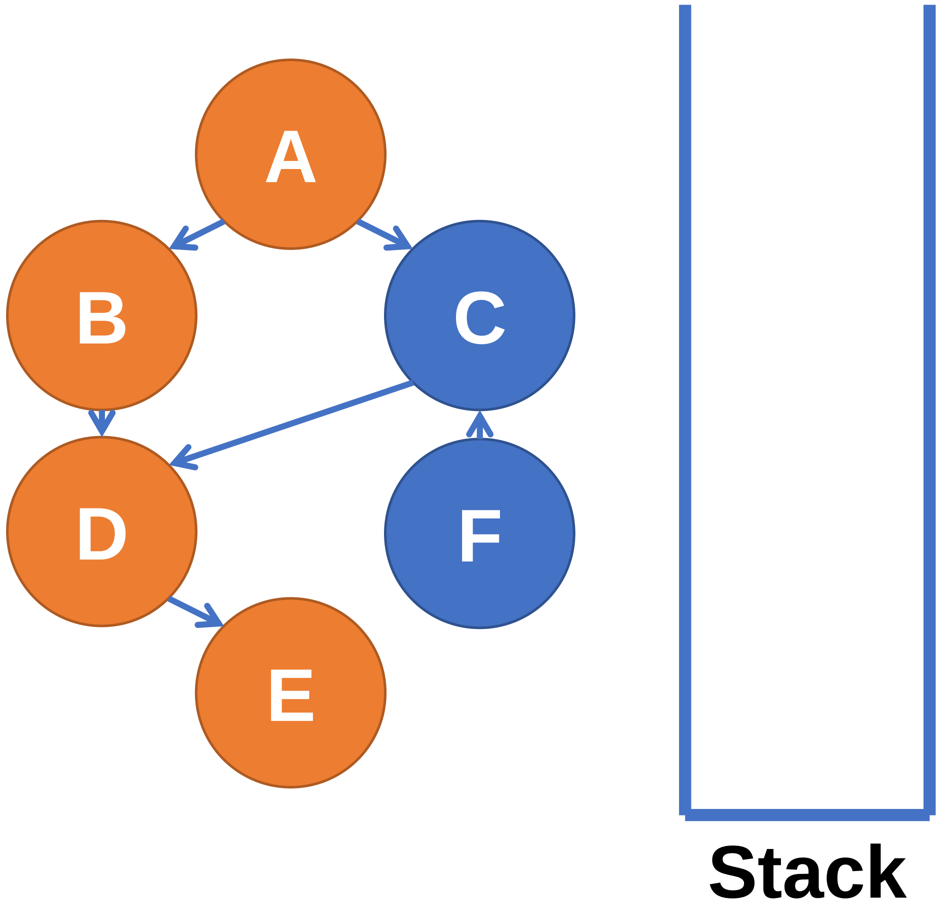
Use Depth First Search using a temporary stack



Applications of DFS

Topological Sort:

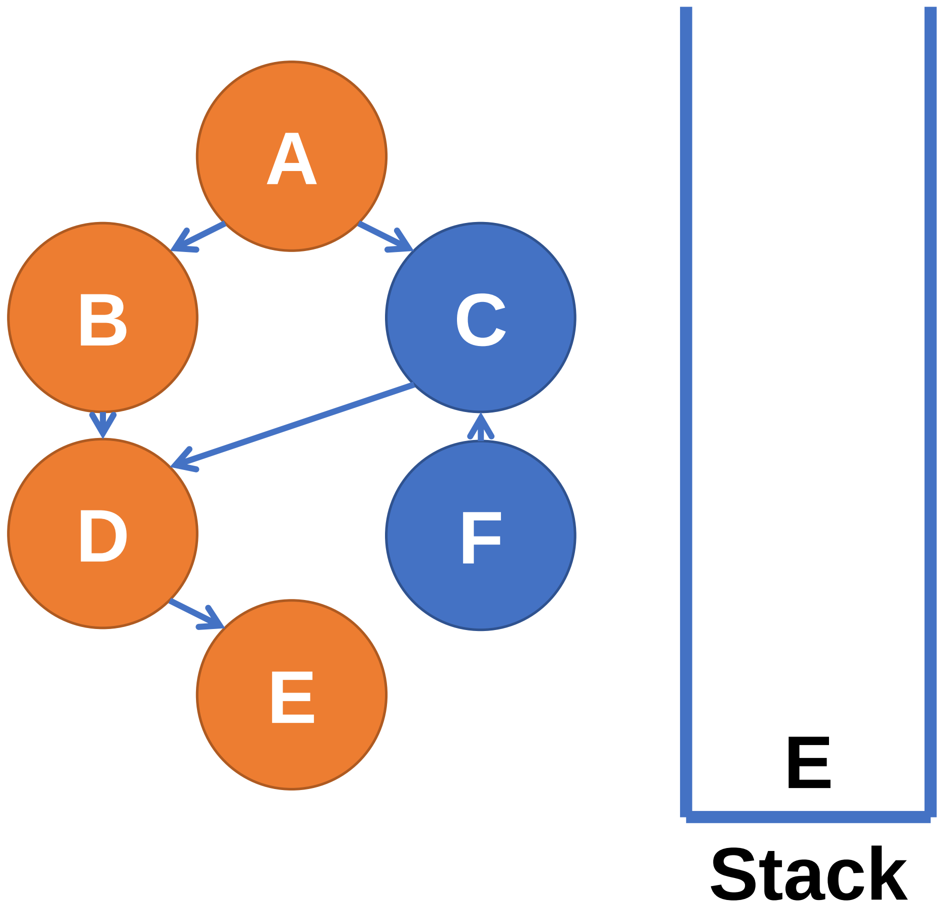
Use Depth First Search using a temporary stack



Applications of DFS

Topological Sort:

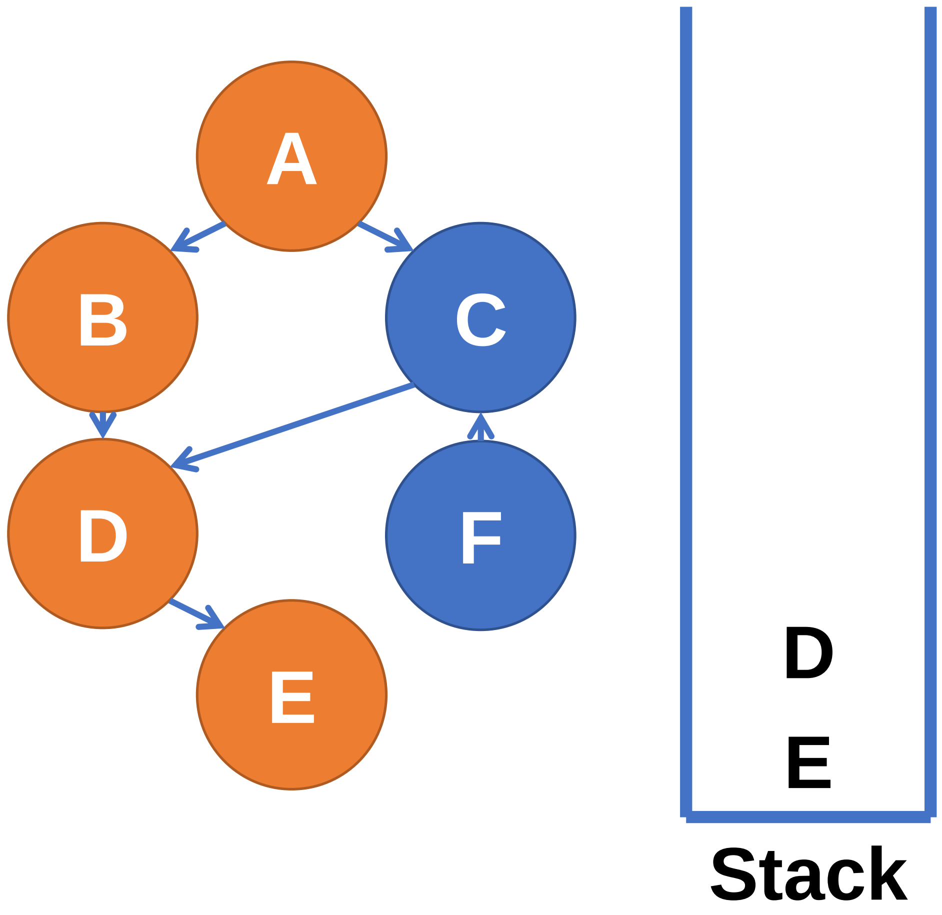
Use Depth First Search using a temporary stack



Applications of DFS

Topological Sort:

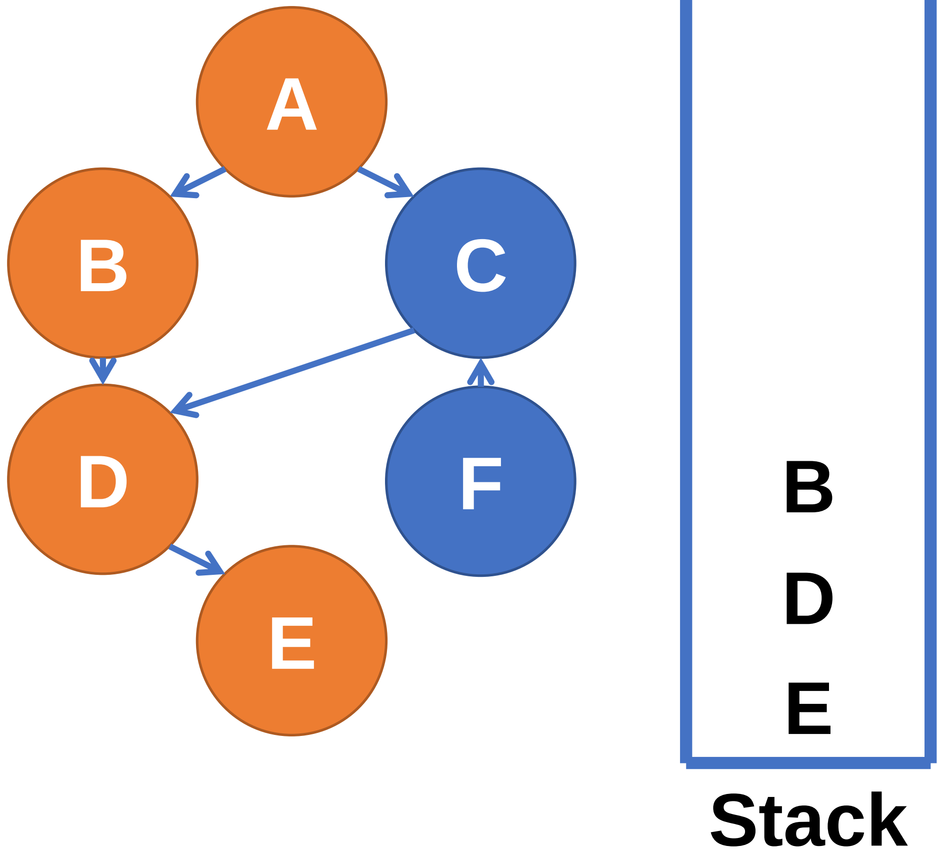
Use Depth First Search using a temporary stack



Applications of DFS

Topological Sort:

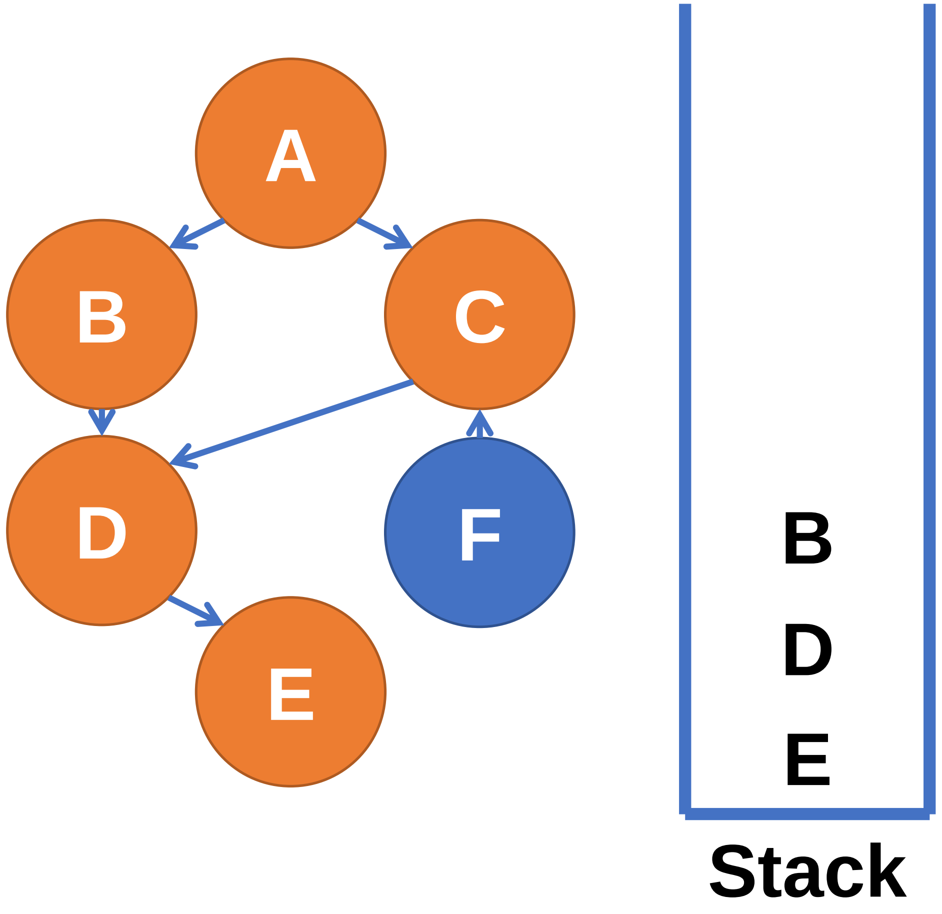
Use Depth First Search using a temporary stack



Applications of DFS

Topological Sort:

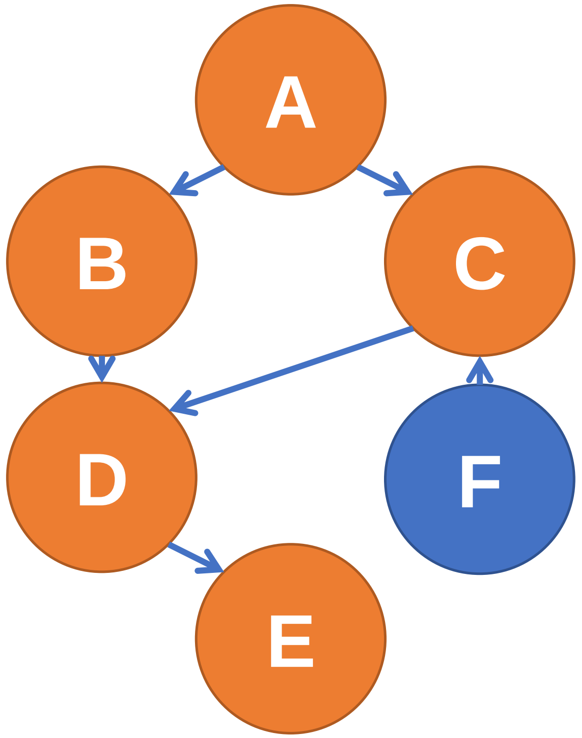
Use Depth First Search using a temporary stack

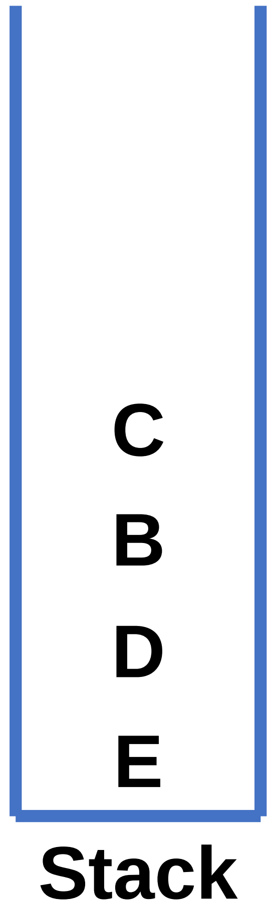


Applications of DFS

Topological Sort:

Use Depth First Search using a temporary stack

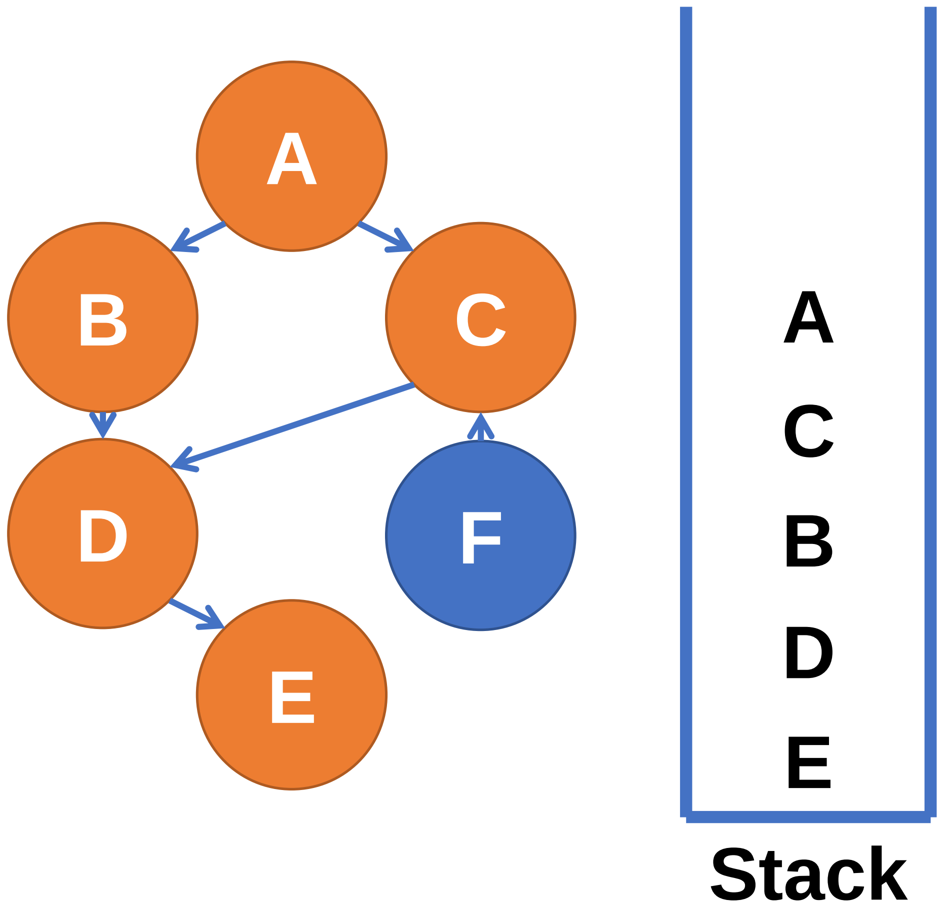




Applications of DFS

Topological Sort:

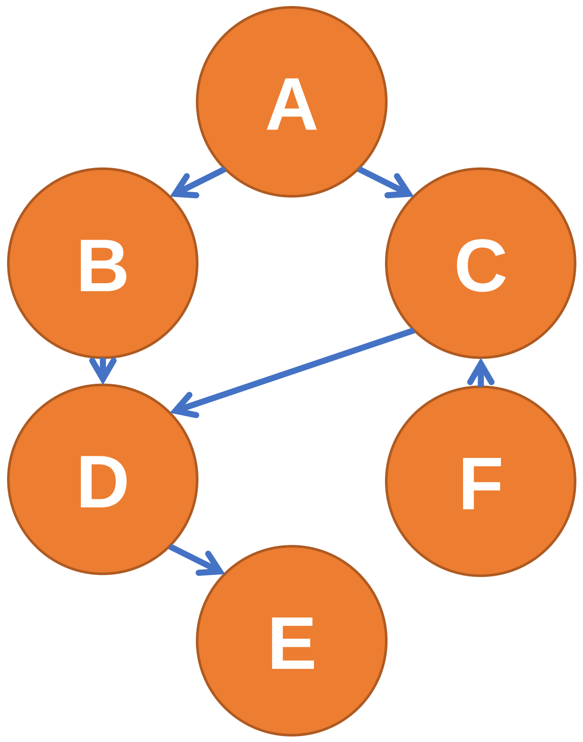
Use Depth First Search using a temporary stack

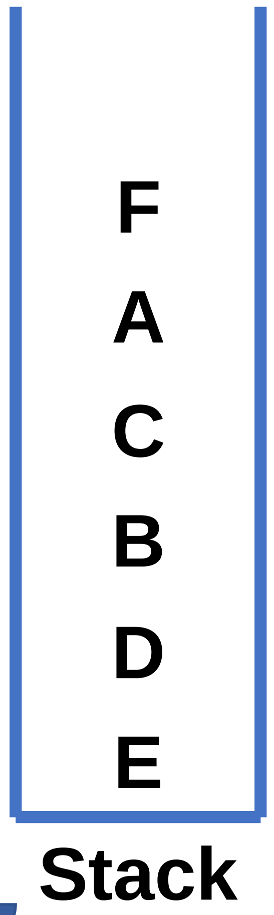


Applications of DFS

Topological Sort:

Use Depth First Search using a temporary stack





F A C B D E

Reference

- Charles Leiserson and Piotr Indyk, “*Introduction to Algorithms*”, September 29, 2004

- https://www.geeksforgeeks.org