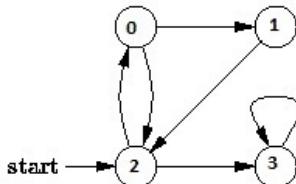


Interview-Corner Archive

Breadth First Traversal or BFS for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of [this post](#)). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



Following are C++ and Java implementations of simple Breadth First Traversal from a given source.

The C++ implementation uses [adjacency list representation](#) of graphs. STL's [list container](#) is used to store lists of adjacent nodes and queue of nodes needed for BFS traversal.

C++

```
// Program to print BFS traversal from a given source vertex. BFS(int s)
// traverses vertices reachable from s.
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using adjacency list representation
class Graph
{
    int V; // No. of vertices
    list<int> *adj; // Pointer to an array containing adjacency lists
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void BFS(int s); // prints BFS traversal from a given source s
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

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
    queue.push_back(s);
    visited[s] = true;
```

```

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);
        }
    }
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
        << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}

```

Java

```

// Java program to print BFS traversal from a given source vertex.
// BFS(int s) traverses vertices reachable from s.
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V; // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency Lists

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
    }
}

```

```

}

// prints BFS traversal from a given source s
void BFS(int s)
{
    // Mark all the vertices as not visited(By default
    // set as false)
    boolean visited[] = new boolean[V];

    // Create a queue for BFS
    LinkedList<Integer> queue = new LinkedList<Integer>();

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

    while (queue.size() != 0)
    {
        // Dequeue a vertex from queue and print it
        s = queue.poll();
        System.out.print(s+" ");

        // Get all adjacent vertices of the dequeued vertex s
        // If a adjacent has not been visited, then mark it
        // visited and enqueue it
        Iterator<Integer> i = adj[s].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
            {
                visited[n] = true;
                queue.add(n);
            }
        }
    }
}

// Driver method to
public static void main(String args[])
{
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    System.out.println("Following is Breadth First Traversal "+
        "(starting from vertex 2)");

    g.BFS(2);
}
}
// This code is contributed by Aakash Hasija

```

Python

```

# Program to print BFS traversal from a given source
# vertex. BFS(int s) traverses vertices reachable
# from s.
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph

```

```

self.graph = defaultdict(list)

# function to add an edge to graph
def addEdge(self,u,v):
    self.graph[u].append(v)

# Function to print a BFS of graph
def BFS(self, s):

    # Mark all the vertices as not visited
    visited = [False]*len(self.graph))

    # Create a queue for BFS
    queue = []

    # Mark the source node as visited and enqueue it
    queue.append(s)
    visited[s] = True

    while queue:

        # Dequeue a vertex from queue and print it
        s = queue.pop(0)
        print s,

        # 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 in self.graph[s]:
            if visited[i] == False:
                queue.append(i)
                visited[i] = True

# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is Breadth First Traversal (starting from vertex 2)"
g.BFS(2)

```

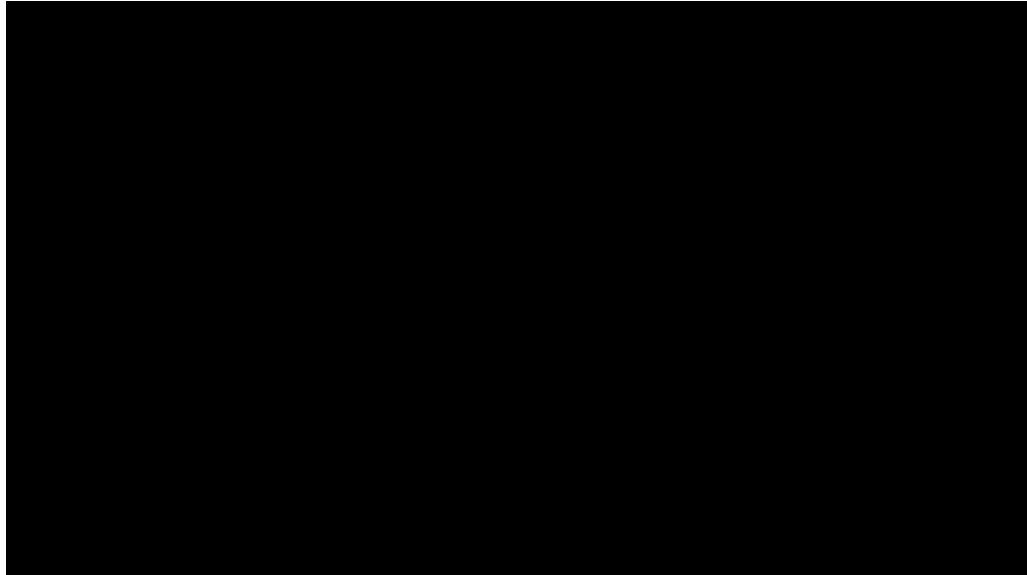
This code is contributed by Neelam Yadav

Output:

Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1

Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the [DFS modified version](#)).

Time Complexity: $O(V+E)$ where V is number of vertices in the graph and E is number of edges in the graph.



You may like to see below also :

- Depth First Traversal
- Applications of Breadth First Traversal
- Applications of Depth First Search

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

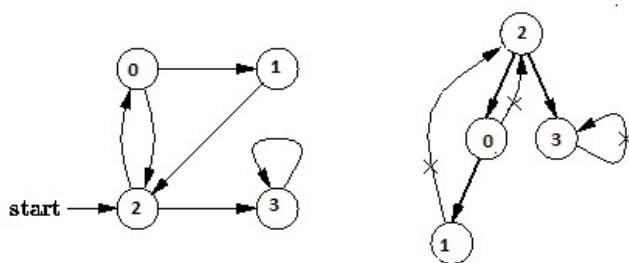
GATE CS Corner Company Wise Coding Practice

Graph
Queue
BFS
Stack-Queue

Depth First Traversal or DFS for a Graph

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.



See [this post](#) for all applications of Depth First Traversal.

Following are implementations of simple Depth First Traversal. The C++ implementation uses adjacency list representation of graphs. STL's list container is used to store lists of adjacent nodes.

C++

```
// C++ program to print DFS traversal from a given vertex in a given graph
#include<iostream>
#include<list>

using namespace std;

// Graph class represents a directed graph using adjacency list representation
class Graph
```

```

{
    int V; // No. of vertices
    list<int> *adj; // Pointer to an array containing adjacency lists
    void DFSUtil(int v, bool visited[]); // A function used by DFS
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void DFS(int v); // DFS traversal of the vertices reachable from v
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

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);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}

```

Java

```

// Java program to print DFS traversal from a given given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph

```

```

{
    private int V; // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w); // Add w to v's list.
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS(int v)
    {
        // Mark all the vertices as not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];

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

    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Depth First Traversal "+ 
                           "(starting from vertex 2)");

        g.DFS(2);
    }
}

// This code is contributed by Aakash Hasija

```

Python

```

# Python program to print DFS traversal from a
# given given graph
from collections import defaultdict

```

```

# This class represents a directed graph using
# adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self,v,visited):

        # Mark the current node as visited and print it
        visited[v]= True
        print v,

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)

    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self,v):

        # Mark all the vertices as not visited
        visited = [False]*len(self.graph))

        # Call the recursive helper function to print
        # DFS traversal
        self.DFSUtil(v,visited)

# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is DFS from (starting from vertex 2)"
g.DFS(2)

# This code is contributed by Neelam Yadav

```

Output:

```

Following is Depth First Traversal (starting from vertex 2)
2 0 1 3

```

Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To do complete DFS traversal of such graphs, we must call DFSUtil() for every vertex. Also, before calling DFSUtil(), we should check if it is already printed by some other call of DFSUtil(). Following implementation does the complete graph traversal even if the nodes are unreachable. The differences from the above code are highlighted in the below code.

C++

```

// C++ program to print DFS traversal for a given given graph
#include<iostream>
#include <list>
using namespace std;

class Graph

```

```

{
    int V; // No. of vertices
    list<int> *adj; // Pointer to an array containing adjacency lists
    void DFSUtil(int v, bool visited[]); // A function used by DFS
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void DFS(); // prints DFS traversal of the complete graph
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

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);
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void Graph::DFS()
{
    // 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
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            DFSUtil(i, visited);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal\n";
    g.DFS();

    return 0;
}

```

Java

```

// Java program to print DFS traversal from a given given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list

```

```

// representation
class Graph
{
    private int V; // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0;i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w); // Add w to v's list.
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n,visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS()
    {
        // Mark all the vertices as not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];

        // Call the recursive helper function to print DFS traversal
        // starting from all vertices one by one
        for (int i=0;i<V; ++i)
            if (visited[i] == false)
                DFSUtil(i, visited);
    }

    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Depth First Traversal");

        g.DFS();
    }
}

// This code is contributed by Aakash Hasija

```

Python

```
# Python program to print DFS traversal for complete graph
```

```
from collections import defaultdict
```

```
# This class represents a directed graph using adjacency
```

```
# list representation
```

```
class Graph:
```

```
    # Constructor
```

```
    def __init__(self):
```

```
        # default dictionary to store graph
```

```
        self.graph = defaultdict(list)
```

```
    # function to add an edge to graph
```

```
    def addEdge(self,u,v):
```

```
        self.graph[u].append(v)
```

```
    # A function used by DFS
```

```
    def DFSUtil(self, v, visited):
```

```
        # Mark the current node as visited and print it
```

```
        visited[v]= True
```

```
        print v,
```

```
        # Recur for all the vertices adjacent to
```

```
        # this vertex
```

```
        for i in self.graph[v]:
```

```
            if visited[i] == False:
```

```
                self.DFSUtil(i, visited)
```

```
    # The function to do DFS traversal. It uses
```

```
    # recursive DFSUtil()
```

```
    def DFS(self):
```

```
        V = len(self.graph) #total vertices
```

```
        # Mark all the vertices as not visited
```

```
        visited =[False]*V
```

```
        # Call the recursive helper function to print
```

```
        # DFS traversal starting from all vertices one
```

```
        # by one
```

```
        for i in range(V):
```

```
            if visited[i] == False:
```

```
                self.DFSUtil(i, visited)
```

```
# Driver code
```

```
# Create a graph given in the above diagram
```

```
g = Graph()
```

```
g.addEdge(0, 1)
```

```
g.addEdge(0, 2)
```

```
g.addEdge(1, 2)
```

```
g.addEdge(2, 0)
```

```
g.addEdge(2, 3)
```

```
g.addEdge(3, 3)
```

```
print "Following is Depth First Traversal"
```

```
g.DFS()
```

```
# This code is contributed by Neelam Yadav
```

Output:

```
Following is Depth First Traversal
```

```
0 1 2 3
```

Time Complexity: $O(V+E)$ where V is number of vertices in the graph and E is number of edges in the graph.

Breadth First Traversal for a Graph

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
DFS

Greedy Algorithms | Set 7 (Dijkstra's shortest path algorithm)

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

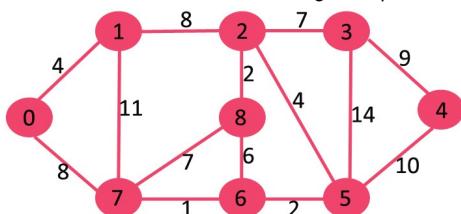
Dijkstra's algorithm is very similar to [Prim's algorithm for minimum spanning tree](#). Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

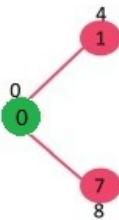
Algorithm

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

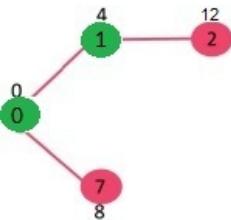
Let us understand with the following example:



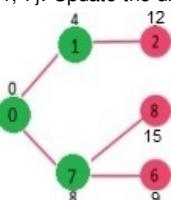
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*,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



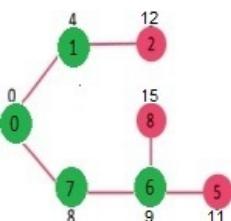
Pick the vertex with minimum distance value and not already included in SPT (not in sptSet). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



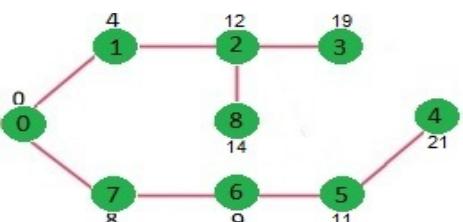
Pick the vertex with minimum distance value and not already included in SPT (not in sptSet). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



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



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



How to implement the above algorithm?

We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store shortest distance values of all vertices.

C/C++

```
// A C / C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph

#include <stdio.h>
#include <limits.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
    int min = INT_MAX, min_index;
```



```
dijkstra(graph, 0);

return 0;
}
```

Java

```
// A Java program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    // A utility function to find the vertex with minimum distance value,
    // from the set of vertices not yet included in shortest path tree
    static final int V=9;
    int minDistance(int dist[], Boolean sptSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index=-1;

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

        return min_index;
    }

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

    // Function that implements Dijkstra's single source shortest path
    // algorithm for a graph represented using adjacency matrix
    // representation
    void dijkstra(int graph[][], int src)
    {
        int dist[] = new int[V]; // The output array. dist[i] will hold
                               // the shortest distance from src to i

        // sptSet[i] will true if vertex i is included in shortest
        // path tree or shortest distance from src to i is finalized
        Boolean sptSet[] = new Boolean[V];

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

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

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

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

            // Update dist value of the adjacent vertices of u
            for (int v = 0; v < V; v++)
                if (!sptSet[v] && graph[u][v] != 0 && dist[v] > dist[u] + graph[u][v])
                    dist[v] = dist[u] + graph[u][v];
        }
    }
}
```

```

// Update dist value of the adjacent vertices of the
// picked vertex.
for (int v = 0; v < V; v++)

    // Update dist[v] only if it is not in sptSet, there is an
    // edge from u to v, and total weight of path from src to
    // v through u is smaller than current value of dist[v]
    if (!sptSet[v] && graph[u][v] != Integer.MAX_VALUE &&
        dist[u] != Integer.MAX_VALUE &&
        dist[u] + graph[u][v] < dist[v])
        dist[v] = dist[u] + graph[u][v];
}

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

// Driver method
public static void main (String[] args)
{
/* Let us create the example graph discussed above */
int graph[][] = new int[][]{{0, 4, 0, 0, 0, 0, 0, 8, 0},
                           {4, 0, 8, 0, 0, 0, 0, 11, 0},
                           {0, 8, 0, 7, 0, 4, 0, 0, 2},
                           {0, 0, 7, 0, 9, 14, 0, 0, 0},
                           {0, 0, 0, 9, 0, 10, 0, 0, 0},
                           {0, 0, 4, 14, 10, 0, 2, 0, 0},
                           {0, 0, 0, 0, 2, 0, 1, 6},
                           {8, 11, 0, 0, 0, 0, 1, 0, 7},
                           {0, 0, 2, 0, 0, 0, 6, 7, 0}
                         };
ShortestPath t = new ShortestPath();
t.dijkstra(graph, 0);
}
}

//This code is contributed by Aakash Hasija

```

Output:

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
Greedy
Dijkstra
Graph
Greedy Algorithm
shortest path

Dynamic Programming | Set 16 (Floyd Warshall Algorithm)

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

Example:

Input:

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

which represents the following graph



Note that the value of $graph[i][j]$ is 0 if i is equal to j

And $graph[i][j]$ is INF (infinite) if there is no edge from vertex i to j .

Output:

Shortest distance matrix

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

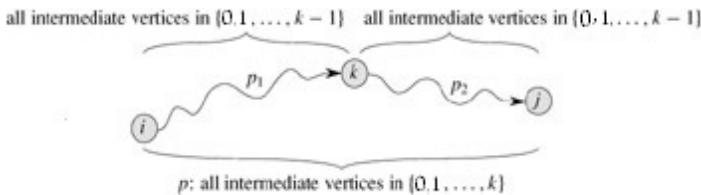
Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices $\{0, 1, 2, \dots, k-1\}$ as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j . We keep the value of $dist[i][j]$ as it is.

2) k is an intermediate vertex in shortest path from i to j . We update the value of $dist[i][j]$ as $dist[i][k] + dist[k][j]$.

The following figure is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.



Following is implementations of the Floyd Warshall algorithm.

C/C++

```
// C Program for Floyd Warshall Algorithm
#include<stdio.h>

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough value. This value will be used
for vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm
void floydWarshall (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the shortest
    distances between every pair of vertices */
    int dist[V][V], i, j, k;

    /* Initialize the solution matrix same as input graph matrix. Or
    we can say the initial values of shortest distances are based
    on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate vertices.
    ---> Before start of a iteration, we have shortest distances between all
    pairs of vertices such that the shortest distances consider only the
    vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
    ----> After the end of a iteration, vertex no. k is added to the set of
    intermediate vertices and the set becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++)
            {
                // If vertex k is on the shortest path from
                // i to j, then update the value of dist[i][j]
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
    printf ("Following matrix shows the shortest distances"
           " between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {

```

```

for (int j = 0; j < V; j++)
{
    if (dist[i][j] == INF)
        printf("%7s", "INF");
    else
        printf ("%7d", dist[i][j]);
}
printf("\n");
}

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
       10
       (0)----->(3)
       |         |
       5 |         |
       |         | 1
       \|         |
       (1)----->(2)
       3           */
    int graph[V][V] = { {0, 5, INF, 10},
                        {INF, 0, 3, INF},
                        {INF, INF, 0, 1},
                        {INF, INF, INF, 0}
                      };
    // Print the solution
    floydWarshall(graph);
    return 0;
}

```

Java

```

// A Java program for Floyd Warshall All Pairs Shortest
// Path algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

class AllPairShortestPath
{
    final static int INF = 99999, V = 4;

    void floydWarshall(int graph[][])
    {
        int dist[][] = new int[V][V];
        int i, j, k;

        /* Initialize the solution matrix same as input graph matrix.
        Or we can say the initial values of shortest distances
        are based on shortest paths considering no intermediate
        vertex. */
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i][j] = graph[i][j];

        /* Add all vertices one by one to the set of intermediate
        vertices.
        --> Before start of a iteration, we have shortest
        distances between all pairs of vertices such that
        the shortest distances consider only the vertices in
        set {0, 1, 2, .. k-1} as intermediate vertices.
        ----> After the end of a iteration, vertex no. k is added
        to the set of intermediate vertices and the set
        becomes {0, 1, 2, .. k} */
        for (k = 0; k < V; k++)
        {
            // Pick all vertices as source one by one
            for (i = 0; i < V; i++)

```

```

{
    // Pick all vertices as destination for the
    // above picked source
    for (j = 0; j < V; j++)
    {
        // If vertex k is on the shortest path from
        // i to j, then update the value of dist[i][j]
        if (dist[i][k] + dist[k][j] < dist[i][j])
            dist[i][j] = dist[i][k] + dist[k][j];
    }
}

// Print the shortest distance matrix
printSolution(dist);
}

void printSolution(int dist[][])
{
    System.out.println("Following matrix shows the shortest "+
        "distances between every pair of vertices");
    for (int i=0;i<V; ++i)
    {
        for (int j=0; j<V; ++j)
        {
            if (dist[i][j]==INF)
                System.out.print("INF ");
            else
                System.out.print(dist[i][j]+ " ");
        }
        System.out.println();
    }
}

// Driver program to test above function
public static void main (String[] args)
{
    /* Let us create the following weighted graph
     10
     (0)----->(3)
     |       \
     5 |       |
     |       | 1
     \|       |
     (1)----->(2)
     3       */
    int graph[][] = { {0, 5, INF, 10},
                     {INF, 0, 3, INF},
                     {INF, INF, 0, 1},
                     {INF, INF, INF, 0}
                 };
    AllPairShortestPath a = new AllPairShortestPath();

    // Print the solution
    a.floydWarshall(graph);
}
}

// Contributed by Aakash Hasija

```

Python

```

# Python Program for Floyd Warshall Algorithm

# Number of vertices in the graph
V = 4

# Define infinity as the large enough value. This value will be
# used for vertices not connected to each other
INF = 99999

# Solves all pair shortest path via Floyd Warshall Algorithm

```

```

def floydWarshall(graph):

    """ dist[] will be the output matrix that will finally
        have the shortest distances between every pair of vertices """
    """ initializing the solution matrix same as input graph matrix
    OR we can say that the initial values of shortest distances
    are based on shortest paths considering no
    intermediate vertices """
    dist = map(lambda i : map(lambda j : j , i) , graph)

    """ Add all vertices one by one to the set of intermediate
    vertices.

    --> Before start of a iteration, we have shortest distances
    between all pairs of vertices such that the shortest
    distances consider only the vertices in set
    {0, 1, 2, .. k-1} as intermediate vertices.

    ---> After the end of a iteration, vertex no. k is
    added to the set of intermediate vertices and the
    set becomes {0, 1, 2, .. k}

    """
    for k in range(V):

        # pick all vertices as source one by one
        for i in range(V):

            # Pick all vertices as destination for the
            # above picked source
            for j in range(V):

                # If vertex k is on the shortest path from
                # i to j, then update the value of dist[i][j]
                dist[i][j] = min(dist[i][j] ,
                                  dist[i][k]+ dist[k][j]
                )
    printSolution(dist)

# A utility function to print the solution
def printSolution(dist):
    print "Following matrix shows the shortest distances\
between every pair of vertices"
    for i in range(V):
        for j in range(V):
            if(dist[i][j] == INF):
                print "%7s"%(INF),
            else:
                print "%7d\n" %(dist[i][j]),
            if j == V-1:
                print ""

# Driver program to test the above program
# Let us create the following weighted graph
"""

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

graph = [[0,5,INF,10],
          [INF,0,3,INF],
          [INF, INF, 0, 1],
          [INF, INF, INF, 0]
         ]
# Print the solution
floydWarshall(graph);
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

Following matrix shows the shortest distances between every pair of vertices

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

Time Complexity: O(V^3)

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.

Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

```
#include <limits.h>

#define INF INT_MAX
.....
if ( dist[i][k] != INF &&
    dist[k][j] != INF &&
    dist[i][k] + dist[k][j] < dist[i][j]
)
dist[i][j] = dist[i][k] + dist[k][j];
.....
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Graph
shortest path

Union-Find Algorithm | Set 1 (Detect Cycle in an Undirected Graph)

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

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

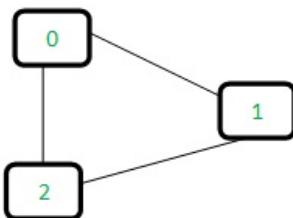
Union: Join two subsets into a single subset.

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

Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an [algorithm to detect cycle](#). This is another method based on *Union-Find*. This method assumes that graph doesn't contain any self-loops.

We can keep track of the subsets in a 1D array, let's call it parent[].

Let us consider the following graph:



For each edge, make subsets using both the vertices of the edge. If both the vertices are in the same subset, a cycle is found.

Initially, all slots of parent array are initialized to -1 (means there is only one item in every subset).

0	1	2
-1	-1	-1

Now process all edges one by one.

Edge 0-1: Find the subsets in which vertices 0 and 1 are. Since they are in different subsets, we take the union of them. For taking the union, either make node 0 as parent of node 1 or vice-versa.

```
0 1 2
```

Edge 1-2: 1 is in subset 1 and 2 is in subset 2. So, take union.

```
0 1 2
```

Edge 0-2: 0 is in subset 2 and 2 is also in subset 2. Hence, including this edge forms a cycle.

How subset of 0 is same as 2?

```
0->1->2 // 1 is parent of 0 and 2 is parent of 1
```

Based on the above explanation, below are implementations:

C/C++

```
// A union-find algorithm to detect cycle in a graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent an edge in graph
struct Edge
{
    int src, dest;
};

// a structure to represent a graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph =
        (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge =
        (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A utility function to find the subset of an element i
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
    parent[xset] = yset;
}

// The main function to check whether a given graph contains
```

```

// cycle or not
int isCycle( struct Graph* graph )
{
    // Allocate memory for creating V subsets
    int *parent = (int*) malloc( graph->V * sizeof(int) );

    // Initialize all subsets as single element sets
    memset(parent, -1, sizeof(int) * graph->V);

    // Iterate through all edges of graph, find subset of both
    // vertices of every edge, if both subsets are same, then
    // there is cycle in graph.
    for(int i = 0; i < graph->E; ++i)
    {
        int x = find(parent, graph->edge[i].src);
        int y = find(parent, graph->edge[i].dest);

        if (x == y)
            return 1;

        Union(parent, x, y);
    }
    return 0;
}

```

// Driver program to test above functions

```

int main()
{
    /* Let us create following graph
     0
     | \
     | \
     1---2 */
    int V = 3, E = 3;
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;

    // add edge 1-2
    graph->edge[1].src = 1;
    graph->edge[1].dest = 2;

    // add edge 0-2
    graph->edge[2].src = 0;
    graph->edge[2].dest = 2;

    if (isCycle(graph))
        printf( "graph contains cycle" );
    else
        printf( "graph doesn't contain cycle" );

    return 0;
}

```

Java

```

// Java Program for union-find algorithm to detect cycle in a graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
    int V, E; // V-> no. of vertices & E->no.of edges
    Edge edge[]; // /collection of all edges
}

```

```

class Edge
{
    int src, dest;
};

// Creates a graph with V vertices and E edges
Graph(int v,int e)
{
    V = v;
    E = e;
    edge = new Edge[E];
    for (int i=0; i<e; ++i)
        edge[i] = new Edge();
}

// A utility function to find the subset of an element
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
    parent[xset] = yset;
}

// The main function to check whether a given graph
// contains cycle or not
int isCycle( Graph graph)
{
    // Allocate memory for creating V subsets
    int parent[] = new int[graph.V];

    // Initialize all subsets as single element sets
    for (int i=0; i<graph.V; ++i)
        parent[i]=-1;

    // Iterate through all edges of graph, find subset of both
    // vertices of every edge, if both subsets are same, then
    // there is cycle in graph.
    for (int i = 0; i < graph.E; ++i)
    {
        int x = graph.find(parent, graph.edge[i].src);
        int y = graph.find(parent, graph.edge[i].dest);

        if (x == y)
            return 1;

        graph.Union(parent, x, y);
    }
    return 0;
}

// Driver Method
public static void main (String[] args)
{
    /* Let us create following graph
     0
     | \
     | \
     1---2 */
    int V = 3, E = 3;
    Graph graph = new Graph(V, E);

    // add edge 0-1
    graph.edge[0].src = 0;
    graph.edge[0].dest = 1;

    // add edge 1-2
}

```

```

graph.edge[1].src = 1;
graph.edge[1].dest = 2;

// add edge 0-2
graph.edge[2].src = 0;
graph.edge[2].dest = 2;

if (graph.isCycle(graph)==1)
    System.out.println( "graph contains cycle" );
else
    System.out.println( "graph doesn't contain cycle" );
}
}

```

Python

```

# Python Program for union-find algorithm to detect cycle in a undirected graph
# we have one egde for any two vertex i.e 1-2 is either 1-2 or 2-1 but not both

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A utility function to find the subset of an element i
    def find_parent(self, parent,i):
        if parent[i] == -1:
            return i
        if parent[i]!= -1:
            return self.find_parent(parent,parent[i])

    # A utility function to do union of two subsets
    def union(self,parent,x,y):
        x_set = self.find_parent(parent, x)
        y_set = self.find_parent(parent, y)
        parent[x_set] = y_set

    # The main function to check whether a given graph
    # contains cycle or not
    def isCyclic(self):

        # Allocate memory for creating V subsets and
        # Initialize all subsets as single element sets
        parent = [-1]*(self.V)

        # Iterate through all edges of graph, find subset of both
        # vertices of every edge, if both subsets are same, then
        # there is cycle in graph.
        for i in self.graph:
            for j in self.graph[i]:
                x = self.find_parent(parent, i)
                y = self.find_parent(parent, j)
                if x == y:
                    return True
                self.union(parent,x,y)

```

```
# Create a graph given in the above diagram
g = Graph(3)
g.addEdge(0, 1)
g.addEdge(1, 2)
g.addEdge(2, 0)

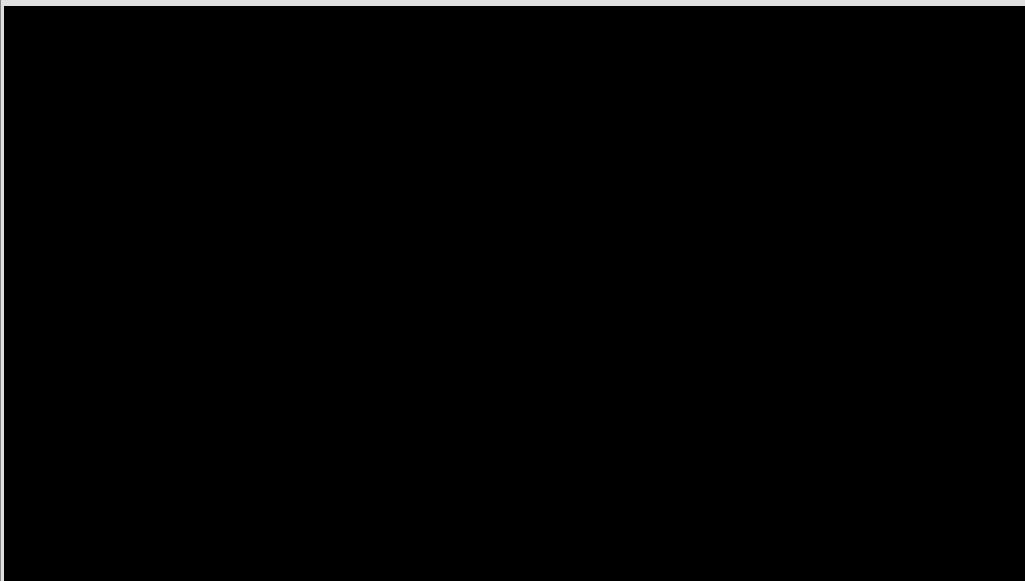
if g.isCyclic():
    print "Graph contains cycle"
else :
    print "Graph does not contain cycle "

#This code is contributed by Neelam Yadav
```

Output:

```
graph contains cycle
```

Note that the implementation of *union()* and *find()* is naive and takes $O(n)$ time in worst case. These methods can be improved to $O(\log n)$ using *Union by*



Related Articles :

[Union-Find Algorithm | Set 2 \(Union By Rank and Path Compression\)](#)

[Disjoint Set Data Structures \(Java Implementation\)](#)

[Greedy Algorithms | Set 2 \(Kruskal's Minimum Spanning Tree Algorithm\)](#)

[Job Sequencing Problem | Set 2 \(Using Disjoint Set\)](#)

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to s

GATE CS Corner

Company Wise Coding Practice

Graph
Graph
union-find

Greedy Algorithms | Set 5 (Prim's Minimum Spanning Tree (MST))

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

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 the cut and include this vertex to MST Set (the set that contains already included vertices).

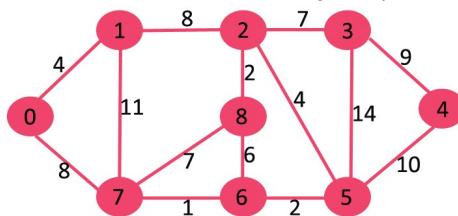
How does Prim's Algorithm Work? The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a **Spanning Tree**. And they must be connected with the minimum weight edge to make it a **Minimum Spanning Tree**.

Algorithm

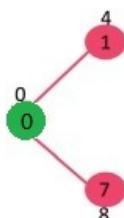
- 1) Create a set *mstSet* that keeps track of vertices already included in MST.
- 2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
- 3) While *mstSet* doesn't include all vertices
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

The idea of using key values is to pick the minimum weight edge from **cut**. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

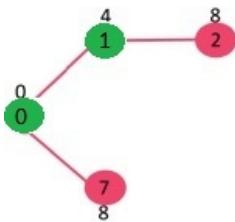
Let us understand with the following example:



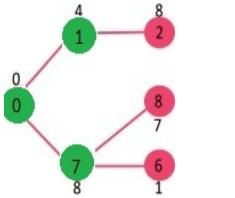
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key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



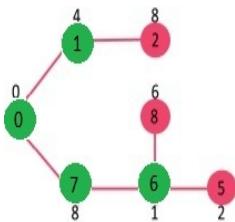
Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). The vertex 1 is picked and added to *mstSet*. So *mstSet* now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



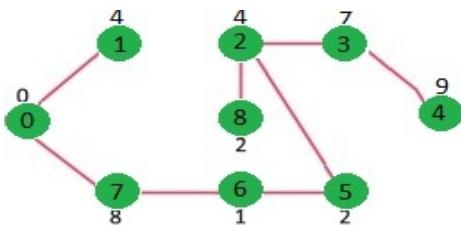
Pick the vertex with minimum key value and not already included in MST (not in *mstSet*). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So *mstSet* now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).



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



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



How to implement the above algorithm?

We use a boolean array *mstSet[]* to represent the set of vertices included in MST. If a value *mstSet[v]* is true, then vertex v is included in MST, otherwise not. Array *key[]* is used to store key values of all vertices. Another array *parent[]* to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

C/C++

```
// A C / C++ program for Prim's Minimum Spanning Tree (MST) algorithm.
// The program is for adjacency matrix representation of the graph

#include <stdio.h>
#include <limits.h>

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

// A utility function to find the vertex with minimum key value, from
// the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}
```

```

// A utility function to print the constructed MST stored in parent[]
int printMST(int parent[], int n, int graph[V][V])
{
    printf("Edge  Weight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d  %d \n", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST for a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    int parent[V]; // Array to store constructed MST
    int key[V]; // Key values used to pick minimum weight edge in cut
    bool mstSet[V]; // To represent set of vertices not yet included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    key[0] = 0; // Make key 0 so that this vertex is picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V-1; count++)
    {
        // Pick the minimum key vertex from the set of vertices
        // not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of the adjacent vertices of
        // the picked vertex. Consider only those vertices which are not yet
        // included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, V, graph);
}

// driver program to test above function
int main()
{
    /* Let us create the following graph
     2 3
     (0)--(1)--(2)
     |  / \ |
     6| 8 / \ 5 |7
     | /   \ |
     (3)-----(4)
     9       */
    int graph[V][V] = {{0, 2, 0, 6, 0},
                       {2, 0, 3, 8, 5},
                       {0, 3, 0, 0, 7},
                       {6, 8, 0, 0, 9},
                       {0, 5, 7, 9, 0},
                       };

    // Print the solution
    primMST(graph);

    return 0;
}

```

Java

```
// A Java program for Prim's Minimum Spanning Tree (MST) algorithm.  
// The program is for adjacency matrix representation of the graph  
  
import java.util.*;  
import java.lang.*;  
import java.io.*;  
  
class MST  
{  
    // Number of vertices in the graph  
    private static final int V=5;  
  
    // A utility function to find the vertex with minimum key  
    // value, from the set of vertices not yet included in MST  
    int minKey(int key[], Boolean mstSet[])  
    {  
        // Initialize min value  
        int min = Integer.MAX_VALUE, min_index=-1;  
  
        for (int v = 0; v < V; v++)  
            if (mstSet[v] == false && key[v] < min)  
            {  
                min = key[v];  
                min_index = v;  
            }  
  
        return min_index;  
    }  
  
    // A utility function to print the constructed MST stored in  
    // parent[]  
    void printMST(int parent[], int n, int graph[][])  
    {  
        System.out.println("Edge  Weight");  
        for (int i = 1; i < V; i++)  
            System.out.println(parent[i]+ " - "+ i+ "  "+  
                               graph[i][parent[i]]);  
    }  
  
    // Function to construct and print MST for a graph represented  
    // using adjacency matrix representation  
    void primMST(int graph[][])  
    {  
        // Array to store constructed MST  
        int parent[] = new int[V];  
  
        // Key values used to pick minimum weight edge in cut  
        int key[] = new int[V];  
  
        // To represent set of vertices not yet included in MST  
        Boolean mstSet[] = new Boolean[V];  
  
        // Initialize all keys as INFINITE  
        for (int i = 0; i < V; i++)  
        {  
            key[i] = Integer.MAX_VALUE;  
            mstSet[i] = false;  
        }  
  
        // Always include first 1st vertex in MST.  
        key[0] = 0; // Make key 0 so that this vertex is  
                    // picked as first vertex  
        parent[0] = -1; // First node is always root of MST  
  
        // The MST will have V vertices  
        for (int count = 0; count < V-1; count++)  
        {  
            // Pick thd minimum key vertex from the set of vertices  
            // not yet included in MST  
            int u = minKey(key, mstSet);  
  
            // Add the picked vertex to the MST Set
```

```

mstSet[u] = true;

// Update key value and parent index of the adjacent
// vertices of the picked vertex. Consider only those
// vertices which are not yet included in MST
for (int v = 0; v < V; v++)

    // graph[u][v] is non zero only for adjacent vertices of m
    // mstSet[v] is false for vertices not yet included in MST
    // Update the key only if graph[u][v] is smaller than key[v]
    if (graph[u][v]!=0 && mstSet[v] == false &&
        graph[u][v] < key[v])
    {
        parent[v] = u;
        key[v] = graph[u][v];
    }
}

// print the constructed MST
printMST(parent, V, graph);
}

public static void main (String[] args)
{
    /* Let us create the following graph
     2 3
     (0)--(1)--(2)
     |  \ |
     6| 8/ \5 |7
     | /   \| |
     (3)-----(4)
     9           */
    MST t = new MST();
    int graph[][] = new int[][] {{0, 2, 0, 6, 0},
                                {2, 0, 3, 8, 5},
                                {0, 3, 0, 0, 7},
                                {6, 8, 0, 0, 9},
                                {0, 5, 7, 9, 0},
                                };
}

// Print the solution
t.primMST(graph);
}
}

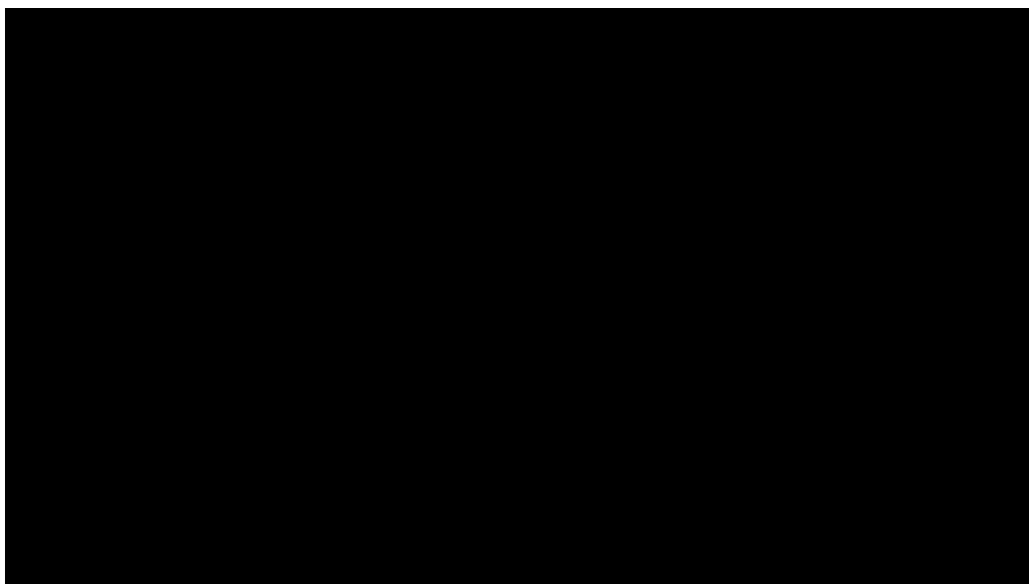
// This code is contributed by Aakash Hasija

```

Output:

Edge	Weight
0 - 1	2
1 - 2	3
0 - 3	6
1 - 4	5

Time Complexity of the above program 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. Please see [Prim's MST for Adjacency List Representation](#) for more details.



Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
Greedy
Greedy Algorithm
Minimum Spanning Tree
Prim's Algorithm.MST

Greedy Algorithms | Set 2 (Kruskal's Minimum Spanning Tree Algorithm)

What is Minimum Spanning Tree?

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

How many edges does a minimum spanning tree has?

A minimum spanning tree has $(V - 1)$ edges where V is the number of vertices in the given graph.

What are the applications of Minimum Spanning Tree?

See [this](#) for applications of MST.

Below are the steps for finding MST using Kruskal's algorithm

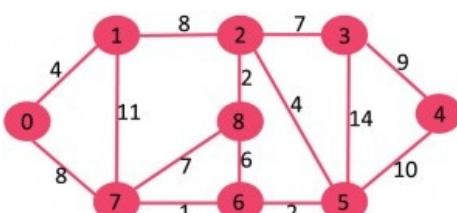
1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are $(V-1)$ edges in the spanning tree.

The step#2 uses [Union-Find algorithm](#) to detect cycle. So we recommend to read following post as a prerequisite.

[Union-Find Algorithm | Set 1 \(Detect Cycle in a Graph\)](#)

[Union-Find Algorithm | Set 2 \(Union By Rank and Path Compression\)](#)

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will have 8 edges.

After sorting:

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

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

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



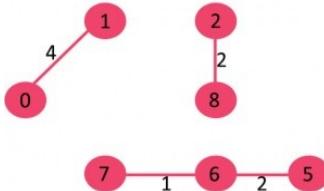
2. Pick edge 8-2: No cycle is formed, include it.



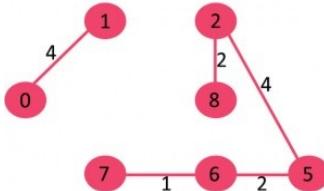
3. Pick edge 6-5: No cycle is formed, include it.



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

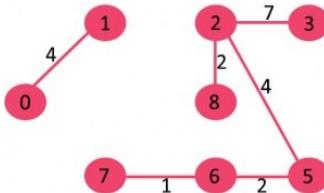


5. Pick edge 2-5: No cycle is formed, include it.



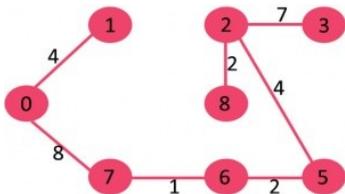
6. Pick edge 8-6: Since including this edge results in cycle, discard it.

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



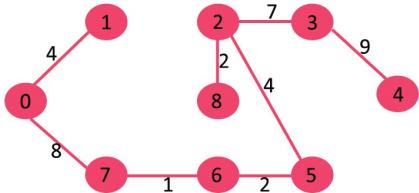
8. Pick edge 7-8: Since including this edge results in cycle, discard it.

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



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

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



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

C/C++

```
// C++ program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, undirected and weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges. Since the graph is
    // undirected, the edge from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
}
```

```

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges

    // Step 1: Sort all the edges in non-decreasing order of their weight
    // If we are not allowed to change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    // Allocate memory for creating V subsets
    struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1)
    {
        // Step 2: Pick the smallest edge. And increment the index
        // for next iteration
        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge doesn't cause cycle, include it
        // in result and increment the index of result for next edge
        if (x != y)
        {
            result[e++] = next_edge;

```

```

        Union(subsets, x, y);
    }
    // Else discard the next_edge
}

// print the contents of result[] to display the built MST
printf("Following are the edges in the constructed MST\n");
for (i = 0; i < e; ++i)
    printf("%d -- %d == %d\n", result[i].src, result[i].dest,
           result[i].weight);
return;
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
       10
       0-----1
       | \   |
       6|  5\  |15
       |   \ |
       2-----3
       4     */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;
}

```

Java

```

// Java program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
    // A class to represent a graph edge
    class Edge implements Comparable<Edge>
    {
        int src, dest, weight;

```

```

// Comparator function used for sorting edges based on
// their weight
public int compareTo(Edge compareEdge)
{
    return this.weight - compareEdge.weight;
}

// A class to represent a subset for union-find
class subset
{
    int parent, rank;
};

int V, E; // V-> no. of vertices & E->no.of edges
Edge edge[]; // collection of all edges

// Creates a graph with V vertices and E edges
Graph(int v, int e)
{
    V = v;
    E = e;
    edge = new Edge[E];
    for (int i=0; i<e; ++i)
        edge[i] = new Edge();
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST()
{
    Edge result[] = new Edge[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges
    for (i=0; i<V; ++i)
        result[i] = new Edge();

    // Step 1: Sort all the edges in non-decreasing order of their
    // weight. If we are not allowed to change the given graph, we
    // can create a copy of array of edges
    Arrays.sort(edge);
}

```

```

// Allocate memory for creating V subsets
subset subsets[] = new subset[V];
for(i=0; i<V; ++i)
    subsets[i]=new subset();

// Create V subsets with single elements
for (int v = 0; v < V; ++v)
{
    subsets[v].parent = v;
    subsets[v].rank = 0;
}

i = 0; // Index used to pick next edge

// Number of edges to be taken is equal to V-1
while (e < V - 1)
{
    // Step 2: Pick the smallest edge. And increment the index
    // for next iteration
    Edge next_edge = new Edge();
    next_edge = edge[i++];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);

    // If including this edge doesn't cause cycle, include it
    // in result and increment the index of result for next edge
    if (x != y)
    {
        result[e++] = next_edge;
        Union(subsets, x, y);
    }
    // Else discard the next_edge
}

// print the contents of result[] to display the built MST
System.out.println("Following are the edges in the constructed MST");
for (i = 0; i < e; ++i)
    System.out.println(result[i].src+" -- "+result[i].dest+" == "+
                       result[i].weight);
}

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

/* Let us create following weighted graph
   10
   0-----1
   | \   |
   6|  5\  |15
   |   \| |
   2-----3
   4     */
int V = 4; // Number of vertices in graph
int E = 5; // Number of edges in graph
Graph graph = new Graph(V, E);

// add edge 0-1
graph.edge[0].src = 0;
graph.edge[0].dest = 1;
graph.edge[0].weight = 10;

// add edge 0-2
graph.edge[1].src = 0;
graph.edge[1].dest = 2;
graph.edge[1].weight = 6;

// add edge 0-3
graph.edge[2].src = 0;
graph.edge[2].dest = 3;
graph.edge[2].weight = 5;

// add edge 1-3

```

```

graph.edge[3].src = 1;
graph.edge[3].dest = 3;
graph.edge[3].weight = 15;

// add edge 2-3
graph.edge[4].src = 2;
graph.edge[4].dest = 3;
graph.edge[4].weight = 4;

graph.KruskalMST();
}
}

//This code is contributed by Aakash Hasija

```

Python

```

# Python program for Kruskal's algorithm to find Minimum Spanning Tree
# of a given connected, undirected and weighted graph

from collections import defaultdict

#Class to represent a graph
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = {} # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,u,v,w):
        self.graph[u] = self.graph.get(u,[])
        self.graph[u].append([v,w])

    # A utility function to find set of an element i
    # (uses path compression technique)
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    # A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of high rank tree
        # (Union by Rank)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        #If ranks are same, then make one as root and increment
        # its rank by one
        else :
            parent[yroot] = xroot
            rank[xroot] += 1

    # The main function to construct MST using Kruskal's algorithm
    def KruskalMST(self):

        result =[] #This will store the resultant MST

        i = 0 # An index variable, used for sorted edges
        e = 0 # An index variable, used for result[]

        #Step 1: Sort all the edges in non-decreasing order of their
        # weight. If we are not allowed to change the given graph, we
        # can create a copy of graph
        self.graph = sorted(self.graph,key=lambda item: item[2])
        #print self.graph

        parent = [] ; rank = []

```

```

# Create V subsets with single elements
for node in range(self.V):
    parent.append(node)
    rank.append(0)

# Number of edges to be taken is equal to V-1
while e < self.V - 1 :

    # Step 2: Pick the smallest edge and increment the index
        # for next iteration
    u,v,w = self.graph[i]
    i = i + 1
    x = self.find(parent, u)
    y = self.find(parent, v)

    # If including this edge does't cause cycle, include it
        # in result and increment the index of result for next edge
    if x != y:
        e = e + 1
        result.append([u,v,w])
        self.union(parent, rank, x, y)
    # Else discard the edge

# print the contents of result[] to display the built MST
print "Following are the edges in the constructed MST"
for u,v,weight in result:
    #print str(u) + " -- " + str(v) + " == " + str(weight)
    print ("%d -- %d == %d" % (u,v,weight))

g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

g.KruskalMST()

```

#This code is contributed by Neelam Yadav

Following are the edges in the constructed MST

2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10

Time Complexity: $O(E\log E)$ or $O(E\log V)$. Sorting of edges takes $O(E\log E)$ time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost $O(\log V)$ time. So overall complexity is $O(E\log E + E\log V)$ time. The value of E can be atmost $O(V^2)$, so $O(\log V)$ are $O(\log E)$ same. Therefore, overall time complexity is $O(E\log E)$ or $O(E\log V)$

References:

<http://www.ics.uci.edu/~eppstein/161/960206.html>
http://en.wikipedia.org/wiki/Minimum_spanning_tree

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

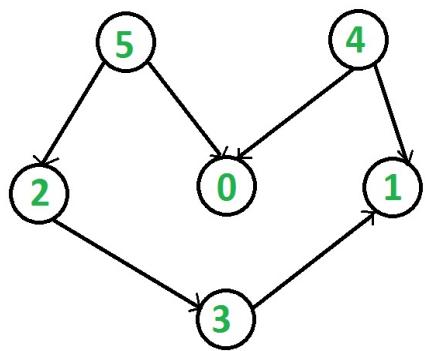
Graph
Greedy
Graph
Greedy Algorithm
Kruskal
Kruskal'sAlgorithm
MST

Topological Sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For

example, another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).



Topological Sorting vs Depth First Traversal (DFS):

In [DFS](#), we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike [DFS](#), the vertex '4' should also be printed before vertex '0'. So Topological sorting is different from DFS. For example, a DFS of the above graph is "5 2 3 1 0 4", but it is not a topological sorting

Algorithm to find Topological Sorting:

We recommend to first see implementation of DFS [here](#). We can modify [DFS](#) to find Topological Sorting of a graph. In [DFS](#), we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Following are C++ and Java implementations of topological sorting. Please see the code for [Depth First Traversal for a disconnected Graph](#) and note the differences between the second code given there and the below code.

C++

```

// A C++ program to print topological sorting of a DAG
#include<iostream>
#include <list>
#include <stack>
using namespace std;

// Class to represent a graph
class Graph
{
    int V; // No. of vertices

    // Pointer to an array containing adjacency listsList
    list<int> *adj;

    // A function used by topologicalSort
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);

public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints a Topological Sort of the complete graph
    void topologicalSort();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// A recursive function used by topologicalSort
  
```

```

void Graph::topologicalSortUtil(int v, bool visited[],
                                stack<int> &Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // 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])
            topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack which stores result
    Stack.push(v);
}

// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
void Graph::topologicalSort()
{
    stack<int> Stack;

    // 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 store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Print contents of stack
    while (Stack.empty() == false)
    {
        cout << Stack.top() << " ";
        Stack.pop();
    }
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    cout << "Following is a Topological Sort of the given graph \n";
    g.topologicalSort();

    return 0;
}

```

Java

```

// A Java program to print topological sorting of a DAG
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    private int V; // No. of vertices
    private LinkedList<Integer> adj[]; // Adjacency List

    // Constructor

```

```

Graph(int v)
{
    V = v;
    adj = new LinkedList[v];
    for (int i=0; i<v; ++i)
        adj[i] = new LinkedList();
}

// Function to add an edge into the graph
void addEdge(int v,int w) { adj[v].add(w);}

// A recursive function used by topologicalSort
void topologicalSortUtil(int v, boolean visited[],
                         Stack stack)
{
    // Mark the current node as visited.
    visited[v] = true;
    Integer i;

    // Recur for all the vertices adjacent to this
    // vertex
    Iterator<Integer> it = adj[v].iterator();
    while (it.hasNext())
    {
        i = it.next();
        if (!visited[i])
            topologicalSortUtil(i, visited, stack);
    }

    // Push current vertex to stack which stores result
    stack.push(new Integer(v));
}

// The function to do Topological Sort. It uses
// recursive topologicalSortUtil()
void topologicalSort()
{
    Stack stack = new Stack();

    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store
    // Topological Sort starting from all vertices
    // one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, stack);

    // Print contents of stack
    while (stack.empty()==false)
        System.out.print(stack.pop() + " ");
}

// Driver method
public static void main(String args[])
{
    // Create a graph given in the above diagram
    Graph g = new Graph(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    System.out.println("Following is a Topological " +
                       "sort of the given graph");
    g.topologicalSort();
}
}

// This code is contributed by Aakash Hasija

```

Python

```
#Python program to print topological sorting of a DAG
from collections import defaultdict

#Class to represent a graph
class Graph:
    def __init__(self,vertices):
        self.graph = defaultdict(list) #dictionary containing adjacency List
        self.V = vertices #No. of vertices

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A recursive function used by topologicalSort
    def topologicalSortUtil(self,v,visited,stack):

        # Mark the current node as visited.
        visited[v] = True

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.topologicalSortUtil(i,visited,stack)

        # Push current vertex to stack which stores result
        stack.insert(0,v)

    # The function to do Topological Sort. It uses recursive
    # topologicalSortUtil()
    def topologicalSort(self):
        # Mark all the vertices as not visited
        visited = [False]*self.V
        stack = []

        # Call the recursive helper function to store Topological
        # Sort starting from all vertices one by one
        for i in range(self.V):
            if visited[i] == False:
                self.topologicalSortUtil(i,visited,stack)

        # Print contents of stack
        print stack

g= Graph(6)
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);

print "Following is a Topological Sort of the given graph"
g.topologicalSort()
#This code is contributed by Neelam Yadav
```

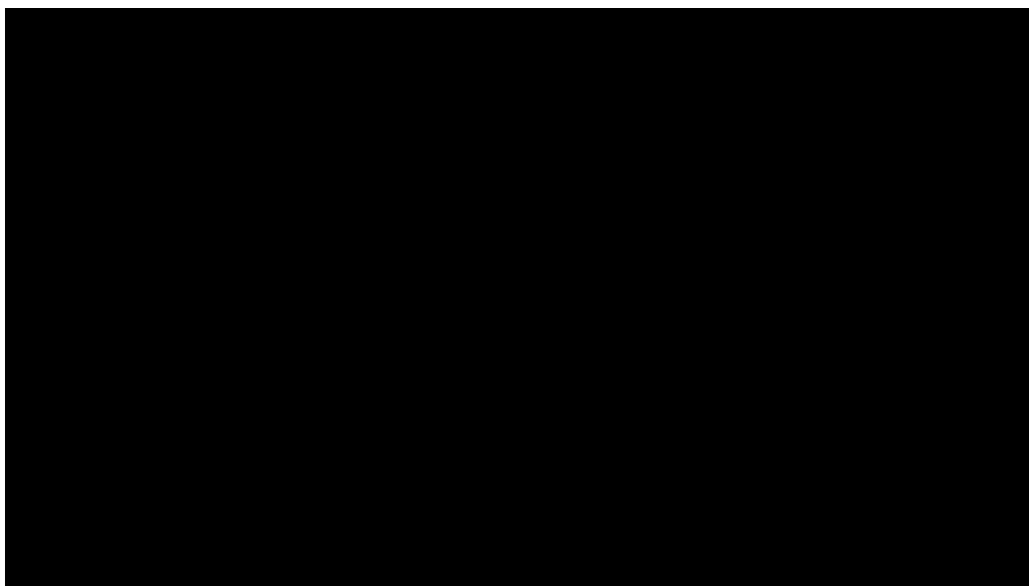
Output:

```
Following is a Topological Sort of the given graph
5 4 2 3 1 0
```

Time Complexity: The above algorithm is simply DFS with an extra stack. So time complexity is same as DFS which is $O(V+E)$.

Applications:

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

**Related Articles:**

Kahn's algorithm for Topological Sorting : Another O(V + E) algorithm.

All Topological Sorts of a Directed Acyclic Graph

References:

<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/topoSort.htm>

http://en.wikipedia.org/wiki/Topological_sorting

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Graph
DFS
Topological Sorting

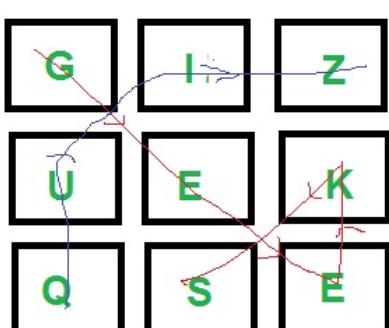
Boggle (Find all possible words in a board of characters) | Set 1

Given a dictionary, a method to do lookup in dictionary and a M x N board where every cell has one character. Find all possible words that can be formed by a sequence of adjacent characters. Note that we can move to any of 8 adjacent characters, but a word should not have multiple instances of same cell.

Example:

```
Input: dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
      boggle[][] = {{'G','I','Z'},
                    {'U','E','K'},
                    {'Q','S','E'}};
      isWord(str): returns true if str is present in dictionary
                  else false.
```

```
Output: Following words of dictionary are present
      GEEKS
      QUIZ
```



We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

The idea is to consider every character as a starting character and find all words starting with it. All words starting from a character can be found using [Depth First Traversal](#). We do depth first traversal starting from every cell. We keep track of visited cells to make sure that a cell is considered only once in a word.

```
// C++ program for Boggle game
#include<iostream>
#include<cstring>
using namespace std;

#define M 3
#define N 3

// Let the given dictionary be following
string dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
int n = sizeof(dictionary)/sizeof(dictionary[0]);

// A given function to check if a given string is present in
// dictionary. The implementation is naive for simplicity. As
// per the question dictionary is given to us.
bool isWord(string &str)
{
    // Linearly search all words
    for (int i=0; i<n; i++)
        if (str.compare(dictionary[i]) == 0)
            return true;
    return false;
}

// A recursive function to print all words present on boggle
void findWordsUtil(char boggle[M][N], bool visited[M][N], int i,
                   int j, string &str)
{
    // Mark current cell as visited and append current character
    // to str
    visited[i][j] = true;
    str = str + boggle[i][j];

    // If str is present in dictionary, then print it
    if (isWord(str))
        cout << str << endl;

    // Traverse 8 adjacent cells of boggle[i][j]
    for (int row=i-1; row<=i+1 && row<M; row++)
        for (int col=j-1; col<=j+1 && col<N; col++)
            if (row>=0 && col>=0 && !visited[row][col])
                findWordsUtil(boggle, visited, row, col, str);

    // Erase current character from string and mark visited
    // of current cell as false
    str.erase(str.length()-1);
    visited[i][j] = false;
}

// Prints all words present in dictionary.
void findWords(char boggle[M][N])
{
    // Mark all characters as not visited
    bool visited[M][N] = {{false}};

    // Initialize current string
    string str = "";

    // Consider every character and look for all words
    // starting with this character
    for (int i=0; i<M; i++)
        for (int j=0; j<N; j++)
            findWordsUtil(boggle, visited, i, j, str);
}

// Driver program to test above function
int main()
{
    char boggle[M][N] = {{'G','I','Z'},
                        {'U','E','K'},
```

```

{'Q','S','E'});
cout << "Following words of dictionary are present\n";
findWords(boggle);
return 0;
}

```

Output:

```

Following words of dictionary are present
GEEKS
QUIZ

```

Note that the above solution may print same word multiple times. For example, if we add “SEEK” to dictionary, it is printed multiple times. To avoid this, we can use hashing to keep track of all printed words.

In below set 2, we have discussed Trie based optimized solution:

Boggle | Set 2 (Using Trie)

This article is contributed by **Rishabh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

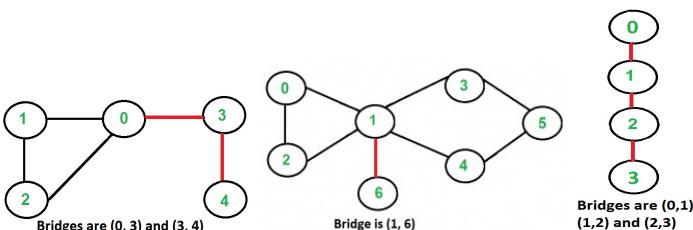
Graph
DFS

Bridges in a graph

An edge in an undirected connected graph is a bridge iff removing it disconnects the graph. For a disconnected undirected graph, definition is similar, a bridge is an edge removing which increases number of connected components.

Like [Articulation Points](#), bridges represent vulnerabilities in a connected network and are useful for designing reliable networks. For example, in a wired computer network, an articulation point indicates the critical computers and a bridge indicates the critical wires or connections.

Following are some example graphs with bridges highlighted with red color.



How to find all bridges in a given graph?

A simple approach is to one by one remove all edges and see if removal of a edge causes disconnected graph. Following are steps of simple approach for connected graph.

- 1) For every edge (u, v) , do following
 -a) Remove (u, v) from graph
 -b) See if the graph remains connected (We can either use BFS or DFS)
 -c) Add (u, v) back to the graph.

Time complexity of above method is $O(E^*(V+E))$ for a graph represented using adjacency list. Can we do better?

A $O(V+E)$ algorithm to find all Bridges

The idea is similar to [O\(V+E\) algorithm for Articulation Points](#). We do DFS traversal of the given graph. In DFS tree an edge (u, v) (u is parent of v in DFS tree) is bridge if there does not exist any other alternative to reach u or an ancestor of u from subtree rooted with v . As discussed in the [previous post](#), the value $low[v]$ indicates earliest visited vertex reachable from subtree rooted with v . *The condition for an edge (u, v) to be a bridge is, $low[v] > disc[u]$.*

Following are C++ and Java implementations of above approach.

C++

```

// A C++ program to find bridges in a given undirected graph
#include<iostream>
#include <list>

```

```

#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V; // No. of vertices
    list<int> *adj; // A dynamic array of adjacency lists
    void bridgeUtil(int v, bool visited[], int disc[], int low[],
                    int parent[]);
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    void bridge(); // prints all bridges
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the graph is undirected
}

// A recursive function that finds and prints bridges using
// DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
void Graph::bridgeUtil(int u, bool visited[], int disc[],
                      int low[], int parent[])
{
    // A static variable is used for simplicity, we can
    // avoid use of static variable by passing a pointer.
    static int time = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i; // v is current adjacent of u

        // If v is not visited yet, then recur for it
        if (!visited[v])
        {
            parent[v] = u;
            bridgeUtil(v, visited, disc, low, parent);

            // Check if the subtree rooted with v has a
            // connection to one of the ancestors of u
            low[u] = min(low[u], low[v]);
        }

        // If the lowest vertex reachable from subtree
        // under v is below u in DFS tree, then u-v
        // is a bridge
        if (low[v] > disc[u])
            cout << u << " " << v << endl;
    }

    // Update low value of u for parent function calls.
    else if (v != parent[u])
        low[u] = min(low[u], disc[v]);
}
}

```

```

// DFS based function to find all bridges. It uses recursive
// function bridgeUtil()
void Graph::bridge()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];

    // Initialize parent and visited arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
    }

    // Call the recursive helper function to find Bridges
    // in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            bridgeUtil(i, visited, disc, low, parent);
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    cout << "\nBridges in first graph \n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.bridge();

    cout << "\nBridges in second graph \n";
    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.bridge();

    cout << "\nBridges in third graph \n";
    Graph g3(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
    g3.bridge();

    return 0;
}

```

Java

```

// A Java program to find bridges in a given undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a undirected graph using adjacency list
// representation
class Graph
{
    private int V; // No. of vertices

```

```

// Array of lists for Adjacency List Representation
private LinkedList<Integer> adj[][];
int time = 0;
static final int NIL = -1;

// Constructor
Graph(int v)
{
    V = v;
    adj = new LinkedList[v];
    for (int i=0; i<v; ++i)
        adj[i] = new LinkedList();
}

// Function to add an edge into the graph
void addEdge(int v, int w)
{
    adj[v].add(w); // Add w to v's list.
    adj[w].add(v); // Add v to w's list
}

// A recursive function that finds and prints bridges
// using DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps track of visited vertices
// disc[] -> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
void bridgeUtil(int u, boolean visited[], int disc[],
                int low[], int parent[])
{

    // Count of children in DFS Tree
    int children = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices adjacent to this
    Iterator<Integer> i = adj[u].iterator();
    while (i.hasNext())
    {
        int v = i.next(); // v is current adjacent of u

        // If v is not visited yet, then make it a child
        // of u in DFS tree and recur for it.
        // If v is not visited yet, then recur for it
        if (!visited[v])
        {
            parent[v] = u;
            bridgeUtil(v, visited, disc, low, parent);

            // Check if the subtree rooted with v has a
            // connection to one of the ancestors of u
            low[u] = Math.min(low[u], low[v]);

            // If the lowest vertex reachable from subtree
            // under v is below u in DFS tree, then u-v is
            // a bridge
            if (low[v] > disc[u])
                System.out.println(u+" "+v);
        }
    }

    // Update low value of u for parent function calls.
    else if (v != parent[u])
        low[u] = Math.min(low[u], disc[v]);
}

// DFS based function to find all bridges. It uses recursive
// function bridgeUtil()
void bridge()

```

```

{
    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    int disc[] = new int[V];
    int low[] = new int[V];
    int parent[] = new int[V];

    // Initialize parent and visited, and ap(articulation point)
    // arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
    }

    // Call the recursive helper function to find Bridges
    // in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            bridgeUtil(i, visited, disc, low, parent);
}

public static void main(String args[])
{
    // Create graphs given in above diagrams
    System.out.println("Bridges in first graph ");
    Graph g1 = new Graph(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.bridge();
    System.out.println();

    System.out.println("Bridges in Second graph");
    Graph g2 = new Graph(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.bridge();
    System.out.println();

    System.out.println("Bridges in Third graph ");
    Graph g3 = new Graph(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
    g3.bridge();
}
}

// This code is contributed by Aakash Hasija

```

Python

```

# Python program to find bridges in a given undirected graph
#Complexity : O(V+E)

from collections import defaultdict

#This class represents an undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph
        self.Time = 0

```

```

# function to add an edge to graph
def addEdge(self,u,v):
    self.graph[u].append(v)
    self.graph[v].append(u)

"A recursive function that finds and prints bridges
using DFS traversal
u --> The vertex to be visited next
visited[] --> keeps tract of visited vertices
disc[] --> Stores discovery times of visited vertices
parent[] --> Stores parent vertices in DFS tree"
def bridgeUtil(self,u, visited, parent, low, disc):

    #Count of children in current node
    children =0

    # Mark the current node as visited and print it
    visited[u]= True

    # Initialize discovery time and low value
    disc[u] = self.Time
    low[u] = self.Time
    self.Time += 1

    #Recur for all the vertices adjacent to this vertex
    for v in self.graph[u]:
        # If v is not visited yet, then make it a child of u
        # in DFS tree and recur for it
        if visited[v] == False :
            parent[v] = u
            children += 1
            self.bridgeUtil(v, visited, parent, low, disc)

    # Check if the subtree rooted with v has a connection to
    # one of the ancestors of u
    low[u] = min(low[u], low[v])

    """ If the lowest vertex reachable from subtree
    under v is below u in DFS tree, then u-v is
    a bridge"""
    if low[v] > disc[u]:
        print ("%d %d" %(u,v))

    elif v != parent[u]: # Update low value of u for parent function calls.
        low[u] = min(low[u], disc[v])

# DFS based function to find all bridges. It uses recursive
# function bridgeUtil()
def bridge(self):

    # Mark all the vertices as not visited and Initialize parent and visited,
    # and ap(articulation point) arrays
    visited = [False] * (self.V)
    disc = [float("Inf")] * (self.V)
    low = [float("Inf")] * (self.V)
    parent = [-1] * (self.V)

    # Call the recursive helper function to find bridges
    # in DFS tree rooted with vertex 'i'
    for i in range(self.V):
        if visited[i] == False:
            self.bridgeUtil(i, visited, parent, low, disc)

    # Create a graph given in the above diagram
    g1 = Graph(5)
    g1.addEdge(1, 0)
    g1.addEdge(0, 2)
    g1.addEdge(2, 1)
    g1.addEdge(0, 3)
    g1.addEdge(3, 4)

```

```

print "Bridges in first graph "
g1.bridge()

g2 = Graph(4)
g2.addEdge(0, 1)
g2.addEdge(1, 2)
g2.addEdge(2, 3)
print "\nBridges in second graph "
g2.bridge()

g3 = Graph (7)
g3.addEdge(0, 1)
g3.addEdge(1, 2)
g3.addEdge(2, 0)
g3.addEdge(1, 3)
g3.addEdge(1, 4)
g3.addEdge(1, 6)
g3.addEdge(3, 5)
g3.addEdge(4, 5)
print "\nBridges in third graph "
g3.bridge()

```

#This code is contributed by Neelam Yadav

Output:

```

Bridges in first graph
3 4
0 3

Bridges in second graph
2 3
1 2
0 1

Bridges in third graph
1 6

```

Time Complexity: The above function is simple DFS with additional arrays. So time complexity is same as DFS which is $O(V+E)$ for adjacency list representation of graph.

References:

<https://www.cs.washington.edu/education/courses/421/04su/slides/artic.pdf>
<http://www.slideshare.net/TraianRebedea/algorithm-design-and-complexity-course-8>
http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L25-Connectivity.htm
<http://www.youtube.com/watch?v=bmyyxNyZKzI>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
graph-connectivity

Given a linked list which is sorted, how will you insert in sorted way

Algorithm:

Let input linked list is sorted in increasing order.

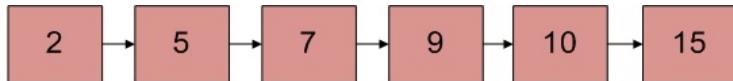
- 1) If Linked list is empty then make the node as head and return it.
- 2) If value of the node to be inserted is smaller than value of head node then insert the node at start and make it head.
- 3) In a loop, find the appropriate node after which the input node (let 9) is to be inserted. To find the appropriate node start from head, keep moving until you reach a node GN (10 in the below diagram) who's value is greater than the input node. The node just before GN is the appropriate

node (7).
4) Insert the node (9) after the appropriate node (7) found in step 3.

Initial Linked List



Linked List after insertion of 9



Implementation:

C/C++

```

/* Program to insert in a sorted list */
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* function to insert a new_node in a list. Note that this
function expects a pointer to head_ref as this can modify the
head of the input linked list (similar to push())*/
void sortedInsert(struct node** head_ref, struct node* new_node)
{
    struct node* current;
    /* Special case for the head end */
    if (*head_ref == NULL || (*head_ref)->data >= new_node->data)
    {
        new_node->next = *head_ref;
        *head_ref = new_node;
    }
    else
    {
        /* Locate the node before the point of insertion */
        current = *head_ref;
        while (current->next!=NULL &&
               current->next->data < new_node->data)
        {
            current = current->next;
        }
        new_node->next = current->next;
        current->next = new_node;
    }
}

/* BELOW FUNCTIONS ARE JUST UTILITY TO TEST sortedInsert */

/* A utility function to create a new node */
struct node *newNode(int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;
    new_node->next = NULL;

    return new_node;
}

/* Function to print linked list */
void printList(struct node *head)
  
```

```
{
    struct node *temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

/* Driver program to test count function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    struct node *new_node = newNode(5);
    sortedInsert(&head, new_node);
    new_node = newNode(10);
    sortedInsert(&head, new_node);
    new_node = newNode(7);
    sortedInsert(&head, new_node);
    new_node = newNode(3);
    sortedInsert(&head, new_node);
    new_node = newNode(1);
    sortedInsert(&head, new_node);
    new_node = newNode(9);
    sortedInsert(&head, new_node);
    printf("\n Created Linked List\n");
    printList(head);

    return 0;
}
```

Java

```
// Java Program to insert in a sorted list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    /* function to insert a new_node in a list. */
    void sortedInsert(Node new_node)
    {
        Node current;

        /* Special case for head node */
        if (head == null || head.data >= new_node.data)
        {
            new_node.next = head;
            head = new_node;
        }
        else {

            /* Locate the node before point of insertion.*/
            current = head;

            while (current.next != null &&
                   current.next.data < new_node.data)
                current = current.next;

            new_node.next = current.next;
            current.next = new_node;
        }
    }

    /*Utility functions*/
}
```

```

/* Function to create a node */
Node newNode(int data)
{
    Node x = new Node(data);
    return x;
}

/* Function to print linked list */
void printList()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
}

/* Driver function to test above methods */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();
    Node new_node;
    new_node = llist.newNode(5);
    llist.sortedInsert(new_node);
    new_node = llist.newNode(10);
    llist.sortedInsert(new_node);
    new_node = llist.newNode(7);
    llist.sortedInsert(new_node);
    new_node = llist.newNode(3);
    llist.sortedInsert(new_node);
    new_node = llist.newNode(1);
    llist.sortedInsert(new_node);
    new_node = llist.newNode(9);
    llist.sortedInsert(new_node);
    System.out.println("Created Linked List");
    llist.printList();
}
}

/* This code is contributed by Rajat Mishra */

```

Python

```

# Python program to insert in sorted list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def sortedInsert(self, new_node):

        # Special case for the empty linked list
        if self.head is None:
            new_node.next = self.head
            self.head = new_node

        # Special case for head at end
        elif self.head.data >= new_node.data:
            new_node.next = self.head
            self.head = new_node

```

```

else :

    # Locate the node before the point of insertion
    current = self.head
    while(current.next is not None and
        current.next.data < new_node.data):
        current = current.next

    new_node.next = current.next
    current.next = new_node

# Function to insert a new node at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program
llist = LinkedList()
new_node = Node(5)
llist.sortedInsert(new_node)
new_node = Node(10)
llist.sortedInsert(new_node)
new_node = Node(7)
llist.sortedInsert(new_node)
new_node = Node(3)
llist.sortedInsert(new_node)
new_node = Node(1)
llist.sortedInsert(new_node)
new_node = Node(9)
llist.sortedInsert(new_node)
print "Create Linked List"
llist.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

Created Linked List
1 3 5 7 9 10

Shorter Implementation using double pointers

Thanks to Murat M Ozturk for providing this solution. Please see Murat M Ozturk's comment below for complete function. The code uses double pointer to keep track of the next pointer of the previous node (after which new node is being inserted).

Note that below line in code changes *current* to have address of next pointer in a node.

```
current = &(*current)->next);
```

Also, note below comments.

```

/* Copies the value-at-address current to
new_node's next pointer*/
new_node->next = *current;

/* Fix next pointer of the node (using it's address)
after which new_node is being inserted */
*current = new_node;

```

Time Complexity: O(n)

References:

<http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

Delete a given node in Linked List under given constraints

Given a Singly Linked List, write a function to delete a given node. Your function must follow following constraints:

- 1) It must accept pointer to the start node as first parameter and node to be deleted as second parameter i.e., pointer to head node is not global.
- 2) It should not return pointer to the head node.
- 3) It should not accept pointer to pointer to head node.

You may assume that the Linked List never becomes empty.

Let the function name be `deleteNode()`. In a straightforward implementation, the function needs to modify head pointer when the node to be deleted is first node. As discussed in [previous post](#), when a function modifies the head pointer, the function must use one of the [given approaches](#), we can't use any of those approaches here.

Solution

We explicitly handle the case when node to be deleted is first node, we copy the data of next node to head and delete the next node. The cases when deleted node is not the head node can be handled normally by finding the previous node and changing next of previous node. Following is C implementation.

C

```
#include <stdio.h>
#include <stdlib.h>

/* structure of a linked list node */
struct node
{
    int data;
    struct node *next;
};

void deleteNode(struct node *head, struct node *n)
{
    // When node to be deleted is head node
    if(head == n)
    {
        if(head->next == NULL)
        {
            printf("There is only one node. The list can't be made empty ");
            return;
        }

        /* Copy the data of next node to head */
        head->data = head->next->data;

        // store address of next node
        n = head->next;

        // Remove the link of next node
        head->next = head->next->next;

        // free memory
        free(n);

        return;
    }

    // When not first node, follow the normal deletion process

    // find the previous node
    struct node *prev = head;
    while(prev->next != NULL && prev->next != n)
        prev = prev->next;

    // Check if node really exists in Linked List
```

```

if(prev->next == NULL)
{
    printf("\n Given node is not present in Linked List");
    return;
}

// Remove node from Linked List
prev->next = prev->next->next;

// Free memory
free(n);

return;
}

/* Utility function to insert a node at the begining */
void push(struct node **head_ref, int new_data)
{
    struct node *new_node =
        (struct node *)malloc(sizeof(struct node));
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

/* Utility function to print a linked list */
void printList(struct node *head)
{
    while(head!=NULL)
    {
        printf("%d ",head->data);
        head=head->next;
    }
    printf("\n");
}

/* Driver program to test above functions */
int main()
{
    struct node *head = NULL;

    /* Create following linked list
     12->15->10->11->5->6->2->3 */
    push(&head,3);
    push(&head,2);
    push(&head,6);
    push(&head,5);
    push(&head,11);
    push(&head,10);
    push(&head,15);
    push(&head,12);

    printf("Given Linked List: ");
    printList(head);

    /* Let us delete the node with value 10 */
    printf("\nDeleting node %d: ", head->next->next->data);
    deleteNode(head, head->next->next);

    printf("\nModified Linked List: ");
    printList(head);

    /* Let us delete the the first node */
    printf("\nDeleting first node ");
    deleteNode(head, head);

    printf("\nModified Linked List: ");
    printList(head);

    getchar();
    return 0;
}

```

Java

```
// Java program to delete a given node in linked list under given constraints

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    void deleteNode(Node node, Node n) {

        // When node to be deleted is head node
        if (node == n) {
            if (node.next == null) {
                System.out.println("There is only one node. The list "
                    + "can't be made empty ");
                return;
            }
        }

        /* Copy the data of next node to head */
        node.data = node.next.data;

        // store address of next node
        n = node.next;

        // Remove the link of next node
        node.next = node.next.next;

        // free memory
        System.gc();

        return;
    }

    // When not first node, follow the normal deletion process
    // find the previous node
    Node prev = node;
    while (prev.next != null && prev.next != n) {
        prev = prev.next;
    }

    // Check if node really exists in Linked List
    if (prev.next == null) {
        System.out.println("Given node is not present in Linked List");
        return;
    }

    // Remove node from Linked List
    prev.next = prev.next.next;

    // Free memory
    System.gc();

    return;
}

/* Utility function to print a linked list */
void printList(Node head) {
    while (head != null) {
        System.out.print(head.data + " ");
        head = head.next;
    }
    System.out.println("");
}
```

```

public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(12);
    list.head.next = new Node(15);
    list.head.next.next = new Node(10);
    list.head.next.next.next = new Node(11);
    list.head.next.next.next.next = new Node(5);
    list.head.next.next.next.next.next = new Node(6);
    list.head.next.next.next.next.next.next = new Node(2);
    list.head.next.next.next.next.next.next.next = new Node(3);

    System.out.println("Given Linked List :");
    list.printList(head);
    System.out.println("");

    // Let us delete the node with value 10
    System.out.println("Deleting node :" + head.next.next.data);
    list.deleteNode(head, head.next.next);

    System.out.println("Modified Linked list :");
    list.printList(head);
    System.out.println("");

    // Lets delete the first node
    System.out.println("Deleting first Node");
    list.deleteNode(head, head);
    System.out.println("Modified Linked List");
    list.printList(head);

}
}

// this code has been contributed by Mayank Jaiswal

```

Output:

```

Given Linked List: 12 15 10 11 5 6 2 3

Deleting node 10:
Modified Linked List: 12 15 11 5 6 2 3

Deleting first node
Modified Linked List: 15 11 5 6 2 3

```

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Compare two strings represented as linked lists

Given two linked lists, represented as linked lists (every character is a node in linked list). Write a function compare() that works similar to strcmp(), i.e., it returns 0 if both strings are same, 1 if first linked list is lexicographically greater, and -1 if second string is lexicographically greater.

Examples:

```

Input: list1 = g->e->e->k->s->a
      list2 = g->e->e->k->s->b
Output: -1

Input: list1 = g->e->e->k->s->a
      list2 = g->e->e->k->s
Output: 1

Input: list1 = g->e->e->k->s
      list2 = g->e->e->k->s
Output: 0

```

C++

```
// C++ program to compare two strings represented as linked
// lists
#include<bits/stdc++.h>
using namespace std;

// Linked list Node structure
struct Node
{
    char c;
    struct Node *next;
};

// Function to create newNode in a linkedlist
Node* newNode(char c)
{
    Node *temp = new Node;
    temp->c = c;
    temp->next = NULL;
    return temp;
}

int compare(Node *list1, Node *list2)
{
    // Traverse both lists. Stop when either end of a linked
    // list is reached or current characters don't match
    while (list1 && list2 && list1->c == list2->c)
    {
        list1 = list1->next;
        list2 = list2->next;
    }

    // If both lists are not empty, compare mismatching
    // characters
    if (list1 && list2)
        return (list1->c > list2->c)? 1:-1;

    // If either of the two lists has reached end
    if (list1 && !list2) return 1;
    if (list2 && !list1) return -1;

    // If none of the above conditions is true, both
    // lists have reached end
    return 0;
}

// Driver program
int main()
{
    Node *list1 = newNode('g');
    list1->next = newNode('e');
    list1->next->next = newNode('e');
    list1->next->next->next = newNode('k');
    list1->next->next->next->next = newNode('s');
    list1->next->next->next->next = newNode('b');

    Node *list2 = newNode('g');
    list2->next = newNode('e');
    list2->next->next = newNode('e');
    list2->next->next->next = newNode('k');
    list2->next->next->next->next = newNode('s');
    list2->next->next->next->next = newNode('a');

    cout << compare(list1, list2);

    return 0;
}
```

Java

```
// Java program to compare two strings represented as a linked list
```

```

// Linked List Class
class LinkedList {

    Node head; // head of list
    static Node a, b;

    /* Node Class */
    static class Node {

        char data;
        Node next;

        // Constructor to create a new node
        Node(char d) {
            data = d;
            next = null;
        }
    }

    int compare(Node node1, Node node2) {

        if (node1 == null && node2 == null) {
            return 1;
        }
        while (node1 != null && node2 != null && node1.data == node2.data) {
            node1 = node1.next;
            node2 = node2.next;
        }

        // if the list are diffrent in size
        if (node1 != null && node2 != null) {
            return (node1.data > node2.data ? 1 : -1);
        }

        // if either of the list has reached end
        if (node1 != null && node2 == null) {
            return 1;
        }
        if (node1 == null && node2 != null) {
            return -1;
        }
        return 0;
    }

    public static void main(String[] args) {

        LinkedList list = new LinkedList();
        Node result = null;

        list.a = new Node('g');
        list.a.next = new Node('e');
        list.a.next.next = new Node('e');
        list.a.next.next.next = new Node('k');
        list.a.next.next.next.next = new Node('s');
        list.a.next.next.next.next.next = new Node('b');

        list.b = new Node('g');
        list.b.next = new Node('e');
        list.b.next.next = new Node('e');
        list.b.next.next.next = new Node('k');
        list.b.next.next.next.next = new Node('s');
        list.b.next.next.next.next.next = new Node('a');

        int value;
        value = list.compare(a, b);
        System.out.println(value);

    }
}

// This code has been contributed by Mayank Jaiswal

```

Python

```
# Python program to compare two strings represented as
# linked lists

# A linked list node structure
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.c = key ;
        self.next = None

def compare(list1, list2):

    # Traverse both lists. Stop when either end of linked
    # list is reached or current characters don't match
    while(list1 and list2 and list1.c == list2.c):
        list1 = list1.next
        list2 = list2.next

    # If both lists are not empty, compare mismatching
    # characters
    if(list1 and list2):
        return 1 if list1.c > list2.c else -1

    # If either of the two lists has reached end
    if (list1 and not list2):
        return 1

    if (list2 and not list1):
        return -1
    return 0

# Driver program

list1 = Node('g')
list1.next = Node('e')
list1.next.next = Node('e')
list1.next.next.next = Node('k')
list1.next.next.next.next = Node('s')
list1.next.next.next.next.next = Node('b')

list2 = Node('g')
list2.next = Node('e')
list2.next.next = Node('e')
list2.next.next.next = Node('k')
list2.next.next.next.next = Node('s')
list2.next.next.next.next.next = Node('a')

print compare(list1, list2)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

1

Thanks to [Gaurav Ahirwar](#) for suggesting above implementation.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Add two numbers represented by linked lists | Set 2

Given two numbers represented by two linked lists, write a function that returns sum list. The sum list is linked list representation of addition of

two input numbers. It is not allowed to modify the lists. Also, not allowed to use explicit extra space (Hint: Use Recursion).

Example

```
Input:  
First List: 5->6->3 // represents number 563  
Second List: 8->4->2 // represents number 842  
Output  
Resultant list: 1->4->0->5 // represents number 1405
```

We strongly recommend that you click here and practice it, before moving on to the solution.

We have discussed a solution [here](#) which is for linked lists where least significant digit is first node of lists and most significant digit is last node. In this problem, most significant node is first node and least significant digit is last node and we are not allowed to modify the lists. Recursion is used here to calculate sum from right to left.

Following are the steps.

- 1) Calculate sizes of given two linked lists.
- 2) If sizes are same, then calculate sum using recursion. Hold all nodes in recursion call stack till the rightmost node, calculate sum of rightmost nodes and forward carry to left side.
- 3) If size is not same, then follow below steps:
 -a) Calculate difference of sizes of two linked lists. Let the difference be *diff*
 -b) Move *diff* nodes ahead in the bigger linked list. Now use step 2 to calculate sum of smaller list and right sub-list (of same size) of larger list. Also, store the carry of this sum.
 -c) Calculate sum of the carry (calculated in previous step) with the remaining left sub-list of larger list. Nodes of this sum are added at the beginning of sum list obtained previous step.

Following is C implementation of the above approach.

```
// A recursive program to add two linked lists  
  
#include <stdio.h>  
#include <stdlib.h>  
  
// A linked List Node  
struct node  
{  
    int data;  
    struct node* next;  
};  
  
typedef struct node node;  
  
/* A utility function to insert a node at the beginning of linked list */  
void push(struct node** head_ref, int new_data)  
{  
    /* allocate node */  
    struct node* new_node = (struct node*) malloc(sizeof(struct node));  
  
    /* put in the data */  
    new_node->data = new_data;  
  
    /* link the old list off the new node */  
    new_node->next = (*head_ref);  
  
    /* move the head to point to the new node */  
    (*head_ref) = new_node;  
}  
  
/* A utility function to print linked list */  
void printList(struct node *node)  
{  
    while (node != NULL)  
    {  
        printf("%d ", node->data);  
        node = node->next;  
    }  
    printf("\n");  
}
```

```

// A utility function to swap two pointers
void swapPointer( node** a, node** b )
{
    node* t = *a;
    *a = *b;
    *b = t;
}

/* A utility function to get size of linked list */
int getSize(struct node *node)
{
    int size = 0;
    while (node != NULL)
    {
        node = node->next;
        size++;
    }
    return size;
}

// Adds two linked lists of same size represented by head1 and head2 and returns
// head of the resultant linked list. Carry is propagated while returning from
// the recursion
node* addSameSize(node* head1, node* head2, int* carry)
{
    // Since the function assumes linked lists are of same size,
    // check any of the two head pointers
    if (head1 == NULL)
        return NULL;

    int sum;

    // Allocate memory for sum node of current two nodes
    node* result = (node *)malloc(sizeof(node));

    // Recursively add remaining nodes and get the carry
    result->next = addSameSize(head1->next, head2->next, carry);

    // add digits of current nodes and propagated carry
    sum = head1->data + head2->data + *carry;
    *carry = sum / 10;
    sum = sum % 10;

    // Assign the sum to current node of resultant list
    result->data = sum;

    return result;
}

// This function is called after the smaller list is added to the bigger
// lists's sublist of same size. Once the right sublist is added, the carry
// must be added toe left side of larger list to get the final result.
void addCarryToRemaining(node* head1, node* cur, int* carry, node** result)
{
    int sum;

    // If diff. number of nodes are not traversed, add carry
    if (head1 != cur)
    {
        addCarryToRemaining(head1->next, cur, carry, result);

        sum = head1->data + *carry;
        *carry = sum/10;
        sum %= 10;

        // add this node to the front of the result
        push(result, sum);
    }
}

// The main function that adds two linked lists represented by head1 and head2.
// The sum of two lists is stored in a list referred by result
void addList(node* head1, node* head2, node** result)
{

```

```

node *cur;

// first list is empty
if (head1 == NULL)
{
    *result = head2;
    return;
}

// second list is empty
else if (head2 == NULL)
{
    *result = head1;
    return;
}

int size1 = getSize(head1);
int size2 = getSize(head2) ;

int carry = 0;

// Add same size lists
if (size1 == size2)
    *result = addSameSize(head1, head2, &carry);

else
{
    int diff = abs(size1 - size2);

    // First list should always be larger than second list.
    // If not, swap pointers
    if (size1 < size2)
        swapPointer(&head1, &head2);

    // move diff. number of nodes in first list
    for (cur = head1; diff--; cur = cur->next);

    // get addition of same size lists
    *result = addSameSize(cur, head2, &carry);

    // get addition of remaining first list and carry
    addCarryToRemaining(head1, cur, &carry, result);
}

// if some carry is still there, add a new node to the front of
// the result list. e.g. 999 and 87
if (carry)
    push(result, carry);
}

// Driver program to test above functions
int main()
{
    node *head1 = NULL, *head2 = NULL, *result = NULL;

    int arr1[] = {9, 9, 9};
    int arr2[] = {1, 8};

    int size1 = sizeof(arr1) / sizeof(arr1[0]);
    int size2 = sizeof(arr2) / sizeof(arr2[0]);

    // Create first list as 9->9->9
    int i;
    for (i = size1-1; i >= 0; --i)
        push(&head1, arr1[i]);

    // Create second list as 1->8
    for (i = size2-1; i >= 0; --i)
        push(&head2, arr2[i]);

    addList(head1, head2, &result);

    printList(result);

    return 0;
}

```

```
}
```

Output:

```
1 0 1 7
```

Time Complexity: O(m+n) where m and n are the sizes of given two linked lists.

Related Article : [Add two numbers represented by linked lists | Set 1](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Linked Lists

Merge a linked list into another linked list at alternate positions

Given two linked lists, insert nodes of second list into first list at alternate positions of first list.

For example, if first list is 5->7->17->13->11 and second is 12->10->2->4->6, the first list should become 5->12->7->10->17->2->13->4->11->6 and second list should become empty. The nodes of second list should only be inserted when there are positions available. For example, if the first list is 1->2->3 and second list is 4->5->6->7->8, then first list should become 1->4->2->5->3->6 and second list to 7->8.

Use of extra space is not allowed (Not allowed to create additional nodes), i.e., insertion must be done in-place. Expected time complexity is O(n) where n is number of nodes in first list.

The idea is to run a loop while there are available positions in first loop and insert nodes of second list by changing pointers. Following are C and Java implementations of this approach.

C/C++

```
// C program to merge a linked list into another at
// alternate positions
#include <stdio.h>
#include <stdlib.h>

// A nexted list node
struct node
{
    int data;
    struct node *next;
};

/* Function to insert a node at the beginning */
void push(struct node ** head_ref, int new_data)
{
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Utility function to print a singly linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function that inserts nodes of linked list q into p at
// alternate positions. Since head of first list never changes
// and head of second list may change, we need single pointer
// for first list and double pointer for second list.
void merge(struct node *p, struct node **q)
{
```

```

struct node *p_curr = p, *q_curr = q;
struct node *p_next, *q_next;

// While there are available positions in p
while (p_curr != NULL && q_curr != NULL)
{
    // Save next pointers
    p_next = p_curr->next;
    q_next = q_curr->next;

    // Make q_curr as next of p_curr
    q_curr->next = p_next; // Change next pointer of q_curr
    p_curr->next = q_curr; // Change next pointer of p_curr

    // Update current pointers for next iteration
    p_curr = p_next;
    q_curr = q_next;
}

*q = q_curr; // Update head pointer of second list
}

// Driver program to test above functions
int main()
{
    struct node *p = NULL, *q = NULL;
    push(&p, 3);
    push(&p, 2);
    push(&p, 1);
    printf("First Linked List:\n");
    printList(p);

    push(&q, 8);
    push(&q, 7);
    push(&q, 6);
    push(&q, 5);
    push(&q, 4);
    printf("Second Linked List:\n");
    printList(q);

    merge(p, &q);

    printf("Modified First Linked List:\n");
    printList(p);

    printf("Modified Second Linked List:\n");
    printList(q);

    getchar();
    return 0;
}

```

Java

```

// Java program to merge a linked list into another at
// alternate positions
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    /* Inserts a new Node at front of the list.*/
    void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
           Put in the data*/

```

```

Node new_node = new Node(new_data);

/* 3. Make next of new Node as head */
new_node.next = head;

/* 4. Move the head to point to new Node */
head = new_node;
}

// Main function that inserts nodes of linked list q into p at
// alternate positions. Since head of first list never changes
// and head of second list/ may change, we need single pointer
// for first list and double pointer for second list.
void merge(LinkedList q)
{
    Node p_curr = head, q_curr = q.head;
    Node p_next, q_next;

    // While there are available positions in p;
    while (p_curr != null && q_curr != null) {

        // Save next pointers
        p_next = p_curr.next;
        q_next = q_curr.next;

        // make q_curr as next of p_curr
        q_curr.next = p_next; // change next pointer of q_curr
        p_curr.next = q_curr; // change next pointer of p_curr

        // update current pointers for next iteration
        p_curr = p_next;
        q_curr = q_next;
    }
    q.head = q_curr;
}

/* Function to print linked list */
void printList()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Driver program to test above functions */
public static void main(String args[])
{
    LinkedList llist1 = new LinkedList();
    LinkedList llist2 = new LinkedList();
    llist1.push(3);
    llist1.push(2);
    llist1.push(1);

    System.out.println("First Linked List:");
    llist1.printList();

    llist2.push(8);
    llist2.push(7);
    llist2.push(6);
    llist2.push(5);
    llist2.push(4);

    System.out.println("Second Linked List:");

    llist1.merge(llist2);

    System.out.println("Modified first linked list:");
    llist1.printList();

    System.out.println("Modified second linked list:");
    llist2.printList();
}

```

```
}
```

```
/* This code is contributed by Rajat Mishra */
```

Python

```
# Python program to merge a linked list into another at
# alternate positions

class LinkedList(object):
    def __init__(self):
        # head of list
        self.head = None

    # Linked list Node
    class Node(object):
        def __init__(self, d):
            self.data = d
            self.next = None

    # Inserts a new Node at front of the list.
    def push(self, new_data):

        # 1 & 2: Allocate the Node &
        # Put in the data
        new_node = self.Node(new_data)

        # 3. Make next of new Node as head
        new_node.next = self.head

        # 4. Move the head to point to new Node
        self.head = new_node

    # Main function that inserts nodes of linked list q into p at
    # alternate positions. Since head of first list never changes
    # and head of second list/ may change, we need single pointer
    # for first list and double pointer for second list.
    def merge(self, q):
        p_curr = self.head
        q_curr = q.head

        # While there are available positions in p;
        while p_curr != None and q_curr != None:

            # Save next pointers
            p_next = p_curr.next
            q_next = q_curr.next

            # make q_curr as next of p_curr
            q_curr.next = p_next # change next pointer of q_curr
            p_curr.next = q_curr # change next pointer of p_curr

            # update current pointers for next iteration
            p_curr = p_next
            q_curr = q_next
            q.head = q_curr

    # Function to print linked list
    def printList(self):
        temp = self.head
        while temp != None:
            print str(temp.data),
            temp = temp.next
        print ""

    # Driver program to test above functions
    llist1 = LinkedList()
    llist2 = LinkedList()
    llist1.push(3)
    llist1.push(2)
    llist1.push(1)

    print "First Linked List:"
    llist1.printList()
```

```

llist2.push(8)
llist2.push(7)
llist2.push(6)
llist2.push(5)
llist2.push(4)

print "Second Linked List:"

llist2.printList()
llist1.merge(llist2)

print "Modified first linked list:"
llist1.printList()

print "Modified second linked list:"
llist2.printList()

# This code is contributed by BHAVYA JAIN

```

Output:

```

First Linked List:
1 2 3
Second Linked List:
4 5 6 7 8
Modified First Linked List:
1 4 2 5 3 6
Modified Second Linked List:
7 8

```

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Linked Lists

Reverse a Linked List in groups of given size

Given a linked list, write a function to reverse every k nodes (where k is an input to the function).

Example:
Inputs: 1->2->3->4->5->6->7->8->NULL and k = 3
Output: 3->2->1->6->5->4->8->7->NULL.

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 5
Output: 5->4->3->2->1->8->7->6->NULL.

Algorithm: *reverse(head, k)*

- 1) Reverse the first sub-list of size k. While reversing keep track of the next node and previous node. Let the pointer to the next node be *next* and pointer to the previous node be *prev*. See [this post](#) for reversing a linked list.
- 2) *head->next = reverse(next, k)* /* Recursively call for rest of the list and link the two sub-lists */
- 3) return *prev* /* *prev* becomes the new head of the list (see the diagrams of iterative method of [this post](#)) */

C/C++

```

// C program to reverse a linked list in groups of given size
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Reverses the linked list in groups of size k and returns the
   head of reversed list */

```

```

pointer to the new head node.*/
struct node *reverse (struct node *head, int k)
{
    struct node* current = head;
    struct node* next = NULL;
    struct node* prev = NULL;
    int count = 0;

    /*reverse first k nodes of the linked list*/
    while (current != NULL && count < k)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
        count++;
    }

    /* next is now a pointer to (k+1)th node
     Recursively call for the list starting from current.
     And make rest of the list as next of first node */
    if (next != NULL)
        head->next = reverse(next, k);

    /* prev is new head of the input list*/
    return prev;
}

/* UTILITY FUNCTIONS*/
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Created Linked list is 1->2->3->4->5->6->7->8->9 */
    push(&head, 9);
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\nGiven linked list\n");
    printList(head);
}

```

```

head = reverse(head, 3);

printf("\nReversed Linked list \n");
printList(head);

return(0);
}

```

Java

```

// Java program to reverse a linked list in groups of
// given size
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    Node reverse(Node head, int k)
    {
        Node current = head;
        Node next = null;
        Node prev = null;

        int count = 0;

        /* Reverse first k nodes of linked list */
        while (count < k && current != null)
        {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
            count++;
        }

        /* next is now a pointer to (k+1)th node
         * Recursively call for the list starting from current.
         * And make rest of the list as next of first node */
        if (next != null)
            head.next = reverse(next, k);

        // prev is now head of input list
        return prev;
    }

    /* Utility functions */

    /* Inserts a new Node at front of the list.*/
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
           Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    /* Function to print linked list */
    void printList()
    {
        Node temp = head;

```

```

while (temp != null)
{
    System.out.print(temp.data+" ");
    temp = temp.next;
}
System.out.println();
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    /* Constructed Linked List is 1->2->3->4->5->6->
     * 7->8->9->null */
    llist.push(9);
    llist.push(8);
    llist.push(7);
    llist.push(6);
    llist.push(5);
    llist.push(4);
    llist.push(3);
    llist.push(2);
    llist.push(1);

    System.out.println("Given Linked List");
    llist.printList();

    llist.head = llist.reverse(llist.head, 3);

    System.out.println("Reversed list");
    llist.printList();
}
}
/* This code is contributed by Rajat Mishra */

```

Python

```

# Python program to reverse a linked list in group of given size

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def reverse(self, head, k):
        current = head
        next = None
        prev = None
        count = 0

        # Reverse first k nodes of the linked list
        while(current is not None and count < k):
            next = current.next
            current.next = prev
            prev = current
            current = next
            count += 1

        # next is now a pointer to (k+1)th node
        # recursively call for the list starting
        # from current . And make rest of the list as
        # next of first node

```

```

if next is not None:
    head.next = self.reverse(next, k)

# prev is new head of the input list
return prev

# Function to insert a new node at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program
llist = LinkedList()
llist.push(9)
llist.push(8)
llist.push(7)
llist.push(6)
llist.push(5)
llist.push(4)
llist.push(3)
llist.push(2)
llist.push(1)

print "Given linked list"
llist.printList()
llist.head = llist.reverse(llist.head, 3)

print "\nReversed Linked list"
llist.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Given Linked List
1 2 3 4 5 6 7 8 9
Reversed list
3 2 1 6 5 4 9 8 7

```

Time Complexity: O(n) where n is the number of nodes in the given list.

Please write comments if you find the above code/algorithm incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Linked Lists
[Adobe-Question](#)
[Amazon-Question](#)
[Reverse](#)
[Snapdeal-Question](#)
[Yatra.com-Question](#)

Union and Intersection of two Linked Lists

Given two Linked Lists, create union and intersection lists that contain union and intersection of the elements present in the given lists. Order of elements in output lists doesn't matter.

Example:

```

Input:
List1: 10->15->4->20
List2: 8->4->2->10
Output:
Intersection List: 4->10

```

Method 1 (Simple)

Following are simple algorithms to get union and intersection lists respectively.

Intersection (list1, list2)

Initialize result list as NULL. Traverse list1 and look for its each element in list2, if the element is present in list2, then add the element to result.

Union (list1, list2):

Initialize result list as NULL. Traverse list1 and add all of its elements to the result.

Traverse list2. If an element of list2 is already present in result then do not insert it to result, otherwise insert.

This method assumes that there are no duplicates in the given lists.

Thanks to Shekhu for suggesting this method. Following are C and Java implementations of this method.

C/C++

```
// C/C++ program to find union and intersection of two unsorted
// linked lists
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* A utility function to insert a node at the beginning of
   a linked list*/
void push(struct node** head_ref, int new_data);

/* A utility function to check if given data is present in a list */
bool isPresent(struct node *head, int data);

/* Function to get union of two linked lists head1 and head2 */
struct node *getUnion(struct node *head1, struct node *head2)
{
    struct node *result = NULL;
    struct node *t1 = head1, *t2 = head2;

    // Insert all elements of list1 to the result list
    while (t1 != NULL)
    {
        push(&result, t1->data);
        t1 = t1->next;
    }

    // Insert those elements of list2 which are not
    // present in result list
    while (t2 != NULL)
    {
        if (!isPresent(result, t2->data))
            push(&result, t2->data);
        t2 = t2->next;
    }

    return result;
}

/* Function to get intersection of two linked lists
   head1 and head2 */
struct node *getIntersection(struct node *head1,
                           struct node *head2)
{
    struct node *result = NULL;
    struct node *t1 = head1;

    // Traverse list1 and search each element of it in
    // list2. If the element is present in list 2, then
    // insert the element to result
```

```

while (t1 != NULL)
{
    if (isPresent(head2, t1->data))
        push (&result, t1->data);
    t1 = t1->next;
}

return result;
}

/* A utility function to insert a node at the begining of a linked list*/
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* A utility function to print a linked list*/
void printList (struct node *node)
{
    while (node != NULL)
    {
        printf ("%d ", node->data);
        node = node->next;
    }
}

/* A utility function that returns true if data is
   present in linked list else return false */
bool isPresent (struct node *head, int data)
{
    struct node *t = head;
    while (t != NULL)
    {
        if (t->data == data)
            return 1;
        t = t->next;
    }
    return 0;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head1 = NULL;
    struct node* head2 = NULL;
    struct node* intersecn = NULL;
    struct node* unin = NULL;

    /*create a linked lits 10->15->5->20 */
    push (&head1, 20);
    push (&head1, 4);
    push (&head1, 15);
    push (&head1, 10);

    /*create a linked lits 8->4->2->10 */
    push (&head2, 10);
    push (&head2, 2);
    push (&head2, 4);
    push (&head2, 8);

    intersecn = getIntersection (head1, head2);
    unin = getUnion (head1, head2);
}

```

```

printf ("\n First list is \n");
printList (head1);

printf ("\n Second list is \n");
printList (head2);

printf ("\n Intersection list is \n");
printList (intersecn);

printf ("\n Union list is \n");
printList (unin);

return 0;
}

```

Java

```

// Java program to find union and intersection of two unsorted
// linked lists
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    /* Function to get Union of 2 Linked Lists */
    void getUnion(Node head1, Node head2)
    {
        Node t1 = head1, t2 = head2;

        //insert all elements of list1 in the result
        while (t1 != null)
        {
            push(t1.data);
            t1 = t1.next;
        }

        // insert those elements of list2 that are not present
        while (t2 != null)
        {
            if (!isPresent(head, t2.data))
                push(t2.data);
            t2 = t2.next;
        }
    }

    void getIntersection(Node head1, Node head2)
    {
        Node result = null;
        Node t1 = head1;

        // Traverse list1 and search each element of it in list2.
        // If the element is present in list 2, then insert the
        // element to result
        while (t1 != null)
        {
            if (isPresent(head2, t1.data))
                push(t1.data);
            t1 = t1.next;
        }
    }

    /* Utility function to print list */

```

```

void printList()
{
    Node temp = head;
    while(temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Inserts a node at start of linked list */
void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* A utility function that returns true if data is present
   in linked list else return false */
boolean isPresent (Node head, int data)
{
    Node t = head;
    while (t != null)
    {
        if (t.data == data)
            return true;
        t = t.next;
    }
    return false;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    LinkedList llist1 = new LinkedList();
    LinkedList llist2 = new LinkedList();
    LinkedList unin = new LinkedList();
    LinkedList intersecn = new LinkedList();

    /*create a linked lists 10->15->5->20 */
    llist1.push(20);
    llist1.push(4);
    llist1.push(15);
    llist1.push(10);

    /*create a linked lists 8->4->2->10 */
    llist2.push(10);
    llist2.push(2);
    llist2.push(4);
    llist2.push(8);

    intersecn.getIntersection(llist1.head, llist2.head);
    unin.getUnion(llist1.head, llist2.head);

    System.out.println("First List is");
    llist1.printList();

    System.out.println("Second List is");
    llist2.printList();

    System.out.println("Intersection List is");
    intersecn.printList();

    System.out.println("Union List is");
}

```

```

    unin.printList();
}
} /* This code is contributed by Rajat Mishra */

```

Output:

```

First list is
10 15 4 20
Second list is
8 4 2 10
Intersection list is
4 10
Union list is
2 8 20 4 15 10

```

Time Complexity: $O(mn)$ for both union and intersection operations. Here m is the number of elements in first list and n is the number of elements in second list.

Method 2 (Use Merge Sort)

In this method, algorithms for Union and Intersection are very similar. First we sort the given lists, then we traverse the sorted lists to get union and intersection.

Following are the steps to be followed to get union and intersection lists.

- 1) Sort the first Linked List using merge sort. This step takes $O(m\log m)$ time. Refer [this post](#) for details of this step.
- 2) Sort the second Linked List using merge sort. This step takes $O(n\log n)$ time. Refer [this post](#) for details of this step.
- 3) Linearly scan both sorted lists to get the union and intersection. This step takes $O(m + n)$ time. This step can be implemented using the same algorithm as sorted arrays algorithm discussed [here](#).

Time complexity of this method is $O(m\log m + n\log n)$ which is better than method 1's time complexity.

Method 3 (Use Hashing)

Union (list1, list2)

Initialize the result list as NULL and create an empty hash table. Traverse both lists one by one, for each element being visited, look the element in hash table. If the element is not present, then insert the element to result list. If the element is present, then ignore it.

Intersection (list1, list2)

Initialize the result list as NULL and create an empty hash table. Traverse list1. For each element being visited in list1, insert the element in hash table. Traverse list2, for each element being visited in list2, look the element in hash table. If the element is present, then insert the element to result list. If the element is not present, then ignore it.

Both of the above methods assume that there are no duplicates.

Time complexity of this method depends on the hashing technique used and the distribution of elements in input lists. In practical, this approach may turn out to be better than above 2 methods.

Source: <http://geeksforgeeks.org/forum/topic/union-intersection-of-unsorted-lists>

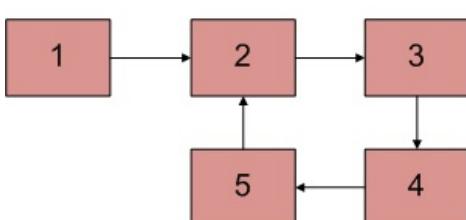
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Detect and Remove Loop in a Linked List

Write a function *detectAndRemoveLoop()* that checks whether a given Linked List contains loop and if loop is present then removes the loop and returns true. And if the list doesn't contain loop then returns false. Below diagram shows a linked list with a loop. *detectAndRemoveLoop()* must change the below list to 1->2->3->4->5->NULL.



We also recommend to read following post as a prerequisite of the solution discussed here.

Write a C function to detect loop in a linked list

Before trying to remove the loop, we must detect it. Techniques discussed in the above post can be used to detect loop. To remove loop, all we need to do is to get pointer to the last node of the loop. For example, node with value 5 in the above diagram. Once we have pointer to the last node, we can make the next of this node as NULL and loop is gone.

We can easily use Hashing or Visited node techniques (discussed in the above mentioned post) to get the pointer to the last node. Idea is simple: the very first node whose next is already visited (or hashed) is the last node.

We can also use Floyd Cycle Detection algorithm to detect and remove the loop. In the Floyd's algo, the slow and fast pointers meet at a loop node. We can use this loop node to remove cycle. There are following two different ways of removing loop when Floyd's algorithm is used for Loop detection.

Method 1 (Check one by one)

We know that Floyd's Cycle detection algorithm terminates when fast and slow pointers meet at a common point. We also know that this common point is one of the loop nodes (2 or 3 or 4 or 5 in the above diagram). We store the address of this in a pointer variable say ptr2. Then we start from the head of the Linked List and check for nodes one by one if they are reachable from ptr2. When we find a node that is reachable, we know that this node is the starting node of the loop in Linked List and we can get pointer to the previous of this node.

C

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. Used by detectAndRemoveLoop() */
void removeLoop(struct node * , struct node *);

/* This function detects and removes loop in the list
   If loop was there in the list then it returns 1,
   otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indicate that there is no loop*/
    return 0;
}

/* Function to remove loop.
loop_node --> Pointer to one of the loop nodes
head --> Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
    struct node *ptr1;
    struct node *ptr2;

    /* Set a pointer to the beginning of the Linked List and
       move it one by one to find the first node which is
       part of the Linked List */
    ptr1 = head;
```

```

while (1)
{
    /* Now start a pointer from loop_node and check if it ever
       reaches ptr2 */
    ptr2 = loop_node;
    while (ptr2->next != loop_node && ptr2->next != ptr1)
        ptr2 = ptr2->next;

    /* If ptr2 reached ptr1 then there is a loop. So break the
       loop */
    if (ptr2->next == ptr1)
        break;

    /* If ptr2 didn't reach ptr1 then try the next node after ptr1 */
    ptr1 = ptr1->next;
}

/* After the end of loop ptr2 is the last node of the loop. So
   make next of ptr2 as NULL */
ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

struct node *newNode(int key)
{
    struct node *temp = new struct node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Driver program to test above function*/
int main()
{
    struct node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);
    return 0;
}

```

Java

```

// Java program to detect and remove loop in linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {

```

```

        data = d;
        next = null;
    }

}

// Function that detects loop in the list
int detectAndRemoveLoop(Node node) {
    Node slow = node, fast = node;
    while (slow != null && fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;

        // If slow and fast meet at same point then loop is present
        if (slow == fast) {
            removeLoop(slow, node);
            return 1;
        }
    }
    return 0;
}

// Function to remove loop
void removeLoop(Node loop, Node curr) {
    Node ptr1 = null, ptr2 = null;

    /* Set a pointer to the begining of the Linked List and
       move it one by one to find the first node which is
       part of the Linked List */
    ptr1 = curr;
    while (1 == 1) {

        /* Now start a pointer from loop_node and check if it ever
           reaches ptr2 */
        ptr2 = loop;
        while (ptr2.next != loop && ptr2.next != ptr1) {
            ptr2 = ptr2.next;
        }

        /* If ptr2 reahced ptr1 then there is a loop. So break the
           loop */
        if (ptr2.next == ptr1) {
            break;
        }

        /* If ptr2 didn't reach ptr1 then try the next node after ptr1 */
        ptr1 = ptr1.next;
    }

    /* After the end of loop ptr2 is the last node of the loop. So
       make next of ptr2 as NULL */
    ptr2.next = null;
}

// Function to print the linked list
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

// Driver program to test above functions
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(50);
    list.head.next = new Node(20);
    list.head.next.next = new Node(15);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(10);

    // Creating a loop for testing
    head.next.next.next.next = head.next.next;
    list.detectAndRemoveLoop(head);
    System.out.println("Linked List after removing loop : ");
    list.printList(head);
}

```

```
}
```

// This code has been contributed by Mayank Jaiswal

Python

```
# Python program to detect and remove loop in linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def detectAndRemoveLoop(self):
        slow_p = fast_p = self.head
        while(slow_p and fast_p and fast_p.next):
            slow_p = slow_p.next
            fast_p = fast_p.next.next

            # If slow_p and fast_p meet at some point
            # then there is a loop
            if slow_p == fast_p:
                self.removeLoop(slow_p)

        # Return 1 to indicate that loop is found
        return 1

    # Return 0 to indicate that there is no loop
    return 0

    # Function to remove loop
    # loop_node-> Pointer to one of the loop nodes
    # head --> Pointer to the start node of the
    # linked list
    def removeLoop(self, loop_node):

        # Set a pointer to the beginning of the linked
        # list and move it one by one to find the first
        # node which is part of the linked list
        ptr1 = self.head
        while(1):
            # Now start a pointer from loop_node and check
            # if it ever reaches ptr2
            ptr2 = loop_node
            while(ptr2.next!= loop_node and ptr2.next !=ptr1):
                ptr2 = ptr2.next

            # If ptr2 reached ptr1 then there is a loop.
            # So break the loop
            if ptr2.next == ptr1 :
                break

            ptr1 = ptr1.next

        # After the end of loop ptr2 is the last node of
        # the loop. So make next of ptr2 as NULL
        ptr2.next = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node
```

```

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program
llist = LinkedList()
llist.push(10)
llist.push(4)
llist.push(15)
llist.push(20)
llist.push(50)

# Create a loop for testing
llist.head.next.next.next.next = llist.head.next.next

llist.detectAndRemoveLoop()

print "Linked List after removing loop"
llist.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Linked List after removing loop
50 20 15 4 10

```

Method 2 (Better Solution)

This method is also dependent on Floyd's Cycle detection algorithm.

- 1) Detect Loop using Floyd's Cycle detection algo and get the pointer to a loop node.
- 2) Count the number of nodes in loop. Let the count be k.
- 3) Fix one pointer to the head and another to kth node from head.
- 4) Move both pointers at the same pace, they will meet at loop starting node.
- 5) Get pointer to the last node of loop and make next of it as NULL.

Thanks to WgpShashank for suggesting this method.

C

```

#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
   If loop was there in the list then it returns 1,
   otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
    }
}

```

```

if (slow_p == fast_p)
{
    removeLoop(slow_p, list);

    /* Return 1 to indicate that loop is found */
    return 1;
}

/* Return 0 to indicate that there is no loop*/
return 0;
}

/* Function to remove loop.
loop_node --> Pointer to one of the loop nodes
head --> Pointer to the start node of the linked list*/
void removeLoop(struct node *loop_node, struct node *head)
{
    struct node *ptr1 = loop_node;
    struct node *ptr2 = loop_node;

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2)
    {
        ptr1 = ptr1->next;
        k++;
    }

    // Fix one pointer to head
    ptr1 = head;

    // And the other pointer to k nodes after head
    ptr2 = head;
    for (i = 0; i < k; i++)
        ptr2 = ptr2->next;

    /* Move both pointers at the same pace,
    they will meet at loop starting node */
    while (ptr2 != ptr1)
    {
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }

    // Get pointer to the last node
    ptr2 = ptr2->next;
    while (ptr2->next != ptr1)
        ptr2 = ptr2->next;

    /* Set the next node of the loop ending node
    to fix the loop */
    ptr2->next = NULL;
}

/* Function to print linked list*/
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

struct node *newNode(int key)
{
    struct node *temp = new struct node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Driver program to test above function*/
int main()

```

```
{
    struct node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);
    return 0;
}
```

Java

```
// Java program to detect and remove loop in linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    // Function that detects loop in the list
    int detectAndRemoveLoop(Node node) {
        Node slow = node, fast = node;
        while (slow != null && fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;

            // If slow and fast meet at same point then loop is present
            if (slow == fast) {
                removeLoop(slow, node);
                return 1;
            }
        }
        return 0;
    }

    // Function to remove loop
    void removeLoop(Node loop, Node head) {
        Node ptr1 = loop;
        Node ptr2 = loop;

        // Count the number of nodes in loop
        int k = 1, i;
        while (ptr1.next != ptr2) {
            ptr1 = ptr1.next;
            k++;
        }

        // Fix one pointer to head
        ptr1 = head;

        // And the other pointer to k nodes after head
        ptr2 = head;
        for (i = 0; i < k; i++) {
            ptr2 = ptr2.next;
        }
    }
}
```

```

/* Move both pointers at the same pace,
they will meet at loop starting node */
while (ptr2 != ptr1) {
    ptr1 = ptr1.next;
    ptr2 = ptr2.next;
}

// Get pointer to the last node
ptr2 = ptr2.next;
while (ptr2.next != ptr1) {
    ptr2 = ptr2.next;
}

/* Set the next node of the loop ending node
to fix the loop */
ptr2.next = null;
}

// Function to print the linked list
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

// Driver program to test above functions
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(50);
    list.head.next = new Node(20);
    list.head.next.next = new Node(15);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(10);

    // Creating a loop for testing
    head.next.next.next.next = head.next.next;
    list.detectAndRemoveLoop(head);
    System.out.println("Linked List after removing loop : ");
    list.printList(head);
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to detect and remove loop in linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def detectAndRemoveLoop(self):
        slow_p = fast_p = self.head

        while(slow_p and fast_p and fast_p.next):
            slow_p = slow_p.next
            fast_p = fast_p.next.next

        # If slow_p and fast_p meet at some point then

```

```

# there is a loop
if slow_p == fast_p:
    self.removeLoop(slow_p)

    # Return 1 to indicate that loop is found
    return 1

# Return 0 to indicate that there is no loop
return 0

# Function to remove loop
# loop_node --> pointer to one of the loop nodes
# head --> Pointer to the start node of the linked list
def removeLoop(self, loop_node):
    ptr1 = loop_node
    ptr2 = loop_node

    # Count the number of nodes in loop
    k = 1
    while(ptr1.next != ptr2):
        ptr1 = ptr1.next
        k += 1

    # Fix one pointer to head
    ptr1 = self.head

    # And the other pointer to k nodes after head
    ptr2 = self.head
    for i in range(k):
        ptr2 = ptr2.next

    # Move both pointers at the same place
    # they will meet at loop starting node
    while(ptr2 != ptr1):
        ptr1 = ptr1.next
        ptr2 = ptr2.next

    # Get pointer to the last node
    ptr2 = ptr2.next
    while(ptr2.next != ptr1):
        ptr2 = ptr2.next

    # Set the next node of the loop ending node
    # to fix the loop
    ptr2.next = None

# Function to insert a new node at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program
llist = LinkedList()
llist.push(10)
llist.push(4)
llist.push(15)
llist.push(20)
llist.push(50)

# Create a loop for testing
llist.head.next.next.next.next = llist.head.next.next

llist.detectAndRemoveLoop()

print "Linked List after removing loop"
llist.printList()

```

Output:

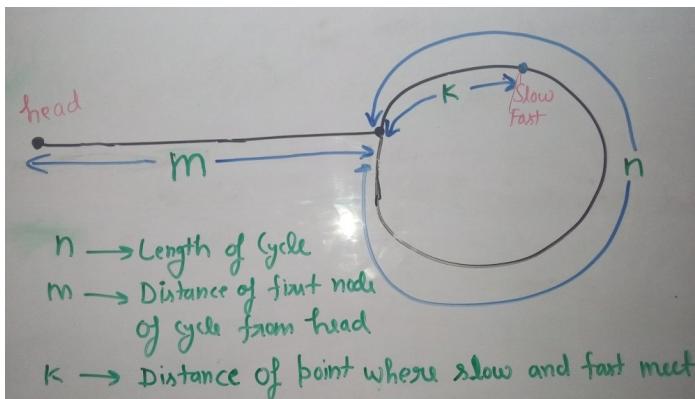
Linked List after removing loop
50 20 15 4 10

Method 3 (Optimized Method 2: Without Counting Nodes in Loop)

We do not need to count number of nodes in Loop. After detecting the loop, if we start slow pointer from head and move both slow and fast pointers at same speed until fast don't meet, they would meet at the beginning of linked list.

How does this work?

Let slow and fast meet at some point after Floyd's Cycle finding algorithm. Below diagram shows the situation when cycle is found.



We can conclude below from above diagram

Distance traveled by fast pointer = $2 * (\text{Distance traveled by slow pointer})$

$$(m + n*x + k) = 2*(m + n*y + k)$$

Note that before meeting the point shown above, fast was moving at twice speed.

$x \rightarrow$ Number of complete cyclic rounds made by fast pointer before they meet first time

$y \rightarrow$ Number of complete cyclic rounds made by slow pointer before they meet first time

From above equation, we can conclude below

$$m + k = (x - 2y)*n$$

Which means **m+k is a multiple of n**.

So if we start moving both pointers again at **same speed** such that one pointer (say slow) begins from head node of linked list and other pointer (say fast) begins from meeting point. When slow pointer reaches beginning of linked list (has made m steps). Fast pointer would have made also moved m steps as they are now moving same pace. Since $m+k$ is a multiple of n and fast starts from k , they would meet at the beginning. Can they meet before also? No because slow pointer enters the cycle first time after m steps.

C++

```
// C++ program to detect and remove loop
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int key;
    struct Node *next;
};

Node *newNode(int key)
```

```

{
    Node *temp = new Node;
    temp->key = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printList(Node *head)
{
    while (head != NULL)
    {
        cout << head->key << " ";
        head = head->next;
    }
    cout << endl;
}

void detectAndRemoveLoop(Node *head)
{
    Node *slow = head;
    Node *fast = head->next;

    // Search for loop using slow and fast pointers
    while (fast && fast->next)
    {
        if (slow == fast)
            break;
        slow = slow->next;
        fast = fast->next->next;
    }

    /* If loop exists */
    if (slow == fast)
    {
        slow = head;
        while (slow != fast->next)
        {
            slow = slow->next;
            fast = fast->next;
        }

        /* since fast->next is the looping point */
        fast->next = NULL; /* remove loop */
    }
}

/* Driver program to test above function*/
int main()
{
    Node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    return 0;
}

```

Java

```

// Java program to detect and remove loop in linked list

class LinkedList {

```

```

static Node head;

static class Node {

    int data;
    Node next;

    Node(int d) {
        data = d;
        next = null;
    }
}

// Function that detects loop in the list
void detectAndRemoveLoop(Node node) {
    Node slow = node;
    Node fast = node.next;

    // Search for loop using slow and fast pointers
    while (fast != null && fast.next != null) {
        if (slow == fast) {
            break;
        }
        slow = slow.next;
        fast = fast.next.next;
    }

    /* If loop exists */
    if (slow == fast) {
        slow = node;
        while (slow != fast.next) {
            slow = slow.next;
            fast = fast.next;
        }
    }

    /* since fast->next is the looping point */
    fast.next = null; /* remove loop */
}

}

// Function to print the linked list
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

// Driver program to test above functions
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(50);
    list.head.next = new Node(20);
    list.head.next.next = new Node(15);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(10);

    // Creating a loop for testing
    head.next.next.next.next = head.next.next;
    list.detectAndRemoveLoop(head);
    System.out.println("Linked List after removing loop : ");
    list.printList(head);
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```
# Python program to detect and remove loop
```

```

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    def detectAndRemoveLoop(self):
        slow = self.head
        fast = self.head.next

        # Search for loop using slow and fast pointers
        while(fast is not None):
            if fast.next is None:
                break
            if slow == fast :
                break
            slow = slow.next
            fast = fast.next.next

        # If loop exists
        if slow == fast :
            slow = self.head
            while(slow != fast.next):
                slow = slow.next
                fast = fast.next

        # Since fast.next is the looping point
        fast.next = None # Remove loop

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

    # Driver program
    llist = LinkedList()
    llist.head = Node(50)
    llist.head.next = Node(20)
    llist.head.next.next = Node(15)
    llist.head.next.next.next = Node(4)
    llist.head.next.next.next.next = Node(10)

    # Create a loop for testing
    llist.head.next.next.next.next = llist.head.next.next

    llist.detectAndRemoveLoop()

    print "Linked List after removing loop"
    llist.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

Linked List after removing loop

50 20 15 4 10

Thanks to [Gaurav Ahirwar](#) for suggesting above solution.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Linked Lists

Merge Sort for Linked Lists

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Let head be the first node of the linked list to be sorted and headRef be the pointer to head. Note that we need a reference to head in MergeSort() as the below implementation changes next links to sort the linked lists (not data at the nodes), so head node has to be changed if the data at original head is not the smallest value in linked list.

```
MergeSort(headRef)
1) If head is NULL or there is only one element in the Linked List
   then return.
2) Else divide the linked list into two halves.
   FrontBackSplit(head, &a, &b); /* a and b are two halves */
3) Sort the two halves a and b.
   MergeSort(a);
   MergeSort(b);
4) Merge the sorted a and b (using SortedMerge() discussed here)
   and update the head pointer using headRef.
   *headRef = SortedMerge(a, b);
```

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* function prototypes */
struct node* SortedMerge(struct node* a, struct node* b);
void FrontBackSplit(struct node* source,
                    struct node** frontRef, struct node** backRef);

/* sorts the linked list by changing next pointers (not data) */
void MergeSort(struct node** headRef)
{
    struct node* head = *headRef;
    struct node* a;
    struct node* b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}

/* See http://geeksforgeeks.org/?p=3622 for details of this
```

```

function */
struct node* SortedMerge(struct node* a, struct node* b)
{
    struct node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return(b);
    else if (b==NULL)
        return(a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return(result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
   and return the two lists using the reference parameters.
   If the length is odd, the extra node should go in the front list.
   Uses the fast/slow pointer strategy. */
void FrontBackSplit(struct node* source,
                     struct node** frontRef, struct node** backRef)
{
    struct node* fast;
    struct node* slow;
    if (source==NULL || source->next==NULL)
    {
        /* length < 2 cases */
        *frontRef = source;
        *backRef = NULL;
    }
    else
    {
        slow = source;
        fast = source->next;

        /* Advance 'fast' two nodes, and advance 'slow' one node */
        while (fast != NULL)
        {
            fast = fast->next;
            if (fast != NULL)
            {
                slow = slow->next;
                fast = fast->next;
            }
        }

        /* 'slow' is before the midpoint in the list, so split it in two
           at that point. */
        *frontRef = source;
        *backRef = slow->next;
        slow->next = NULL;
    }
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

```

```

/* Function to insert a node at the beginning of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* res = NULL;
    struct node* a = NULL;

    /* Let us create a unsorted linked lists to test the functions
     * Created lists shall be a:2->3->20->5->10->15 */
    push(&a, 15);
    push(&a, 10);
    push(&a, 5);
    push(&a, 20);
    push(&a, 3);
    push(&a, 2);

    /* Sort the above created Linked List */
    MergeSort(&a);

    printf("\n Sorted Linked List is:\n");
    printList(a);

    getchar();
    return 0;
}

```

Time Complexity: O(nLogn)

Sources:

http://en.wikipedia.org/wiki/Merge_sort

<http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Linked Lists
Sorting
Merge Sort

Select a Random Node from a Singly Linked List

Given a singly linked list, select a random node from linked list (the probability of picking a node should be 1/N if there are N nodes in list). You are given a random number generator.

Below is a Simple Solution

- 1) Count number of nodes by traversing the list.
- 2) Traverse the list again and select every node with probability 1/N. The selection can be done by generating a random number from 0 to N-i for i'th node, and selecting the i'th node node only if generated number is equal to 0 (or any other fixed number from 0 to N-i).

We get uniform probabilities with above schemes.

i = 1, probability of selecting first node = 1/N
i = 2, probability of selecting second node =
[probability that first node is not selected] *
[probability that second node is selected]

$$= ((N-1)/N) * 1/(N-1)$$

$$= 1/N$$

Similarly, probabilities of other selecting other nodes is $1/N$

The above solution requires two traversals of linked list.

How to select a random node with only one traversal allowed?

The idea is to use [Reservoir Sampling](#). Following are the steps. This is a simpler version of [Reservoir Sampling](#) as we need to select only one key instead of k keys.

- (1) Initialize result as first node
result = head->key
- (2) Initialize n = 2
- (3) Now one by one consider all nodes from 2nd node onward.
 - (3.a) Generate a random number from 0 to n-1.
Let the generated random number is j.
 - (3.b) If j is equal to 0 (we could choose other fixed number between 0 to n-1), then replace result with current node.
 - (3.c) n = n+1
 - (3.d) current = current->next

Below is the implementation of above algorithm.

C

```
/* C program to randomly select a node from a singly
linked list */
#include<stdio.h>
#include<stdlib.h>
#include <time.h>

/* Link list node */
struct node
{
    int key;
    struct node* next;
};

// A reservoir sampling based function to print a
// random node from a linked list
void printRandom(struct node *head)
{
    // IF list is empty
    if(head == NULL)
        return;

    // Use a different seed value so that we don't get
    // same result each time we run this program
    srand(time(NULL));

    // Initialize result as first node
    int result = head->key;

    // Iterate from the (k+1)th element to nth element
    struct node *current = head;
    int n;
    for(n=2; current!=NULL; n++)
    {
        // change result with probability 1/n
        if(rand() % n == 0)
            result = current->key;

        // Move to next node
        current = current->next;
    }

    printf("Randomly selected key is %d\n", result);
}

/* BELOW FUNCTIONS ARE JUST UTILITY TO TEST */

/* A utility function to create a new node */
struct node *newNode(int new_key)
```

```
{
/* allocate node */
struct node* new_node =
    (struct node*) malloc(sizeof(struct node));

/* put in the key */
new_node->key = new_key;
new_node->next = NULL;

return new_node;
}

/* A utility function to insert a node at the beginning
of linked list */
void push(struct node** head_ref, int new_key)
{
    /* allocate node */
    struct node* new_node = new node;

    /* put in the key */
    new_node->key = new_key;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// Driver program to test above functions
int main()
{
    struct node *head = NULL;
    push(&head, 5);
    push(&head, 20);
    push(&head, 4);
    push(&head, 3);
    push(&head, 30);

    printRandom(head);

    return 0;
}
```

Java

```
// Java program to select a random node from singly linked list

import java.util.*;

// Linked List Class
class LinkedList {

    static Node head; // head of list

    /* Node Class */
    static class Node {

        int data;
        Node next;

        // Constructor to create a new node
        Node(int d) {
            data = d;
            next = null;
        }
    }

    // A reservoir sampling based function to print a
    // random node from a linked list
    void printrandom(Node node) {
```

```

// If list is empty
if (node == null) {
    return;
}

// Use a different seed value so that we don't get
// same result each time we run this program
Math.abs(UUID.randomUUID().getMostSignificantBits());

// Initialize result as first node
int result = node.data;

// Iterate from the (k+1)th element to nth element
Node current = node;
int n;
for (n = 2; current != null; n++) {

    // change result with probability 1/n
    if (Math.random() % n == 0) {
        result = current.data;
    }

    // Move to next node
    current = current.next;
}

System.out.println("Randomly selected key is " + result);
}

// Driver program to test above functions
public static void main(String[] args) {

    LinkedList list = new LinkedList();
    list.head = new Node(5);
    list.head.next = new Node(20);
    list.head.next.next = new Node(4);
    list.head.next.next.next = new Node(3);
    list.head.next.next.next.next = new Node(30);

    list.printrandom(head);

}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to randomly select a node from singly
# linked list

import random

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data= data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # A reservoir sampling based function to print a
    # random node from a linked list
    def printRandom(self):

```

```

# If list is empty
if self.head is None:
    return

# Use a different seed value so that we don't get
# same result each time we run this program
random.seed()

# Initialize result as first node
result = self.head.data

# Iterate from the (k+1)th element nth element
current = self.head
n = 2
while(current is not None):

    # change result with probability 1/n
    if (random.randrange(n) == 0):
        result = current.data

    # Move to next node
    current = current.next
    n += 1

print "Randomly selected key is %d" %(result)

# Function to insert a new node at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program to test above function
llist = LinkedList()
llist.push(5)
llist.push(20)
llist.push(4)
llist.push(3)
llist.push(30)
llist.printRandom()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Note that the above program is based on outcome of a random function and may produce different output.

How does this work?

Let there be total N nodes in list. It is easier to understand from last node.

The probability that last node is result simply $1/N$ [For last or N^{th} node, we generate a random number between 0 to $N-1$ and make last node as result if the generated number is 0 (or any other fixed number)]

The probability that second last node is result should also be $1/N$.

The probability that the second last node is result
 $= [\text{Probability that the second last node replaces result}] \times$
 $[\text{Probability that the last node doesn't replace the result}]$
 $= [1 / (N-1)] \times [(N-1)/N]$
 $= 1/N$

Similarly we can show probability for 3rd last node and other nodes.

This article is contributed by **Rajeev**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Dynamic Programming | Set 4 (Longest Common Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in [Set 1](#) and [Set 2](#) respectively. We also discussed one example problem in [Set 3](#). Let us discuss Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, “acefg”, .. etc are subsequences of “abcdefg”. So a string of length n has 2^n different possible subsequences.

It is a classic computer science problem, the basis of [diff](#) (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

Examples:

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

We strongly recommend that you click here and practice it, before moving on to the solution.

The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

1) Optimal Substructure:

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

If last characters of both sequences match (or $X[m-1] == Y[n-1]$) then

$$L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])$$

If last characters of both sequences do not match (or $X[m-1] != Y[n-1]$) then

$$L(X[0..m-1], Y[0..n-1]) = \text{MAX} (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))$$

Examples:

1) Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:

$$L(\text{“AGGTAB”}, \text{“GXTXAYB”}) = 1 + L(\text{“AGGTA”}, \text{“GXTXAY”})$$

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

2) Consider the input strings “ABCDGH” and “AEDFHR”. Last characters do not match for the strings. So length of LCS can be written as:

$$L(\text{“ABCDGH”}, \text{“AEDFHR”}) = \text{MAX} (L(\text{“ABCDG”}, \text{“AEDFHR”}), L(\text{“ABCDGH”}, \text{“AEDFH”}))$$

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

2) Overlapping Subproblems:

Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

C/C++

```
/* A Naive recursive implementation of LCS problem */
#include<bits/stdc++.h>
```

```

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
    if (m == 0 || n == 0)
        return 0;
    if (X[m-1] == Y[n-1])
        return 1 + lcs(X, Y, m-1, n-1);
    else
        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

    return 0;
}

```

Python

```

# A Naive recursive Python implementation of LCS problem

def lcs(X, Y, m, n):

    if m == 0 or n == 0:
        return 0;
    elif X[m-1] == Y[n-1]:
        return 1 + lcs(X, Y, m-1, n-1);
    else:
        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));

# Driver program to test the above function
X = "AGGTAB"
Y = "GXTXAYB"
print "Length of LCS is ", lcs(X , Y, len(X), len(Y))

```

Output:

Length of LCS is 4

Time complexity of the above naive recursive approach is $O(2^n)$ in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.

Considering the above implementation, following is a partial recursion tree for input strings “AXYT” and “AYZX”

```

lcs("AXYT", "AYZX")
  /   \
lcs("AXY", "AYZX")   lcs("AXYT", "AYZ")
  /   \   /   \
lcs("AX", "AYZX") lcs("AXY", "AYZ") lcs("AXY", "AYZ") lcs("AXYT", "AY")

```

In the above partial recursion tree, $\text{lcs}(\text{“AXY”, “AYZ”})$ is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabulated implementation for the LCS problem.

C/C++

```

/* Dynamic Programming C/C++ implementation of LCS problem */
#include<bits/stdc++.h>

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
    int L[m+1][n+1];
    int i, j;

    /* Following steps build L[m+1][n+1] in bottom up fashion. Note
       that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] */
    for (i=0; i<=m; i++)
    {
        for (j=0; j<=n; j++)
        {
            if (i == 0 || j == 0)
                L[i][j] = 0;

            else if (X[i-1] == Y[j-1])
                L[i][j] = L[i-1][j-1] + 1;

            else
                L[i][j] = max(L[i-1][j], L[i][j-1]);
        }
    }

    /* L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1] */
    return L[m][n];
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

    return 0;
}

```

Python

```

# Dynamic Programming implementation of LCS problem

def lcs(X , Y):
    # find the length of the strings
    m = len(X)
    n = len(Y)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in xrange(m+1)]

    """Following steps build L[m+1][n+1] in bottom up fashion
    Note: L[i][j] contains length of LCS of X[0..i-1]
    and Y[0..j-1]"""
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j], L[i][j-1])

```

```

else:
    L[i][j] = max(L[i-1][j], L[i][j-1])

# L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
return L[m][n]
#endif of function lcs

# Driver program to test the above function
X = "AGGTAB"
Y = "GXTXAYB"
print "Length of LCS is ", lcs(X, Y)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Time Complexity of the above implementation is $O(mn)$ which is much better than the worst case time complexity of Naive Recursive implementation.

The above algorithm/code returns only length of LCS. Please see the following post for printing the LCS.

[Printing Longest Common Subsequence](#)

You can also check the space optimized version of LCS at

[Space Optimized Solution of LCS](#)



Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

- <http://www.youtube.com/watch?v=V5hZoJ6uK-s>
- http://www.algorithmist.com/index.php/Longest_Common_Subsequence
- <http://www.ics.uci.edu/~eppstein/161/960229.html>
- http://en.wikipedia.org/wiki/Longest_common_subsequence_problem

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming
LCS
subsequence

Dynamic Programming | Set 3 (Longest Increasing Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in [Set 1](#) and [Set 2](#) respectively.

Let us discuss Longest Increasing Subsequence (LIS) problem as an example problem that can be solved using Dynamic Programming.

The Longest Increasing Subsequence (LIS) problem is to 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, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}.

arr[]	10	22	9	33	21	50	41	60	80
LIS	1	2		3		4		5	6

More Examples:

```
Input : arr[] = {3, 10, 2, 1, 20}
Output : Length of LIS = 3
The longest increasing subsequence is 3, 10, 20

Input : arr[] = {3, 2}
Output : Length of LIS = 1
The longest increasing subsequences are {3} and {2}

Input : arr[] = {50, 3, 10, 7, 40, 80}
Output : Length of LIS = 4
The longest increasing subsequence is {3, 7, 40, 80}
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Optimal Substructure:

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

Then, $L(i)$ can be recursively written as:

$L(i) = 1 + \max(L(j))$ where $0 < j < i$ and $\text{arr}[j] < \text{arr}[i]$; or

$L(i) = 1$, if no such j exists.

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

Thus, we see the LIS problem satisfies the optimal substructure property as the main problem can be solved using solutions to subproblems.

Following is a simple recursive implementation of the LIS problem. It follows the recursive structure discussed above.

C/C++

```
// A naive C/C++ based recursive implementation of LIS problem
#include<stdio.h>
#include<stdlib.h>

// Recursive implementation for calculating the LIS
int _lis(int arr[], int n, int *max_lis_length)
{
    // Base case
    if (n == 1)
        return 1;

    int current_lis_length = 1;
    for (int i=0; i<n-1; i++)
    {
        // Recursively calculate the length of the LIS
        // ending at arr[i]
        int subproblem_lis_length = _lis(arr, i, max_lis_length);

        // Check if appending arr[n-1] to the LIS
        // ending at arr[i] gives us an LIS ending at
        // arr[n-1] which is longer than the previously
        // calculated LIS ending at arr[n-1]
        if (arr[i] < arr[n-1] &&
            current_lis_length < (1+subproblem_lis_length))
            current_lis_length = 1+subproblem_lis_length;
    }

    // Check if currently calculated LIS ending at
    // arr[n-1] is longer than the previously calculated
    // LIS and update max_lis_length accordingly
    if (*max_lis_length < current_lis_length)
        *max_lis_length = current_lis_length;

    return current_lis_length;
}

// The wrapper function for _lis()
```

```

int lis(int arr[], int n)
{
    int max_lis_length = 1; // stores the final LIS

    // max_lis_length is passed as a reference below
    // so that it can maintain its value
    // between the recursive calls
    _lis( arr, n, &max_lis_length );

    return max_lis_length;
}

// Driver program to test the functions above
int main()
{
    int arr[] = {10, 22, 9, 33, 21, 50, 41, 60};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of LIS is %d\n", lis( arr, n ) );
    return 0;
}

```

Java

```

// A naive Java based recursive implementation of LIS problem
class LIS
{
    static int max_lis_length; // stores the final LIS

    // Recursive implementation for calculating the LIS
    static int _lis(int arr[], int n)
    {
        // base case
        if (n == 1)
            return 1;

        int current_lis_length = 1;
        for (int i=0; i<n-1; i++)
        {
            // Recursively calculate the length of the LIS
            // ending at arr[i]
            int subproblem_lis_length = _lis(arr, i);

            // Check if appending arr[n-1] to the LIS
            // ending at arr[i] gives us an LIS ending at
            // arr[n-1] which is longer than the previously
            // calculated LIS ending at arr[n-1]
            if (arr[i] < arr[n-1] &&
                current_lis_length < (1+subproblem_lis_length))
                current_lis_length = 1+subproblem_lis_length;
        }

        // Check if currently calculated LIS ending at
        // arr[n-1] is longer than the previously calculated
        // LIS and update max_lis_length accordingly
        if (max_lis_length < current_lis_length)
            max_lis_length = current_lis_length;

        return current_lis_length;
    }

    // The wrapper function for _lis()
    static int lis(int arr[], int n)
    {
        max_lis_length = 1; // stores the final LIS

        // max_lis_length is declared static above
        // so that it can maintain its value
        // between the recursive calls of _lis()
        _lis( arr, n );

        return max_lis_length;
    }
}

```

```

// Driver program to test the functions above
public static void main(String args[])
{
    int arr[] = {10, 22, 9, 33, 21, 50, 41, 60};
    int n = arr.length;
    System.out.println("Length of LIS is " + lis( arr, n ));
}

} // End of LIS class.

// This code is contributed by Rajat Mishra

```

Python

```

# A naive Python based recursive implementation of LIS problem

global max_lis_length # stores the final LIS

# Recursive implementation for calculating the LIS
def _lis(arr, n):
    # Following declaration is needed to allow modification
    # of the global copy of max_lis_length in _lis()
    global max_lis_length

    # Base Case
    if n == 1:
        return 1

    current_lis_length = 1

    for i in xrange(0, n-1):
        # Recursively calculate the length of the LIS
        # ending at arr[i]
        subproblem_lis_length = _lis(arr, i)

        # Check if appending arr[n-1] to the LIS
        # ending at arr[i] gives us an LIS ending at
        # arr[n-1] which is longer than the previously
        # calculated LIS ending at arr[n-1]
        if arr[i] < arr[n-1] and \
           current_lis_length < (1+subproblem_lis_length):
            current_lis_length = (1+subproblem_lis_length)

    # Check if currently calculated LIS ending at
    # arr[n-1] is longer than the previously calculated
    # LIS and update max_lis_length accordingly
    if (max_lis_length < current_lis_length):
        max_lis_length = current_lis_length

    return current_lis_length

# The wrapper function for _lis()
def lis(arr, n):

    # Following declaration is needed to allow modification
    # of the global copy of max_lis_length in lis()
    global max_lis_length

    max_lis_length = 1 # stores the final LIS

    # max_lis_length is declared global at the top
    # so that it can maintain its value
    # between the recursive calls of _lis()
    _lis(arr, n)

    return max_lis_length

# Driver program to test the functions above
def main():
    arr = [10, 22, 9, 33, 21, 50, 41, 60]
    n = len(arr)
    print "Length of LIS is", lis(arr, n)

```

```

if __name__=="__main__":
    main()

# This code is contributed by NIKHIL KUMAR SINGH

```

Output:

Length of LIS is 5

Overlapping Subproblems:

Considering the above implementation, following is recursion tree for an array of size 4. lis(n) gives us the length of LIS for arr[].

```

        lis(4)
        /   |   \
lis(3)  lis(2)  lis(1)
/ \   /
lis(2) lis(1) lis(1)
/
lis(1)

```

We can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabulated implementation for the LIS problem.

C/C++

```

/* Dynamic Programming C/C++ implementation of LIS problem */
#include<stdio.h>
#include<stdlib.h>

/* lis() returns the length of the longest increasing
subsequence in arr[] of size n */
int lis( int arr[], int n )
{
    int *lis, i, j, max = 0;
    lis = (int*) malloc ( sizeof( int ) * n );

    /* Initialize LIS values for all indexes */
    for (i = 0; i < n; i++)
        lis[i] = 1;

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

    /* Pick maximum of all LIS values */
    for (i = 0; i < n; i++)
        if (max < lis[i])
            max = lis[i];

    /* Free memory to avoid memory leak */
    free(lis);

    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of lis is %d\n", lis( arr, n ) );
    return 0;
}

```

Java

```

/* Dynamic Programming Java implementation of LIS problem */

```

```

class LIS
{
    /* lis() returns the length of the longest increasing
       subsequence in arr[] of size n */
    static int lis(int arr[],int n)
    {
        int lis[] = new int[n];
        int i,j,max = 0;

        /* Initialize LIS values for all indexes */
        for ( i = 0; i < n; i++ )
            lis[i] = 1;

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

        /* Pick maximum of all LIS values */
        for ( i = 0; i < n; i++ )
            if ( max < lis[i] )
                max = lis[i];

        return max;
    }

    public static void main(String args[])
    {
        int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
        int n = arr.length;
        System.out.println("Length of lis is " + lis( arr, n ) + "\n");
    }
}

/*This code is contributed by Rajat Mishra*/

```

Python

```

# Dynamic programming Python implementation of LIS problem

# lis returns length of the longest increasing subsequence
# in arr of size n
def lis(arr):
    n = len(arr)

    # Declare the list (array) for LIS and initialize LIS
    # values for all indexes
    lis = [1]*n

    # Compute optimized LIS values in bottom up manner
    for i in range (1 , n):
        for j in range(0 , i):
            if arr[i] > arr[j] and lis[i]< lis[j] + 1 :
                lis[i] = lis[j]+1

    # Initialize maximum to 0 to get the maximum of all
    # LIS
    maximum = 0

    # Pick maximum of all LIS values
    for i in range(n):
        maximum = max(maximum , lis[i])

    return maximum
# end of lis function

# Driver program to test above function
arr = [10, 22, 9, 33, 21, 50, 41, 60]
print "Length of lis is",lis(arr)
# This code is contributed by Nikhil Kumar Singh

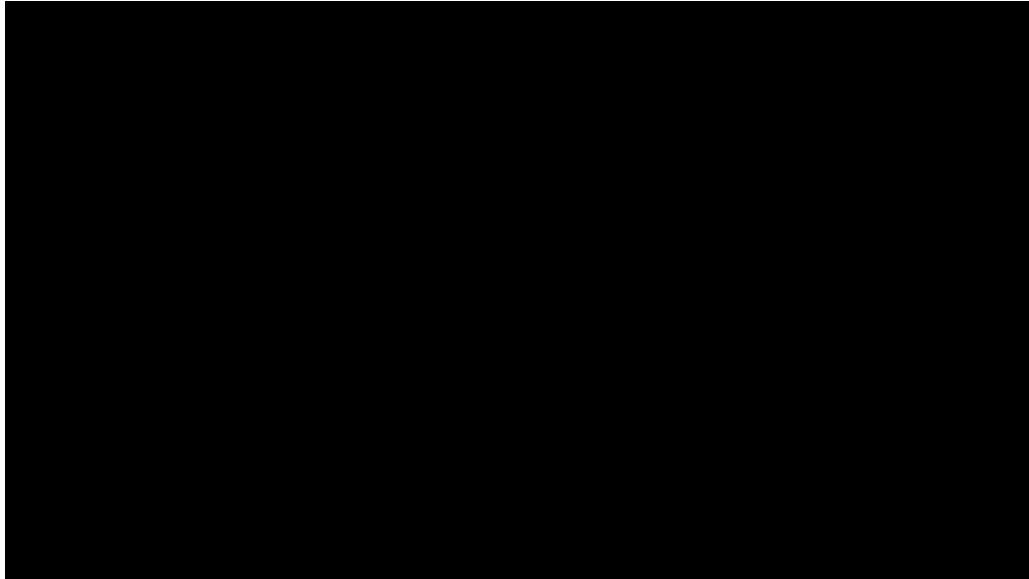
```

Output:

Length of lis is 5

Note that the time complexity of the above Dynamic Programming (DP) solution is $O(n^2)$ and there is a $O(n \log n)$ solution for the LIS problem. We have not discussed the $O(n \log n)$ solution here as the purpose of this post is to explain Dynamic Programming with a simple example. See below post for $O(n \log n)$ solution.

Longest Increasing Subsequence Size ($N \log N$)



Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming
LIS

Dynamic Programming | Set 5 (Edit Distance)

Given two strings str1 and str2 and below operations that can be performed on str1. Find minimum number of edits (operations) required to convert 'str1' into 'str2'.

- a. Insert
- b. Remove
- c. Replace

All of the above operations are of equal cost.

Examples:

Input: str1 = "geek", str2 = "gesek"

Output: 1

We can convert str1 into str2 by inserting a 's'.

Input: str1 = "cat", str2 = "cut"

Output: 1

We can convert str1 into str2 by replacing 'a' with 'u'.

Input: str1 = "sunday", str2 = "saturday"

Output: 3

Last three and first characters are same. We basically need to convert "un" to "atur". This can be done using below three operations.

Replace 'n' with 'r', insert t, insert a

What are the subproblems in this case?

The idea is process all characters one by one starting from either from left or right sides of both strings.

Let's traverse from right corner, there are two possibilities for every pair of character being traversed.

m: Length of str1 (first string)

n: Length of str2 (second string)

1. If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
2. Else (If last characters are not same), we consider all operations on 'str1', consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.
 - a. Insert: Recur for m and n-1
 - b. Remove: Recur for m-1 and n
 - c. Replace: Recur for m-1 and n-1

Below is C++ implementation of above Naive recursive solution.

C++

```
// A Naive recursive C++ program to find minimum number
// operations to convert str1 to str2
#include<bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers
int min(int x, int y, int z)
{
    return min(min(x, y), z);
}

int editDist(string str1 , string str2 , int m ,int n)
{
    // If first string is empty, the only option is to
    // insert all characters of second string into first
    if (m == 0) return n;

    // If second string is empty, the only option is to
    // remove all characters of first string
    if (n == 0) return m;

    // If last characters of two strings are same, nothing
    // much to do. Ignore last characters and get count for
    // remaining strings.
    if (str1[m-1] == str2[n-1])
        return editDist(str1, str2, m-1, n-1);

    // If last characters are not same, consider all three
    // operations on last character of first string, recursively
    // compute minimum cost for all three operations and take
    // minimum of three values.
    return 1 + min ( editDist(str1, str2, m, n-1), // Insert
                     editDist(str1, str2, m-1, n), // Remove
                     editDist(str1, str2, m-1, n-1) // Replace
                   );
}

// Driver program
int main()
{
    // your code goes here
    string str1 = "sunday";
    string str2 = "saturday";

    cout << editDist( str1 , str2 , str1.length(), str2.length());

    return 0;
}
```

Java

```
// A Naive recursive Java program to find minimum number
// operations to convert str1 to str2
class EDIST
{
    static int min(int x,int y,int z)
    {
```

```

if (x<y && x<z) return x;
if (y<x && y<z) return y;
else return z;
}

static int editDist(String str1 , String str2 , int m ,int n)
{
    // If first string is empty, the only option is to
    // insert all characters of second string into first
    if (m == 0) return n;

    // If second string is empty, the only option is to
    // remove all characters of first string
    if (n == 0) return m;

    // If last characters of two strings are same, nothing
    // much to do. Ignore last characters and get count for
    // remaining strings.
    if (str1.charAt(m-1) == str2.charAt(n-1))
        return editDist(str1, str2, m-1, n-1);

    // If last characters are not same, consider all three
    // operations on last character of first string, recursively
    // compute minimum cost for all three operations and take
    // minimum of three values.
    return 1 + min ( editDist(str1 , str2, m, n-1),   // Insert
                    editDist(str1, str2, m-1, n), // Remove
                    editDist(str1, str2, m-1, n-1) // Replace
                );
}

public static void main(String args[])
{
    String str1 = "sunday";
    String str2 = "saturday";

    System.out.println( editDist( str1 , str2 , str1.length(), str2.length() ) );
}
}
/*This code is contributed by Rajat Mishra*/

```

Python

```

# A Naive recursive Python program to fin minimum number
# operations to convert str1 to str2
def editDistance(str1, str2, m ,n):

    # If first string is empty, the only option is to
    # insert all characters of second string into first
    if m==0:
        return n

    # If second string is empty, the only option is to
    # remove all characters of first string
    if n==0:
        return m

    # If last characters of two strings are same, nothing
    # much to do. Ignore last characters and get count for
    # remaining strings.
    if str1[m-1]==str2[n-1]:
        return editDistance(str1,str2,m-1,n-1)

    # If last characters are not same, consider all three
    # operations on last character of first string, recursively
    # compute minimum cost for all three operations and take
    # minimum of three values.
    return 1 + min(editDistance(str1, str2, m, n-1),  # Insert
                  editDistance(str1, str2, m-1, n), # Remove
                  editDistance(str1, str2, m-1, n-1) # Replace
                 )

# Driver program to test the above function

```

```

str1 = "sunday"
str2 = "saturday"
print editDistance(str1, str2, len(str1), len(str2))

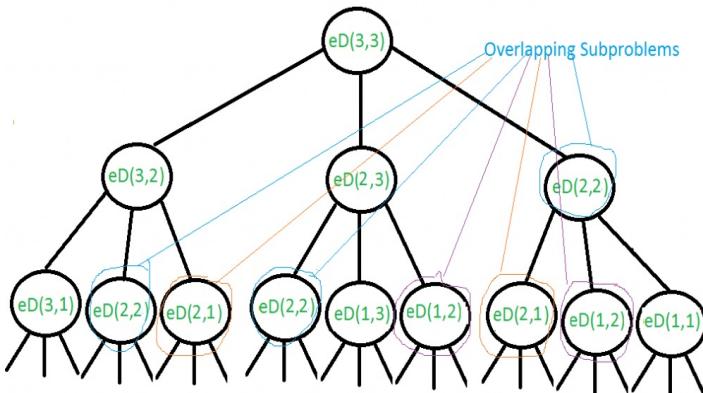
```

This code is contributed by Bhavya Jain

Output:

3

The time complexity of above solution is exponential. In worst case, we may end up doing $O(3^m)$ operations. The worst case happens when none of characters of two strings match. Below is a recursive call diagram for worst case.



Worst case recursion tree when $m = 3, n = 3$.
Worst case example str1="abc" str2="xyz"

We can see that many subproblems are solved again and again, for example $eD(2,2)$ is called three times. Since same subproblems are called again, this problem has Overlapping Subproblems property. So Edit Distance problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array that stores results of subproblems.

C++

```

// A Dynamic Programming based C++ program to find minimum
// number operations to convert str1 to str2
#include<bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers
int min(int x, int y, int z)
{
    return min(min(x, y), z);
}

int editDistDP(string str1, string str2, int m, int n)
{
    // Create a table to store results of subproblems
    int dp[m+1][n+1];

    // Fill d[][], in bottom up manner
    for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            // If first string is empty, only option is to
            // insert all characters of second string
            if (i==0)
                dp[i][j] = j; // Min. operations = j

            // If second string is empty, only option is to
            // remove all characters of second string
            else if (j==0)
                dp[i][j] = i; // Min. operations = i

            // If last characters are same, ignore last char
            // and recur for remaining string
            else if (str1[i-1] == str2[j-1])
                dp[i][j] = dp[i-1][j-1];
            else
                dp[i][j] = 1 + min(dp[i-1][j],      // Insert
                                   dp[i][j-1],      // Remove
                                   dp[i-1][j-1]);   // Replace
        }
    }

    // Return minimum operations required
    return dp[m][n];
}

```

```

// If last character are different, consider all
// possibilities and find minimum
else
    dp[i][j] = 1 + min(dp[i][j-1], // Insert
                        dp[i-1][j], // Remove
                        dp[i-1][j-1]); // Replace
}
}

return dp[m][n];
}

// Driver program
int main()
{
    // your code goes here
    string str1 = "sunday";
    string str2 = "saturday";

    cout << editDistDP(str1, str2, str1.length(), str2.length());

    return 0;
}

```

Java

```

// A Dynamic Programming based Java program to find minimum
// number operations to convert str1 to str2
class EDIST
{
    static int min(int x,int y,int z)
    {
        if (x < y && x <z) return x;
        if (y < x && y < z) return y;
        else return z;
    }

    static int editDistDP(String str1, String str2, int m, int n)
    {
        // Create a table to store results of subproblems
        int dp[][] = new int[m+1][n+1];

        // Fill d[][] in bottom up manner
        for (int i=0; i<=m; i++)
        {
            for (int j=0; j<=n; j++)
            {
                // If first string is empty, only option is to
                // insert all characters of second string
                if (i==0)
                    dp[i][j] = j; // Min. operations = j

                // If second string is empty, only option is to
                // remove all characters of second string
                else if (j==0)
                    dp[i][j] = i; // Min. operations = i

                // If last characters are same, ignore last char
                // and recur for remaining string
                else if (str1.charAt(i-1) == str2.charAt(j-1))
                    dp[i][j] = dp[i-1][j-1];

                // If last character are different, consider all
                // possibilities and find minimum
                else
                    dp[i][j] = 1 + min(dp[i][j-1], // Insert
                                        dp[i-1][j], // Remove
                                        dp[i-1][j-1]); // Replace
            }
        }

        return dp[m][n];
    }
}

```

```

}

public static void main(String args[])
{
    String str1 = "sunday";
    String str2 = "saturday";
    System.out.println( editDistDP( str1 , str2 , str1.length(), str2.length() ) );
}
/*This code is contributed by Rajat Mishra*/

```

Python

```

# A Dynamic Programming based Python program for edit
# distance problem
def editDistDP(str1, str2, m, n):
    # Create a table to store results of subproblems
    dp = [[0 for x in range(n+1)] for x in range(m+1)]

    # Fill dp[][], in bottom up manner
    for i in range(m+1):
        for j in range(n+1):

            # If first string is empty, only option is to
            # insert all characters of second string
            if i == 0:
                dp[i][j] = j    # Min. operations = j

            # If second string is empty, only option is to
            # remove all characters of second string
            elif j == 0:
                dp[i][j] = i    # Min. operations = i

            # If last characters are same, ignore last char
            # and recur for remaining string
            elif str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1]

            # If last character are different, consider all
            # possibilities and find minimum
            else:
                dp[i][j] = 1 + min(dp[i][j-1],      # Insert
                                    dp[i-1][j],      # Remove
                                    dp[i-1][j-1])    # Replace

    return dp[m][n]

# Driver program
str1 = "sunday"
str2 = "saturday"

print(editDistDP(str1, str2, len(str1), len(str2)))
# This code is contributed by Bhavya Jain

```

Output:

3

Time Complexity: $O(m \times n)$

Auxiliary Space: $O(m \times n)$

Applications: There are many practical applications of edit distance algorithm, refer [Lucene API](#) for sample. Another example, display all the words in a dictionary that are near proximity to a given word\incorrectly spelled word.

Thanks to Vivek Kumar for suggesting above updates.

Thanks to **Venki** for providing initial post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Partition a set into two subsets such that the difference of subset sums is minimum

Given a set of integers, the task is to divide it into two sets S1 and S2 such that the absolute difference between their sums is minimum.

If there is a set S with n elements, then if we assume Subset1 has m elements, Subset2 must have n-m elements and the value of $\text{abs}(\text{sum}(\text{Subset1}) - \text{sum}(\text{Subset2}))$ should be minimum.

Example:

```
Input: arr[] = {1, 6, 11, 5}
Output: 1
Explanation:
Subset1 = {1, 5, 6}, sum of Subset1 = 12
Subset2 = {11}, sum of Subset2 = 11
```

We strongly recommend that you click here and practice it, before moving on to the solution.

This problem is mainly an extension to the [Dynamic Programming| Set 18 \(Partition Problem\)](#).

Recursive Solution

The recursive approach is to generate all possible sums from all the values of array and to check which solution is the most optimal one. To generate sums we either include the i'th item in set 1 or don't include, i.e., include in set 2.

```
// A Recursive C program to solve minimum sum partition
// problem.
#include <bits/stdc++.h>
using namespace std;

// Function to find the minimum sum
int findMinRec(int arr[], int i, int sumCalculated, int sumTotal)
{
    // If we have reached last element. Sum of one
    // subset is sumCalculated, sum of other subset is
    // sumTotal-sumCalculated. Return absolute difference
    // of two sums.
    if (i==0)
        return abs((sumTotal-sumCalculated) - sumCalculated);

    // For every item arr[i], we have two choices
    // (1) We do not include it first set
    // (2) We include it in first set
    // We return minimum of two choices
    return min(findMinRec(arr, i-1, sumCalculated+arr[i-1], sumTotal),
               findMinRec(arr, i-1, sumCalculated, sumTotal));
}

// Returns minimum possible difference between sums
// of two subsets
int findMin(int arr[], int n)
{
    // Compute total sum of elements
    int sumTotal = 0;
    for (int i=0; i<n; i++)
        sumTotal += arr[i];

    // Compute result using recursive function
    return findMinRec(arr, n, 0, sumTotal);
}

// Driver program to test above function
int main()
{
    int arr[] = {3, 1, 4, 2, 2, 1};
```

```

int n = sizeof(arr)/sizeof(arr[0]);
cout << "The minimum difference between two sets is "
     << findMin(arr, n);
return 0;
}

```

Output:

The minimum difference between two sets is 1

Time Complexity:

All the sums can be generated by either
(1) including that element in set 1.
(2) without including that element in set 1.
So possible combinations are :-
arr[0] (1 or 2) -> 2 values
arr[1] (1 or 2) -> 2 values
.
.
arr[n] (2 or 2) -> 2 values
So time complexity will be $2^*2^*....^2$ (For n times),
that is $O(2^n)$.

Dynamic Programming

The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array $dp[n+1][sum+1]$ where n is number of elements in given set and sum is sum of all elements. We can construct the solution in bottom up manner.

The task is to divide the set into two parts.

We will consider the following factors for dividing it.

Let

$dp[n+1][sum+1] = \{1 \text{ if some subset from 1st to } i\text{'th has a sum}$
equal to j
0 otherwise}

i ranges from {1..n}

j ranges from {0..(sum of all elements)}

So

$dp[n+1][sum+1]$ will be 1 if
1) The sum j is achieved including i'th item
2) The sum j is achieved excluding i'th item.

Let sum of all the elements be S.

To find Minimum sum difference, we have to find j such
that $\min\{sum - j^*2 : dp[n][j] == 1\}$
where j varies from 0 to sum/2

The idea is, sum of S1 is j and it should be closest
to sum/2, i.e., 2^*j should be closest to sum.

Below is C++ implementation of above code.

```

// A Recursive C program to solve minimum sum partition
// problem.
#include <bits/stdc++.h>
using namespace std;

// Returns the minimum value of the difference of the two sets.
int findMin(int arr[], int n)
{
    // Calculate sum of all elements
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += arr[i];

    // Create an array to store results of subproblems
    bool dp[n+1][sum+1];

    // Initialize first column as true. 0 sum is possible
    // with all elements.
    for (int i = 0; i <= n; i++)
        dp[i][0] = true;

```

```

// Initialize top row, except dp[0][0], as false. With
// 0 elements, no other sum except 0 is possible
for (int i = 1; i <= sum; i++)
    dp[0][i] = false;

// Fill the partition table in bottom up manner
for (int i=1; i<=n; i++)
{
    for (int j=1; j<=sum; j++)
    {
        // If i'th element is excluded
        dp[i][j] = dp[i-1][j];

        // If i'th element is included
        if (arr[i-1] <= j)
            dp[i][j] |= dp[i-1][j-arr[i-1]];
    }
}

// Initialize difference of two sums.
int diff = INT_MAX;

// Find the largest j such that dp[n][j]
// is true where j loops from sum/2 to 0
for (int j=sum/2; j>=0; j--)
{
    // Find the
    if (dp[n][j] == true)
    {
        diff = sum-2*j;
        break;
    }
}
return diff;
}

// Driver program to test above function
int main()
{
    int arr[] = {3, 1, 4, 2, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "The minimum difference between 2 sets is "
         << findMin(arr, n);
    return 0;
}

```

Output:

The minimum difference between 2 sets is 1

Time Complexity = O(n*sum) where n is number of elements and sum is sum of all elements.

Note that the above solution is in Pseudo Polynomial Time (time complexity is dependent on numeric value of input).

This article is contributed by **Abhiraj Smit**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Count number of ways to cover a distance

Given a distance 'dist', count total number of ways to cover the distance with 1, 2 and 3 steps.

Examples:

Input: n = 3
Output: 4
Below are the four ways
1 step + 1 step + 1 step
1 step + 2 step

```
2 step + 1 step  
3 step
```

```
Input: n = 4  
Output: 7
```

We strongly recommend you to minimize your browser and try this yourself first.

```
// A naive recursive C++ program to count number of ways to cover  
// a distance with 1, 2 and 3 steps  
#include<iostream>  
using namespace std;  
  
// Returns count of ways to cover 'dist'  
int printCountRec(int dist)  
{  
    // Base cases  
    if (dist<0)  return 0;  
    if (dist==0) return 1;  
  
    // Recur for all previous 3 and add the results  
    return printCountRec(dist-1) +  
           printCountRec(dist-2) +  
           printCountRec(dist-3);  
}  
  
// driver program  
int main()  
{  
    int dist = 4;  
    cout << printCountRec(dist);  
    return 0;  
}
```

Output:

```
7
```

The time complexity of above solution is exponential, a close upper bound is $O(3^n)$. If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. For example, when we start from $n = 6$, we can reach 4 by subtracting one 2 times and by subtracting 2 one times. So the subproblem for 4 is called twice.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So min square sum problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical [Dynamic Programming\(DP\) problems](#), recomputations of same subproblems can be avoided by constructing a temporary array `count[]` in bottom up manner.

Below is Dynamic Programming based C++ implementation.

```
// A Dynamic Programming based C++ program to count number of ways  
// to cover a distance with 1, 2 and 3 steps  
#include<iostream>  
using namespace std;  
  
int printCountDP(int dist)  
{  
    int count[dist+1];  
  
    // Initialize base values. There is one way to cover 0 and 1  
    // distances and two ways to cover 2 distance  
    count[0] = 1, count[1] = 1, count[2] = 2;  
  
    // Fill the count array in bottom up manner  
    for (int i=3; i<=dist; i++)  
        count[i] = count[i-1] + count[i-2] + count[i-3];  
  
    return count[dist];  
}  
  
// driver program  
int main()  
{  
    int dist = 4;  
    cout << printCountDP(dist);  
    return 0;  
}
```

Output:

7

This article is contributed by Vignesh Venkatesan. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Find the longest path in a matrix with given constraints

Given a $n \times n$ matrix where numbers all numbers are distinct and are distributed from range 1 to n^2 , find the maximum length path (starting from any cell) such that all cells along the path are increasing order with a difference of 1.

We can move in 4 directions from a given cell (i, j) , i.e., we can move to $(i+1, j)$ or $(i, j+1)$ or $(i-1, j)$ or $(i, j-1)$ with the condition that the adjacent cells are increasing by 1.

Example:

```
Input: mat[][] = {{1, 2, 9}
                  {5, 3, 8}
                  {4, 6, 7}}
```

Output: 4

The longest path is 6-7-8-9.

We strongly recommend you to minimize your browser and try this yourself first.

The idea is simple, we calculate longest path beginning with every cell. Once we have computed longest for all cells, we return maximum of all longest paths. One important observation in this approach is many overlapping subproblems. Therefore this problem can be optimally solved using Dynamic Programming.

Below is Dynamic Programming based C implementation that uses a lookup table $dp[i][j]$ to check if a problem is already solved or not.

```
#include<bits/stdc++.h>
#define n 3
using namespace std;

// Returns length of the longest path beginning with mat[i][j].
// This function mainly uses lookup table dp[n][n]
int findLongestFromACell(int i, int j, int mat[n][n], int dp[n][n])
{
    // Base case
    if (i<0 || i>=n || j<0 || j>=n)
        return 0;

    // If this subproblem is already solved
    if (dp[i][j] != -1)
        return dp[i][j];

    // Since all numbers are unique and in range from 1 to n*n,
    // there is atmost one possible direction from any cell
    if (j<n-1 && ((mat[i][j]) + 1 == mat[i][j+1]))
        return dp[i][j] = 1 + findLongestFromACell(i,j+1,mat,dp);

    if (j>0 && (mat[i][j] + 1 == mat[i][j-1]))
        return dp[i][j] = 1 + findLongestFromACell(i,j-1,mat,dp);

    if (i>0 && (mat[i][j] + 1 == mat[i-1][j]))
        return dp[i][j] = 1 + findLongestFromACell(i-1,j,mat,dp);

    if (i<n-1 && (mat[i][j] + 1 == mat[i+1][j]))
        return dp[i][j] = 1 + findLongestFromACell(i+1,j,mat,dp);

    // If none of the adjacent fours is one greater
    return dp[i][j] = 1;
}

// Returns length of the longest path beginning with any cell
int finLongestOverAll(int mat[n][n])
{
    int result = 1; // Initialize result
```

```

// Create a lookup table and fill all entries in it as -1
int dp[n][n];
memset(dp, -1, sizeof dp);

// Compute longest path beginning from all cells
for (int i=0; i<n; i++)
{
    for (int j=0; j<n; j++)
    {
        if (dp[i][j] == -1)
            findLongestFromACell(i, j, mat, dp);

        // Update result if needed
        result = max(result, dp[i][j]);
    }
}

return result;
}

// Driver program
int main()
{
    int mat[n][n] = {{1, 2, 9},
                     {5, 3, 8},
                     {4, 6, 7}};
    cout << "Length of the longest path is "
         << finLongestOverAll(mat);
    return 0;
}

```

Output:

Length of the longest path is 4

Time complexity of the above solution is $O(n^2)$. It may seem more at first look. If we take a closer look, we can notice that all values of $dp[i][j]$ are computed only once.

This article is contributed by [Ekta Goel](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Matrix
Dynamic Programming
Matrix

Dynamic Programming | Set 25 (Subset Sum Problem)

Given a set of non-negative integers, and a value sum , determine if there is a subset of the given set with sum equal to given sum .

Examples: $set[] = \{3, 34, 4, 12, 5, 2\}$, $sum = 9$
Output: True //There is a subset (4, 5) with sum 9.

We strongly recommend that you click here and practice it, before moving on to the solution.

Let $\text{isSubSetSum}(\text{int set[], int n, int sum})$ be the function to find whether there is a subset of set[] with sum equal to sum . n is the number of elements in set[] .

The isSubSetSum problem can be divided into two subproblems

- ...a) Include the last element, recur for $n = n-1$, $sum = sum - \text{set}[n-1]$
- ...b) Exclude the last element, recur for $n = n-1$.

If any of the above the above subproblems return true, then return true.

Following is the recursive formula for $\text{isSubSetSum}()$ problem.

```

isSubSetSum(set, n, sum) = isSubSetSum(set, n-1, sum) ||
                           isSubSetSum(set, n-1, sum-set[n-1])

```

Base Cases:

```

isSubSetSum(set, n, sum) = false, if sum > 0 and n == 0
isSubSetSum(set, n, sum) = true, if sum == 0

```

Following is naive recursive implementation that simply follows the recursive structure mentioned above.

C

```
// A recursive solution for subset sum problem
#include <stdio.h>

// Returns true if there is a subset of set[] with sum equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
    // Base Cases
    if (sum == 0)
        return true;
    if (n == 0 && sum != 0)
        return false;

    // If last element is greater than sum, then ignore it
    if (set[n-1] > sum)
        return isSubsetSum(set, n-1, sum);

    /* else, check if sum can be obtained by any of the following
       (a) including the last element
       (b) excluding the last element */
    return isSubsetSum(set, n-1, sum) ||
           isSubsetSum(set, n-1, sum-set[n-1]);
}

// Driver program to test above function
int main()
{
    int set[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(set)/sizeof(set[0]);
    if (isSubsetSum(set, n, sum) == true)
        printf("Found a subset with given sum");
    else
        printf("No subset with given sum");
    return 0;
}
```

Java

```
// A recursive solution for subset sum problem
class subset_sum
{
    // Returns true if there is a subset of set[] with sum
    // equal to given sum
    static boolean isSubsetSum(int set[], int n, int sum)
    {
        // Base Cases
        if (sum == 0)
            return true;
        if (n == 0 && sum != 0)
            return false;

        // If last element is greater than sum, then ignore it
        if (set[n-1] > sum)
            return isSubsetSum(set, n-1, sum);

        /* else, check if sum can be obtained by any of the following
           (a) including the last element
           (b) excluding the last element */
        return isSubsetSum(set, n-1, sum) ||
               isSubsetSum(set, n-1, sum-set[n-1]);
    }

    /* Driver program to test above function */
    public static void main (String args[])
    {
        int set[] = {3, 34, 4, 12, 5, 2};
        int sum = 9;
        int n = set.length;
        if (isSubsetSum(set, n, sum) == true)
```

```

        System.out.println("Found a subset with given sum");
    else
        System.out.println("No subset with given sum");
    }
/* This code is contributed by Rajat Mishra */

```

Output:

Found a subset with given sum

The above solution may try all subsets of given set in worst case. Therefore time complexity of the above solution is exponential. The problem is in-fact **NP-Complete** (There is no known polynomial time solution for this problem).

We can solve the problem in Pseudo-polynomial time using Dynamic programming. We create a boolean 2D table `subset[][]` and fill it in bottom up manner. The value of `subset[i][j]` will be true if there is a subset of `set[0..j-1]` with sum equal to `i`, otherwise false. Finally, we return `subset[sum][n]`

C

```

// A Dynamic Programming solution for subset sum problem
#include <stdio.h>

// Returns true if there is a subset of set[] with sun equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
    // The value of subset[i][j] will be true if there is a
    // subset of set[0..j-1] with sum equal to i
    bool subset[sum+1][n+1];

    // If sum is 0, then answer is true
    for (int i = 0; i <= n; i++)
        subset[0][i] = true;

    // If sum is not 0 and set is empty, then answer is false
    for (int i = 1; i <= sum; i++)
        subset[i][0] = false;

    // Fill the subset table in botton up manner
    for (int i = 1; i <= sum; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            subset[i][j] = subset[i][j-1];
            if (i >= set[j-1])
                subset[i][j] = subset[i][j] || subset[i - set[j-1]][j-1];
        }
    }

    /* // uncomment this code to print table
    for (int i = 0; i <= sum; i++)
    {
        for (int j = 0; j <= n; j++)
            printf ("%4d", subset[i][j]);
        printf("\n");
    }*/
}

return subset[sum][n];
}

// Driver program to test above function
int main()
{
    int set[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(set)/sizeof(set[0]);
    if (isSubsetSum(set, n, sum) == true)
        printf("Found a subset with given sum");
    else
        printf("No subset with given sum");
    return 0;
}

```

Java

```
// A Dynamic Programming solution for subset sum problem
class subset_sum
{
    // Returns true if there is a subset of set[] with sum equal to given sum
    static boolean isSubsetSum(int set[], int n, int sum)
    {
        // The value of subset[i][j] will be true if there
        // is a subset of set[0..j-1] with sum equal to i
        boolean subset[][] = new boolean[sum+1][n+1];

        // If sum is 0, then answer is true
        for (int i = 0; i <= n; i++)
            subset[0][i] = true;

        // If sum is not 0 and set is empty, then answer is false
        for (int i = 1; i <= sum; i++)
            subset[i][0] = false;

        // Fill the subset table in bottom up manner
        for (int i = 1; i <= sum; i++)
        {
            for (int j = 1; j <= n; j++)
            {
                subset[i][j] = subset[i][j-1];
                if (i >= set[j-1])
                    subset[i][j] = subset[i][j] ||
                        subset[i - set[j-1]][j-1];
            }
        }

        /* // uncomment this code to print table
        for (int i = 0; i <= sum; i++)
        {
            for (int j = 0; j <= n; j++)
                printf("%4d", subset[i][j]);
            printf("\n");
        }*/
    }

    return subset[sum][n];
}

/* Driver program to test above function */
public static void main (String args[])
{
    int set[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = set.length;
    if (isSubsetSum(set, n, sum) == true)
        System.out.println("Found a subset with given sum");
    else
        System.out.println("No subset with given sum");
}

/* This code is contributed by Rajat Mishra */
```

Output:

Found a subset with given sum

Time complexity of the above solution is O(sum*n).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Adobe-Question
Dynamic Programming

Problem statement: Consider a row of n coins of values $v_1 \dots v_n$, where n is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

Note: The opponent is as clever as the user.

Let us understand the problem with few examples:

1. 5, 3, 7, 10 : The user collects maximum value as $15(10 + 5)$

2. 8, 15, 3, 7 : The user collects maximum value as $22(7 + 15)$

Does choosing the best at each move give an optimal solution?

No. In the second example, this is how the game can finish:

1.

.....User chooses 8.

.....Opponent chooses 15.

.....User chooses 7.

.....Opponent chooses 3.

Total value collected by user is $15(8 + 7)$

2.

.....User chooses 7.

.....Opponent chooses 8.

.....User chooses 15.

.....Opponent chooses 3.

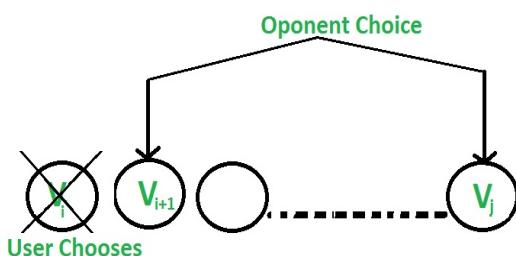
Total value collected by user is $22(7 + 15)$

So if the user follows the second game state, maximum value can be collected although the first move is not the best.

There are two choices:

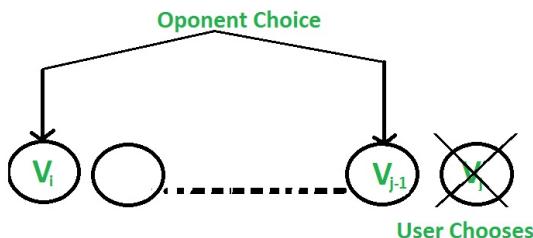
1. The user chooses the ith coin with value V_i : The opponent either chooses $(i+1)$ th coin or j th coin. The opponent intends to choose the coin which leaves the user with minimum value.

i.e. The user can collect the value $V_i + \min(F(i+2, j), F(i+1, j-1))$



2. The user chooses the j th coin with value V_j : The opponent either chooses i th coin or $(j-1)$ th coin. The opponent intends to choose the coin which leaves the user with minimum value.

i.e. The user can collect the value $V_j + \min(F(i+1, j-1), F(i, j-2))$



Following is recursive solution that is based on above two choices. We take the maximum of two choices.

$F(i, j)$ represents the maximum value the user can collect from i 'th coin to j 'th coin.

$$F(i, j) = \max(V_i + \min(F(i+2, j), F(i+1, j-1)), V_j + \min(F(i+1, j-1), F(i, j-2)))$$

Base Cases

$$F(i, j) = V_i \quad \text{If } j == i$$

$$F(i, j) = \max(V_i, V_j) \quad \text{If } j == i+1$$

Why Dynamic Programming?

The above relation exhibits overlapping sub-problems. In the above relation, $F(i+1, j-1)$ is calculated twice.

```
// C program to find out maximum value from a given sequence of coins
#include <stdio.h>
#include <limits.h>

// Utility functions to get maximum and minimum of two integers
int max(int a, int b) { return a > b ? a : b; }
int min(int a, int b) { return a < b ? a : b; }

// Returns optimal value possible that a player can collect from
// an array of coins of size n. Note that n must be even
int optimalStrategyOfGame(int* arr, int n)
{
    // Create a table to store solutions of subproblems
    int table[n][n], gap, i, j, x, y, z;

    // Fill table using above recursive formula. Note that the table
    // is filled in diagonal fashion (similar to http://goo.gl/PQqoS),
    // from diagonal elements to table[0][n-1] which is the result.
    for (gap = 0; gap < n; ++gap)
    {
        for (i = 0, j = gap; j < n; ++i, ++j)
        {
            // Here x is value of F(i+2, j), y is F(i+1, j-1) and
            // z is F(i, j-2) in above recursive formula
            x = ((i+2) <= j) ? table[i+2][j] : 0;
            y = ((i+1) <= (j-1)) ? table[i+1][j-1] : 0;
            z = (i <= (j-2)) ? table[i][j-2] : 0;

            table[i][j] = max(arr[i] + min(x, y), arr[j] + min(y, z));
        }
    }

    return table[0][n-1];
}

// Driver program to test above function
int main()
{
    int arr1[] = {8, 15, 3, 7};
    int n = sizeof(arr1)/sizeof(arr1[0]);
    printf("%d\n", optimalStrategyOfGame(arr1, n));

    int arr2[] = {2, 2, 2, 2};
    n = sizeof(arr2)/sizeof(arr2[0]);
    printf("%d\n", optimalStrategyOfGame(arr2, n));

    int arr3[] = {20, 30, 2, 2, 2, 10};
    n = sizeof(arr3)/sizeof(arr3[0]);
    printf("%d\n", optimalStrategyOfGame(arr3, n));

    return 0;
}
```

Output:

```
22
4
42
```

Exercise

Your thoughts on the strategy when the user wishes to only win instead of winning with the maximum value. Like above problem, number of coins is even.

Can Greedy approach work quite well and give an optimal solution? Will your answer change if number of coins is odd?

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming | Set 10 (0-1 Knapsack Problem)

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

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

1) Optimal Substructure:

To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.

Therefore, the maximum value that can be obtained from n items is max of following two values.

1) Maximum value obtained by n-1 items and W weight (excluding nth item).

2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

C/C++

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
    // Base Case
    if (n == 0 || W == 0)
        return 0;

    // If weight of the nth item is more than Knapsack capacity W, then
    // this item cannot be included in the optimal solution
    if (wt[n-1] > W)
        return knapSack(W, wt, val, n-1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
    else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                    knapSack(W, wt, val, n-1)
                );
}

// Driver program to test above function
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

Java

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
class Knapsack
{

    // A utility function that returns maximum of two integers
    static int max(int a, int b) { return (a > b)? a : b; }
```

```

// Returns the maximum value that can be put in a knapsack of capacity W
static int knapSack(int W, int wt[], int val[], int n)
{
    // Base Case
    if (n == 0 || W == 0)
        return 0;

    // If weight of the nth item is more than Knapsack capacity W, then
    // this item cannot be included in the optimal solution
    if (wt[n-1] > W)
        return knapSack(W, wt, val, n-1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
    else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                    knapSack(W, wt, val, n-1)
                );
}

// Driver program to test above function
public static void main(String args[])
{
    int val[] = new int[]{60, 100, 120};
    int wt[] = new int[]{10, 20, 30};
    int W = 50;
    int n = val.length;
    System.out.println(knapSack(W, wt, val, n));
}
}

/*This code is contributed by Rajat Mishra */

```

Python

```

#A naive recursive implementation of 0-1 Knapsack Problem

# Returns the maximum value that can be put in a knapsack of
# capacity W
def knapSack(W , wt , val , n):

    # Base Case
    if n == 0 or W == 0 :
        return 0

    # If weight of the nth item is more than Knapsack of capacity
    # W, then this item cannot be included in the optimal solution
    if (wt[n-1] > W):
        return knapSack(W , wt , val , n-1)

    # return the maximum of two cases:
    # (1) nth item included
    # (2) not included
    else:
        return max(val[n-1] + knapSack(W-wt[n-1] , wt , val , n-1),
                  knapSack(W , wt , val , n-1))

# end of function knapSack

# To test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print knapSack(W , wt , val , n)

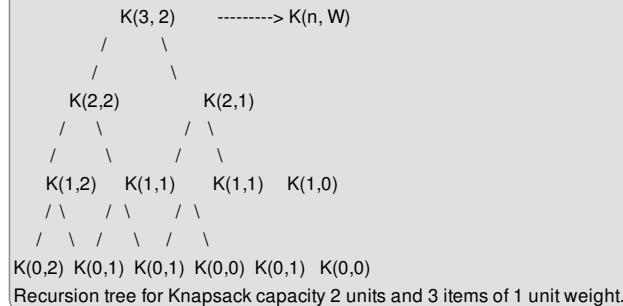
# This code is contributed by Nikhil Kumar Singh

```

Output:

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).

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



Since subproblems are evaluated again, this problem has Overlapping Subproblems property. So the 0-1 Knapsack problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical [Dynamic Programming\(DP\) problems](#), recomputations of same subproblems can be avoided by constructing a temporary array $K[]$ in bottom up manner. Following is Dynamic Programming based implementation.

C++

```

// A Dynamic Programming based solution for 0-1 Knapsack problem
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n+1][W+1];

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++)
    {
        for (w = 0; w <= W; w++)
        {
            if (i==0 || w==0)
                K[i][w] = 0;
            else if (wt[i-1] <= w)
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
            else
                K[i][w] = K[i-1][w];
        }
    }

    return K[n][W];
}

int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}

```

Java

```

// A Dynamic Programming based solution for 0-1 Knapsack problem
class Knapsack
{

```

```

// A utility function that returns maximum of two integers
static int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W
static int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[][] = new int[n+1][W+1];

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++)
    {
        for (w = 0; w <= W; w++)
        {
            if (i==0 || w==0)
                K[i][w] = 0;
            else if (wt[i-1] <= w)
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
            else
                K[i][w] = K[i-1][w];
        }
    }

    return K[n][W];
}

// Driver program to test above function
public static void main(String args[])
{
    int val[] = new int[]{60, 100, 120};
    int wt[] = new int[]{10, 20, 30};
    int W = 50;
    int n = val.length;
    System.out.println(knapSack(W, wt, val, n));
}

```

/*This code is contributed by Rajat Mishra */

Python

```

# A Dynamic Programming based Python Program for 0-1 Knapsack problem
# Returns the maximum value that can be put in a knapsack of capacity W
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]

    # Build table K[][] in bottom up manner
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    return K[n][W]

# Driver program to test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))

# This code is contributed by Bhavya Jain

```

Output:

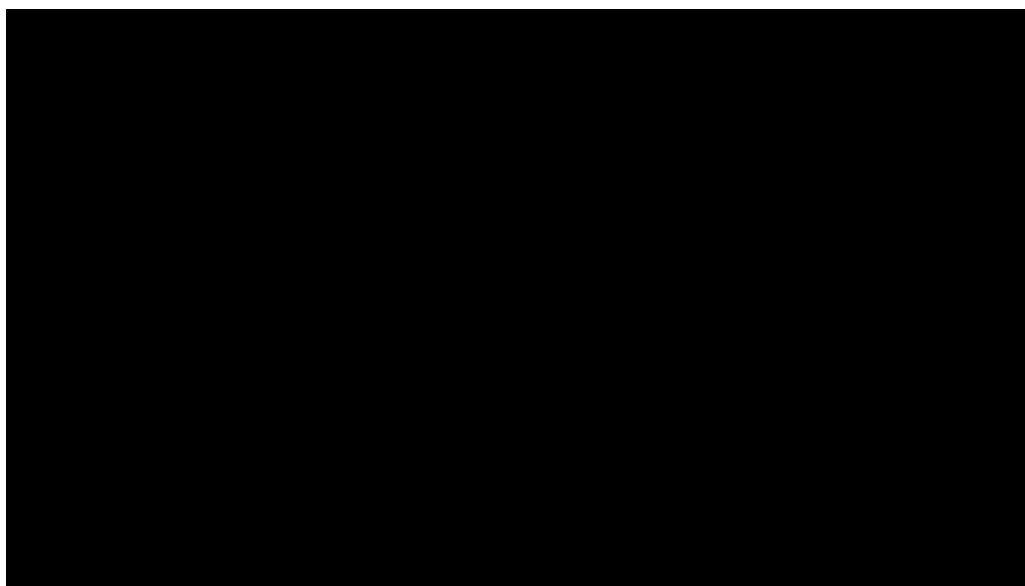
220

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

References:

<http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf>

<http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf>



Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
knapsack

Dynamic Programming | Set 37 (Boolean Parenthesization Problem)

Given a boolean expression with following symbols.

Symbols

'T' ---> true
'F' ---> false

And following operators filled between symbols

Operators

& ---> boolean AND
| ---> boolean OR
^ ---> boolean XOR

Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.

Let the input be in form of two arrays one contains the symbols (T and F) in order and other contains operators (&, | and ^)

Examples:

Input: symbol[] = {T, F, T}
operator[] = {^, &}

Output: 2

The given expression is "T ^ F & T", it evaluates true
in two ways "((T ^ F) & T)" and "(T ^ (F & T))"

Input: symbol[] = {T, F, F}
operator[] = {^, |}

Output: 2

The given expression is "T ^ F | F", it evaluates true
in two ways "(T ^ F) | F" and "(T ^ (F | F))".

Input: symbol[] = {T, T, F, T}
operator[] = {|, &, ^}

Output: 4

The given expression is "T | T & F ^ T", it evaluates true
in 4 ways ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T)
and (T|((T&F)^T)).

Solution:

Let $T(i, j)$ represents the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to true.

$$T(i, j) = \sum_{k=i}^{j-1} \begin{cases} T(i, k) * T(k+1, j) & \text{If operator}[k] \text{ is } \& \\ Total(i, k) * Total(k+1, j) - F(i, k) * F(k+1, j) & \text{If operator}[k] \text{ is } | \\ T(i, k) * F(k+1, j) + F(i, k) * T(k+1) & \text{If operator}[k] \text{ is } ^\wedge \end{cases}$$

$$\text{Total}(i, j) = T(i, j) + F(i, j)$$

Let $F(i, j)$ represents the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to false.

$$F(i, j) = \sum_{k=i}^{j-1} \begin{cases} Total(i, k) * Total(k+1, j) - T(i, k) * T(k+1, j) & \text{If operator}[k] \text{ is } \& \\ F(i, k) * F(k+1, j) & \text{If operator}[k] \text{ is } | \\ T(i, k) * T(k+1, j) + F(i, k) * F(k+1) & \text{If operator}[k] \text{ is } ^\wedge \end{cases}$$

$$\text{Total}(i, j) = T(i, j) + F(i, j)$$

Base Cases:

$T(i, i) = 1$ if $\text{symbol}[i] = \text{T}$
$T(i, i) = 0$ if $\text{symbol}[i] = \text{F}$
$F(i, i) = 1$ if $\text{symbol}[i] = \text{F}$
$F(i, i) = 0$ if $\text{symbol}[i] = \text{T}$

If we draw recursion tree of above recursive solution, we can observe that it many overlapping subproblems. Like other dynamic programming problems, it can be solved by filling a table in bottom up manner. Following is C++ implementation of dynamic programming solution.

```
#include<iostream>
#include<cstring>
using namespace std;

// Returns count of all possible parenthesizations that lead to
// result true for a boolean expression with symbols like true
// and false and operators like &, | and ^ filled between symbols
int countParenth(char symb[], char oper[], int n)
{
    int F[n][n], T[n][n];

    // Fill diagonal entries first
    // All diagonal entries in T[i][i] are 1 if symbol[i]
    // is T (true). Similarly, all F[i][i] entries are 1 if
    // symbol[i] is F (False)
    for (int i = 0; i < n; i++)
    {
        F[i][i] = (symb[i] == 'F')? 1: 0;
        T[i][i] = (symb[i] == 'T')? 1: 0;
    }

    // Now fill T[i][i+1], T[i][i+2], T[i][i+3]... in order
    // And F[i][i+1], F[i][i+2], F[i][i+3]... in order
    for (int gap=1; gap<n; ++gap)
    {
        for (int i=0, j=gap; j<n; ++i, ++j)
        {
            T[i][j] = F[i][j] = 0;
            for (int g=0; g<gap; g++)
            {
                // Find place of parenthesization using current value
                // of gap
                int k = i + g;

                // Store Total[i][k] and Total[k+1][j]
                int tk = T[i][k] + F[i][k];
                int tkj = T[k+1][j] + F[k+1][j];

                // Follow the recursive formulas according to the current
                // operator
                if (oper[k] == '&')
                {
                    T[i][j] += T[i][k]*T[k+1][j];
                    F[i][j] += (tk*tkj - T[i][k]*T[k+1][j]);
                }
                if (oper[k] == '|')
                {
                    T[i][j] += F[i][k]*T[k+1][j];
                    F[i][j] += (tk*tkj - F[i][k]*T[k+1][j]);
                }
            }
        }
    }
}
```

```

    {
        F[i][j] += F[i][k]*F[k+1][j];
        T[i][j] += (tk*tkj - F[i][k]*F[k+1][j]);
    }
    if (oper[k] == '^')
    {
        T[i][j] += F[i][k]*T[k+1][j] + T[i][k]*F[k+1][j];
        F[i][j] += T[i][k]*T[k+1][j] + F[i][k]*F[k+1][j];
    }
}
}
return T[0][n-1];
}

// Driver program to test above function
int main()
{
    char symbols[] = "TTFT";
    char operators[] = "|&^";
    int n = strlen(symbols);

    // There are 4 ways
    // ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T) and (T|((T&F)^T))
    cout << countParenth(symbols, operators, n);
    return 0;
}

```

Output:

4

Time Complexity: $O(n^3)$

Auxiliary Space: $O(n^2)$

References:

http://people.cs.clemson.edu/~bcdean/dp_practice/dp_9.swf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming

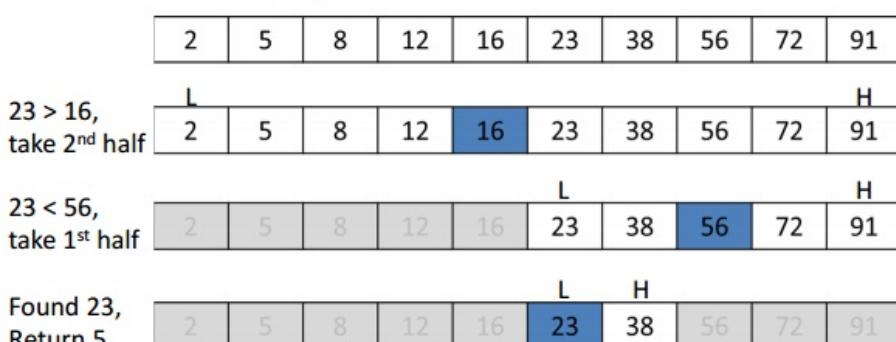
Binary Search

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do **linear search**. The time complexity of above algorithm is $O(n)$. Another approach to perform the same task is using Binary Search.

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

If searching for 23 in the 10-element array:



Example:

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Logn).

We strongly recommend that you click here and practice it, before moving on to the solution.

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

Recursive implementation of Binary Search

C/C++

```
#include <stdio.h>

// A recursive binary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l)/2;

        // If the element is present at the middle itself
        if (arr[mid] == x) return mid;

        // If element is smaller than mid, then it can only be present
        // in left subarray
        if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid+1, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}

int main(void)
{
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr)/ sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n-1, x);
    (result == -1)? printf("Element is not present in array")
                  : printf("Element is present at index %d", result);
    return 0;
}
```

Python

```
# Python Program for recursive binary search.

# Returns index of x in arr if present, else -1
def binarySearch (arr, l, r, x):

    # Check base case
    if r >= l:

        mid = l + (r - l)/2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid
```

```

# If element is smaller than mid, then it can only
# be present in left subarray
elif arr[mid] > x:
    return binarySearch(arr, l, mid-1, x)

# Else the element can only be present in right subarray
else:
    return binarySearch(arr, mid+1, r, x)

else:
    # Element is not present in the array
    return -1

# Test array
arr = [2, 3, 4, 10, 40]
x = 10

# Function call
result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
    print "Element is present at index %d" % result
else:
    print "Element is not present in array"

```

Java

```

// Java implementation of recursive Binary Search
class BinarySearch
{
    // Returns index of x if it is present in arr[l..r], else
    // return -1
    int binarySearch(int arr[], int l, int r, int x)
    {
        if (r >= l)
        {
            int mid = l + (r - l)/2;

            // If the element is present at the middle itself
            if (arr[mid] == x)
                return mid;

            // If element is smaller than mid, then it can only
            // be present in left subarray
            if (arr[mid] > x)
                return binarySearch(arr, l, mid-1, x);

            // Else the element can only be present in right
            // subarray
            return binarySearch(arr, mid+1, r, x);
        }

        // We reach here when element is not present in array
        return -1;
    }

    // Driver method to test above
    public static void main(String args[])
    {
        BinarySearch ob = new BinarySearch();
        int arr[] = {2,3,4,10,40};
        int n = arr.length;
        int x = 10;
        int result = ob.binarySearch(arr,0,n-1,x);
        if (result == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at index "+result);
    }
}
/* This code is contributed by Rajat Mishra */

```

Output:

Element is present at index 3

Iterative implementation of Binary Search

C/C++

```
#include <stdio.h>

// A iterative binary search function. It returns location of x in
// given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        int m = l + (r-l)/2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // If we reach here, then element was not present
    return -1;
}

int main(void)
{
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr)/ sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n-1, x);
    (result == -1)? printf("Element is not present in array")
                  : printf("Element is present at index %d", result);
    return 0;
}
```

Python

```
# Iterative Binary Search Function
# It returns location of x in given array arr if present,
# else returns -1
def binarySearch(arr, l, r, x):

    while l <= r:

        mid = l + (r - l)/2;

        # Check if x is present at mid
        if arr[mid] == x:
            return mid

        # If x is greater, ignore left half
        elif arr[mid] < x:
            l = mid + 1

        # If x is smaller, ignore right half
        else:
            r = mid - 1

    # If we reach here, then the element was not present
    return -1
```

```

# Test array
arr = [2, 3, 4, 10, 40]
x = 10

# Function call
result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
    print "Element is present at index %d" % result
else:
    print "Element is not present in array"

```

Java

```

// Java implementation of iterative Binary Search
class BinarySearch
{
    // Returns index of x if it is present in arr[], else
    // return -1
    int binarySearch(int arr[], int x)
    {
        int l = 0, r = arr.length - 1;
        while (l <= r)
        {
            int m = l + (r-l)/2;

            // Check if x is present at mid
            if (arr[m] == x)
                return m;

            // If x greater, ignore left half
            if (arr[m] < x)
                l = m + 1;

            // If x is smaller, ignore right half
            else
                r = m - 1;
        }

        // if we reach here, then element was not present
        return -1;
    }

    // Driver method to test above
    public static void main(String args[])
    {
        BinarySearch ob = new BinarySearch();
        int arr[] = {2, 3, 4, 10, 40};
        int n = arr.length;
        int x = 10;
        int result = ob.binarySearch(arr, x);
        if (result == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at index "+result);
    }
}

```

Output:

Element is present at index 3

Time Complexity:

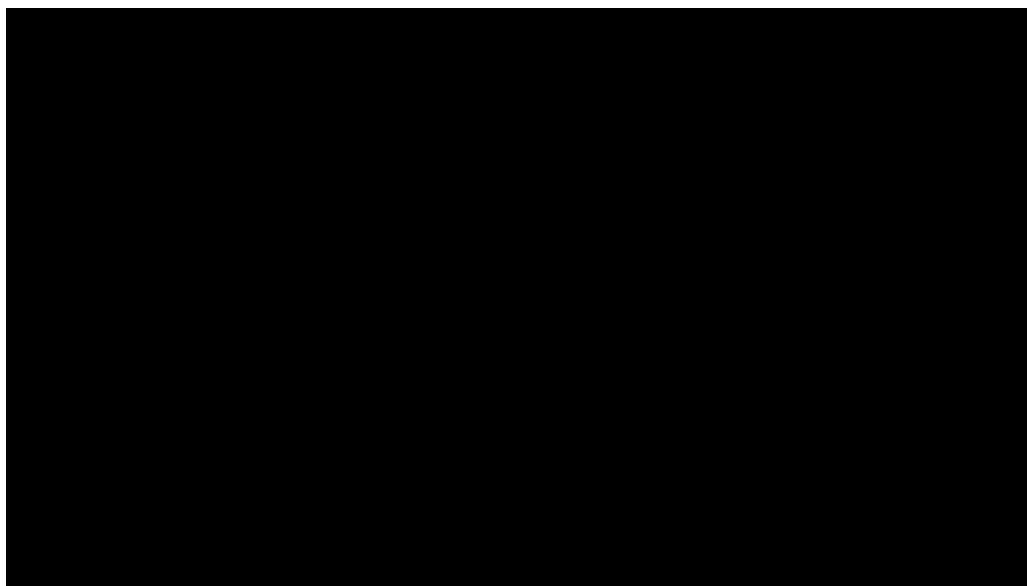
The time complexity of Binary Search can be written as

$$T(n) = T(n/2) + c$$

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\Theta(\log n)$.

Auxiliary Space: O(1) in case of iterative implementation. In case of recursive implementation, O(log n) recursion call stack space.

Algorithmic Paradigm: Divide and Conquer



Interesting articles based on Binary Search.

- The Ubiquitous Binary Search
- Interpolation search vs Binary search
- Find the minimum element in a sorted and rotated array
- Find a peak element
- Find a Fixed Point in a given array
- Count the number of occurrences in a sorted array
- Median of two sorted arrays
- Floor and Ceiling in a sorted array
- Find the maximum element in an array which is first increasing and then decreasing

Coding Practice Questions on Binary Search

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

Search an element in a sorted and rotated array

An element in a sorted array can be found in $O(\log n)$ time via binary search. But suppose we rotate an ascending order sorted array at some pivot unknown to you beforehand. So for instance, 1 2 3 4 5 might become 3 4 5 1 2. Devise a way to find an element in the rotated array in $O(\log n)$ time.

3	4	5	1	2
---	---	---	---	---

Input : arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};

key = 3

Output : Found at index 8

Input : arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};

key = 30

```

Output : Not found

Input : arr[] = {30, 40, 50, 10, 20}
        key = 10
Output : Found at index 3

```

We strongly recommend that you click here and practice it, before moving on to the solution.

All solutions provided here assume that all elements in array are distinct.

The idea is to find the pivot point, divide the array in two sub-arrays and call binary search.

The main idea for finding pivot is – for a sorted (in increasing order) and pivoted array, pivot element is the only element for which next element to it is smaller than it.

Using above criteria and binary search methodology we can get pivot element in O(logn) time

```

Input arr[] = {3, 4, 5, 1, 2}
Element to Search = 1
1) Find out pivot point and divide the array in two
   sub-arrays. (pivot = 2) /*Index of 5*/
2) Now call binary search for one of the two sub-arrays.
   (a) If element is greater than 0th element then
       search in left array
   (b) Else Search in right array
       (1 will go in else as 1 If element is found in selected sub-array then return index
Else return -1.

```

Implementation:

```

/* Program to search an element in a sorted and pivoted array*/
#include <stdio.h>

int findPivot(int[], int, int);
int binarySearch(int[], int, int, int);

/* Searches an element key in a pivoted sorted array arr[]
   of size n */
int pivotedBinarySearch(int arr[], int n, int key)
{
    int pivot = findPivot(arr, 0, n-1);

    // If we didn't find a pivot, then array is not rotated at all
    if (pivot == -1)
        return binarySearch(arr, 0, n-1, key);

    // If we found a pivot, then first compare with pivot and then
    // search in two subarrays around pivot
    if (arr[pivot] == key)
        return pivot;
    if (arr[0] <= key)
        return binarySearch(arr, 0, pivot-1, key);
    return binarySearch(arr, pivot+1, n-1, key);
}

/* Function to get pivot. For array 3, 4, 5, 6, 1, 2 it returns
   3 (index of 6) */
int findPivot(int arr[], int low, int high)
{
    // base cases
    if (high < low) return -1;
    if (high == low) return low;

    int mid = (low + high)/2; /*low + (high - low)/2;*/
    if (mid < high && arr[mid] > arr[mid + 1])
        return mid;
    if (mid > low && arr[mid] < arr[mid - 1])
        return (mid-1);
    if (arr[low] >= arr[mid])
        return findPivot(arr, low, mid-1);
    return findPivot(arr, mid + 1, high);
}

```

```

/* Standard Binary Search function*/
int binarySearch(int arr[], int low, int high, int key)
{
    if (high < low)
        return -1;
    int mid = (low + high)/2; /*low + (high - low)/2;*/
    if (key == arr[mid])
        return mid;
    if (key > arr[mid])
        return binarySearch(arr, (mid + 1), high, key);
    return binarySearch(arr, low, (mid -1), key);
}

/* Driver program to check above functions */
int main()
{
    // Let us search 3 in below array
    int arr1[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};
    int n = sizeof(arr1)/sizeof(arr1[0]);
    int key = 3;
    printf("Index: %d\n", pivotedBinarySearch(arr1, n, key));
    return 0;
}

```

Output:

Index of the element is 8

Time Complexity O(logn). Thanks to Ajay Mishra for initial solution.

Improved Solution:

We can search an element in one pass of Binary Search. The idea is to search

- 1) Find middle point $mid = (l + h)/2$
- 2) If key is present at middle point, return mid.
- 3) Else If $arr[l..mid]$ is sorted
 - a) If key to be searched lies in range from $arr[l]$ to $arr[mid]$, recur for $arr[l..mid]$.
 - b) Else recur for $arr[mid+1..r]$.
- 4) Else ($arr[mid+1..r]$ must be sorted)
 - a) If key to be searched lies in range from $arr[mid+1]$ to $arr[r]$, recur for $arr[mid+1..r]$.
 - b) Else recur for $arr[l..mid]$

Below is C++ implementation of above idea.

```

// Search an element in sorted and rotated array using
// single pass of Binary Search
#include<bits/stdc++.h>
using namespace std;

// Returns index of key in arr[l..h] if key is present,
// otherwise returns -1
int search(int arr[], int l, int h, int key)
{
    if (l > h) return -1;

    int mid = (l+h)/2;
    if (arr[mid] == key) return mid;

    /* If arr[l...mid] is sorted */
    if (arr[l] <= arr[mid])
    {
        /* As this subarray is sorted, we can quickly
         * check if key lies in half or other half */
        if (key >= arr[l] && key <= arr[mid])
            return search(arr, l, mid-1, key);

        return search(arr, mid+1, h, key);
    }

    /* If arr[l..mid] is not sorted, then arr[mid...r]
     * must be sorted*/
    if (key >= arr[mid] && key <= arr[h])
        return search(arr, mid+1, h, key);
}

```

```

    return search(arr, l, mid-1, key);
}

// Driver program
int main()
{
    int arr[] = {4, 5, 6, 7, 8, 9, 1, 2, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    int key = 6;
    int i = search(arr, 0, n-1, key);
    if (i != -1) cout << "Index: " << i << endl;
    else cout << "Key not found\n";
}

```

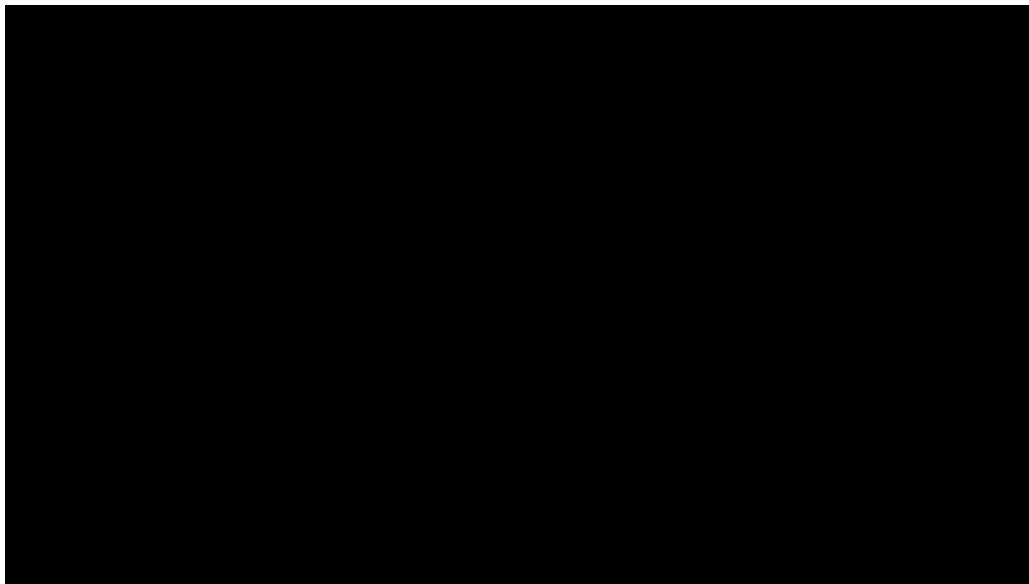
Output:

Index: 2

Thanks to [Gaurav Ahirwar](#) for suggesting above solution.

How to handle duplicates?

It doesn't look possible to search in $O(\log n)$ time in all cases when duplicates are allowed. For example consider searching 0 in {2, 2, 2, 2, 2, 2, 2, 0, 2} and {2, 0, 2, 2, 2, 2, 2, 2, 2, 2}. It doesn't look possible to decide whether to recur for left half or right half by doing constant number of comparisons at the middle.



Similar Articles:

[Find the minimum element in a sorted and rotated array](#)

[Given a sorted and rotated array, find if there is a pair with a given sum.](#)

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Searching
array
Binary-Search
rotation

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:

(5 1 4 2 8) \rightarrow (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since $5 > 1$.

(1 5 4 2 8) \rightarrow (1 4 5 2 8), Swap since $5 > 4$

(1 4 5 2 8) \rightarrow (1 4 2 5 8), Swap since $5 > 2$

(1 4 2 5 8) \rightarrow (1 4 2 5 8), Now, since these elements are already in order ($8 > 5$), algorithm does not swap them.

Second Pass:

```
(1 4 2 5 8) -> (1 4 2 5 8)
(1 4 2 5 8) -> (1 2 4 5 8), Swap since 4 > 2
(1 2 4 5 8) -> (1 2 4 5 8)
(1 2 4 5 8) -> (1 2 4 5 8)
```

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

```
(1 2 4 5 8) -> (1 2 4 5 8)
(1 2 4 5 8) -> (1 2 4 5 8)
(1 2 4 5 8) -> (1 2 4 5 8)
(1 2 4 5 8) -> (1 2 4 5 8)
```

Following are C/C++, Python and Java implementations of Bubble Sort.

C/C++

```
// C program for implementation of Bubble sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)
        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    for (int i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Java

```
// Java program for implementation of Bubble Sort
class BubbleSort
{
    void bubbleSort(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n-1; i++)
            for (int j = 0; j < n-i-1; j++)
                if (arr[j] > arr[j+1])
                    {
```

```

        // swap temp and arr[i]
        int temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
    }

}

/* Prints the array */
void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i] + " ");
    System.out.println();
}

// Driver method to test above
public static void main(String args[])
{
    BubbleSort ob = new BubbleSort();
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    ob.bubbleSort(arr);
    System.out.println("Sorted array");
    ob.printArray(arr);
}
}

/* This code is contributed by Rajat Mishra */

```

Python

```

# Python program for implementation of Bubble Sort

def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

    # Driver code to test above
    arr = [64, 34, 25, 12, 22, 11, 90]

    bubbleSort(arr)

    print ("Sorted array is:")
    for i in range(len(arr)):
        print ("%d" %arr[i]),

```

Output:

```

Sorted array:
11 12 22 25 34 64 90

```

Illustration :

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i=1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i=2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i=3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i=4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i=5	0	1	2	3	4				
	1	1	2	3					
i=6	0	1	2	3					
	1	2							

Optimized Implementation:

The above function always runs $O(n^2)$ time even if the array is sorted. It can be optimized by stopping the algorithm if inner loop didn't cause any swap.

```
// Optimized implementation of Bubble sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n-1; i++)
    {
        swapped = false;
        for (j = 0; j < n-i-1; j++)
        {
            if (arr[j] > arr[j+1])
            {
                swap(&arr[j], &arr[j+1]);
                swapped = true;
            }
        }
    }

    // IF no two elements were swapped by inner loop, then break
    if (swapped == false)
        break;
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
```

```
{
int arr[] = {64, 34, 25, 12, 22, 11, 90};
int n = sizeof(arr)/sizeof(arr[0]);
bubbleSort(arr, n);
printf("Sorted array: \n");
printArray(arr, n);
return 0;
}
```

Output:

Sorted array:
11 12 22 25 34 64 90

Worst and Average Case Time Complexity: $O(n^2)$. Worst case occurs when array is reverse sorted.

Best Case Time Complexity: $O(n)$. Best case occurs when array is already sorted.

Auxiliary Space: $O(1)$

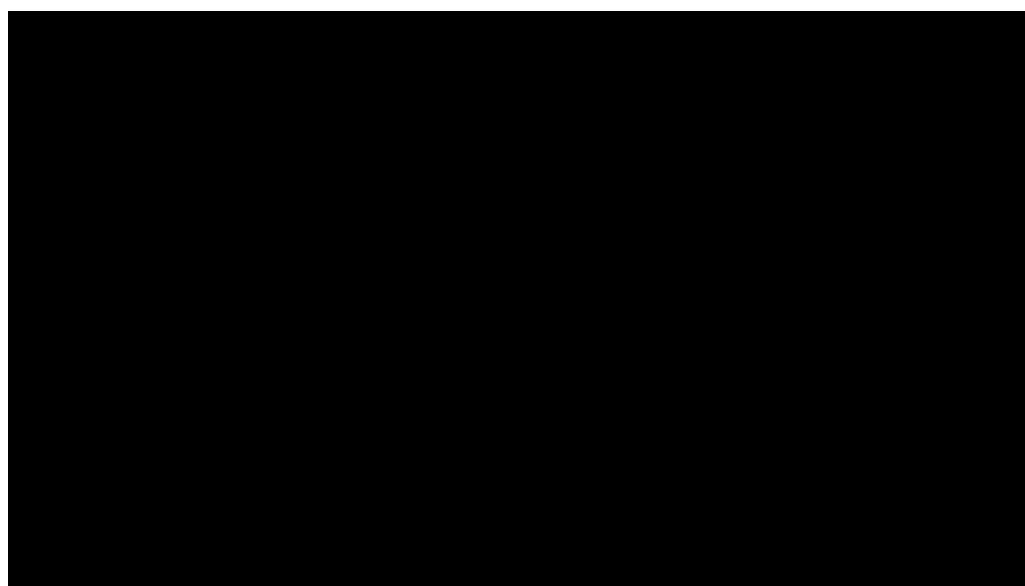
Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Sorting In Place: Yes

Stable: Yes

Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm.

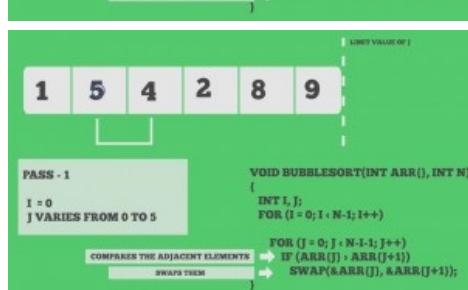
In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity ($2n$). For example, it is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to x axis) and with incrementing y their order changes (two elements are swapped) only at intersections of two lines (Source: [Wikipedia](#))



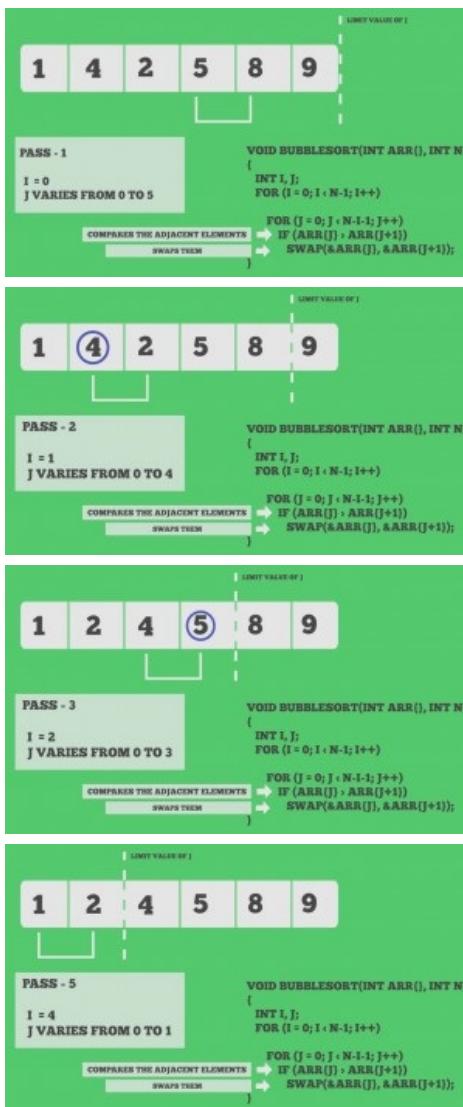
Snapshots:



```
VOID BUBBLESORT(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}
```



```
VOID BUBBLESORT(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}
```



Quiz on Bubble Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

- Selection Sort
- Insertion Sort
- Merge Sort
- Heap Sort
- QuickSort
- Radix Sort
- Counting Sort
- Bucket Sort
- ShellSort

Recursive Bubble Sort

Coding practice for sorting.

Reference:

- Wikipedia – Bubble Sort
- Image Source

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Insertion Sort

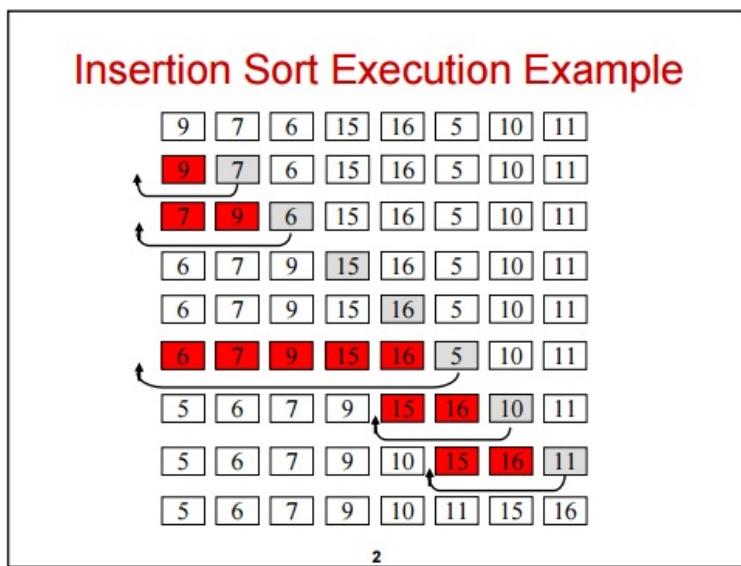
Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.



Algorithm

```
// Sort an arr[] of size n
insertionSort(arr, n)
Loop from i = 1 to n-1.
.....a) Pick element arr[i] and insert it into sorted sequence arr[0...i-1]
```

Example:



Another Example:

12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 5 (Size of input array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..i-1] are smaller than 13

11, 12, 13, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12, 13

We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

C/C++

```
// C program for insertion sort
#include <stdio.h>
#include <math.h>

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i-1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
{
    int i;
    for (i=0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

/* Driver program to test insertion sort*/
int main()
{
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

Python

```
# Python program for implementation of Insertion Sort

# Function to do insertion sort
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < arr[j] :
            arr[j+1] = arr[j]
            j -= 1
```

```

arr[j+1] = key

# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print ("Sorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i])

# This code is contributed by Mohit Kumra

```

Java

```

// Java program for implementation of Insertion Sort
class InsertionSort
{
    /*Function to sort array using insertion sort*/
    void sort(int arr[])
    {
        int n = arr.length;
        for (int i=1; i<n; ++i)
        {
            int key = arr[i];
            int j = i-1;

            /* Move elements of arr[0..i-1], that are
               greater than key, to one position ahead
               of their current position */
            while (j>=0 && arr[j] > key)
            {
                arr[j+1] = arr[j];
                j = j-1;
            }
            arr[j+1] = key;
        }
    }

    /* A utility function to print array of size n*/
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");

        System.out.println();
    }

    // Driver method
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6};

        InsertionSort ob = new InsertionSort();
        ob.sort(arr);

        printArray(arr);
    }
} /* This code is contributed by Rajat Mishra. */

```

Output:

```
5 6 11 12 13
```

Time Complexity: $O(n^2)$

Auxiliary Space: $O(1)$

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

Online: Yes

Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

What is Binary Insertion Sort?

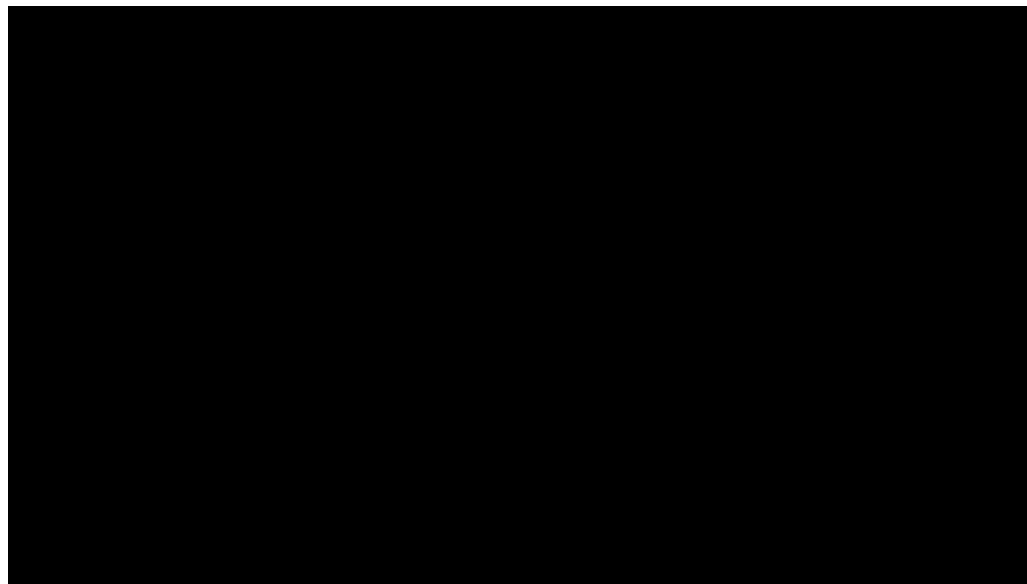
We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort find use binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sort it takes $O(i)$ (at i th iteration) in worst case. we can reduce it to $O(\log i)$ by using binary search. The algorithm as a whole still has a running worst case running time of $O(n^2)$ because of the series of swaps required for each insertion. Refer [this](#) for implementation.

How to implement Insertion Sort for Linked List?

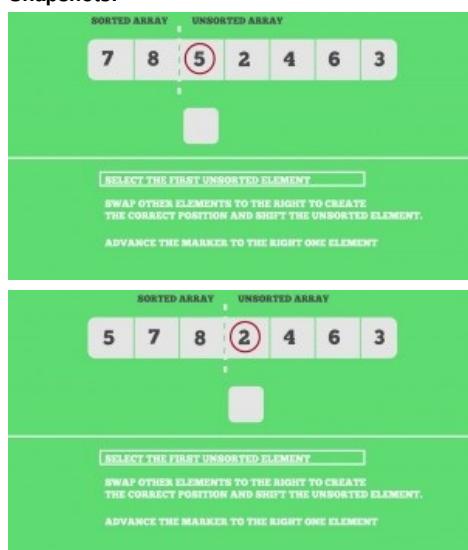
Below is simple insertion sort algorithm for linked list.

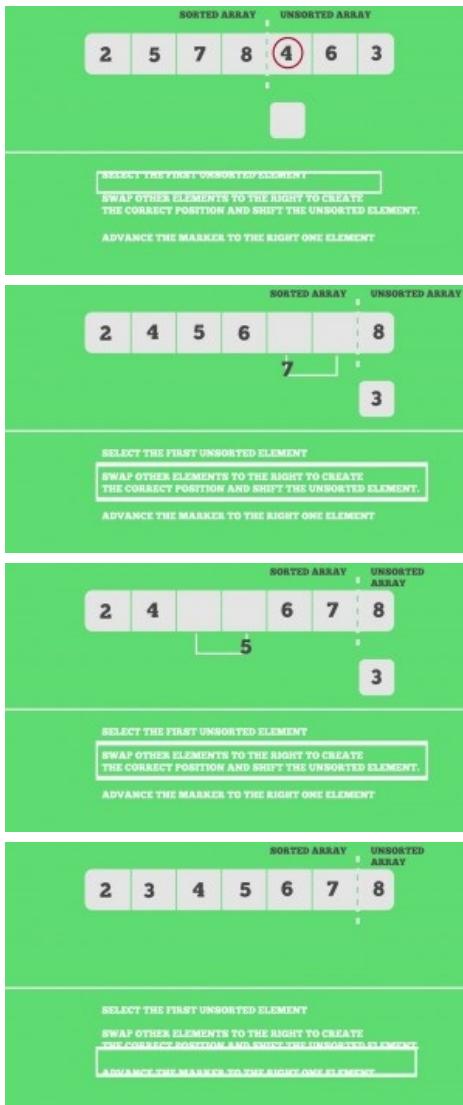
- 1) Create an empty sorted (or result) list
- 2) Traverse the given list, do following for every node.
 -a) Insert current node in sorted way in sorted or result list.
- 3) Change head of given linked list to head of sorted (or result) list.

Refer [this](#) for implementation.



Snapshots:





Quiz on Insertion Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz

Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort,

Coding practice for sorting.

Image Source: http://www.just.edu.jo/~basel/algorithms/Algo%20Slides/algo_ch2_getting_started.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, **GATE Corner** for GATE CS Preparation and **Quiz Corner** for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

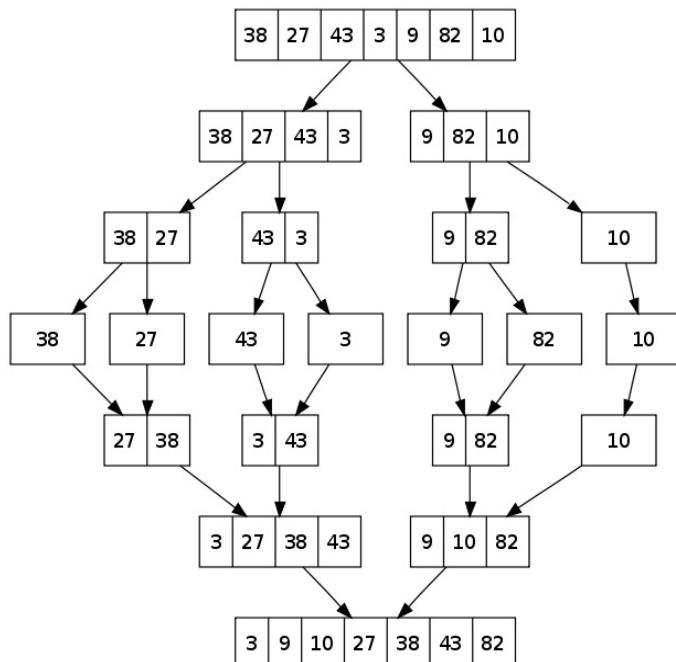
Merge Sort

Like **QuickSort**, **Merge Sort** is a **Divide and Conquer** algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The `merge(arr, l, m, r)` is key process that assumes that

`arr[l..m]` and `arr[m+1..r]` are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

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

The following diagram from [wikipedia](#) shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



We strongly recommend that you [click here and practice it, before moving on to the solution.](#)

C/C++

```
/* C program for Merge Sort */
#include<stdlib.h>
#include<stdio.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    /* Create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[]
     */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]
     */
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
        if (L[i] < R[j])
            arr[k] = L[i]
            i++;
        else
            arr[k] = R[j]
            j++;

    /* Copy remaining elements of L[], if any */
    while (i < n1)
        arr[k] = L[i]
        i++;
    /* Copy remaining elements of R[], if any */
    while (j < n2)
        arr[k] = R[j]
        j++;

    /* Print merged array */
    for (i = l; i <= r; i++)
        printf("%d ", arr[i]);
}
```

```

while (i < n1 && j < n2)
{
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}

/* Copy the remaining elements of L[], if there
   are any */
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}

/* Copy the remaining elements of R[], if there
   are any */
while (j < n2)
{
    arr[k] = R[j];
    j++;
    k++;
}

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

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}

/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
}

```

```
    return 0;  
}
```

Java

```
/* Java program for Merge Sort */  
class MergeSort  
{  
    // Merges two subarrays of arr[].  
    // First subarray is arr[l..m]  
    // Second subarray is arr[m+1..r]  
    void merge(int arr[], int l, int m, int r)  
    {  
        // Find sizes of two subarrays to be merged  
        int n1 = m - l + 1;  
        int n2 = r - m;  
  
        /* Create temp arrays */  
        int L[] = new int [n1];  
        int R[] = new int [n2];  
  
        /*Copy data to temp arrays*/  
        for (int i=0; i<n1; ++i)  
            L[i] = arr[l + i];  
        for (int j=0; j<n2; ++j)  
            R[j] = arr[m + 1+ j];  
  
        /* Merge the temp arrays */  
  
        // Initial indexes of first and second subarrays  
        int i = 0, j = 0;  
  
        // Initial index of merged subarry array  
        int k = l;  
        while (i < n1 && j < n2)  
        {  
            if (L[i] <= R[j])  
            {  
                arr[k] = L[i];  
                i++;  
            }  
            else  
            {  
                arr[k] = R[j];  
                j++;  
            }  
            k++;  
        }  
  
        /* Copy remaining elements of L[] if any */  
        while (i < n1)  
        {  
            arr[k] = L[i];  
            i++;  
            k++;  
        }  
  
        /* Copy remaining elements of R[] if any */  
        while (j < n2)  
        {  
            arr[k] = R[j];  
            j++;  
            k++;  
        }  
    }  
  
    // Main function that sorts arr[l..r] using  
    // merge()  
    void sort(int arr[], int l, int r)  
    {  
        if (l < r)  
        {
```

```

// Find the middle point
int m = (l+r)/2;

// Sort first and second halves
sort(arr, l, m);
sort(arr , m+1, r);

// Merge the sorted halves
merge(arr, l, m, r);
}

}

/* A utility function to print array of size n */
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i] + " ");
    System.out.println();
}

// Driver method
public static void main(String args[])
{
    int arr[] = {12, 11, 13, 5, 6, 7};

    System.out.println("Given Array");
    printArray(arr);

    MergeSort ob = new MergeSort();
    ob.sort(arr, 0, arr.length-1);

    System.out.println("\nSorted array");
    printArray(arr);
}
}

/* This code is contributed by Rajat Mishra */

```

Python

```

# Python program for implementation of MergeSort

# Merges two subarrays of arr[].
# First subarray is arr[l..m]
# Second subarray is arr[m+1..r]
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
    for i in range(0 , n1):
        L[i] = arr[l + i]

    for j in range(0 , n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0      # Initial index of first subarray
    j = 0      # Initial index of second subarray
    k = l      # Initial index of merged subarray

    while i < n1 and j < n2 :
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

```

```

# Copy the remaining elements of L[], if there
# are any
while i < n1:
    arr[k] = L[i]
    i += 1
    k += 1

# Copy the remaining elements of R[], if there
# are any
while j < n2:
    arr[k] = R[j]
    j += 1
    k += 1

# l is for left index and r is right index of the
# sub-array of arr to be sorted
def mergeSort(arr,l,r):
    if l < r:

        # Same as (l+r)/2, but avoids overflow for
        # large l and h
        m = (l+(r-1))/2

        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)

# Driver code to test above
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print ("Given array is")
for i in range(n):
    print ("%d" %arr[i]),

mergeSort(arr,0,n-1)
print ("\n\nSorted array is")
for i in range(n):
    print ("%d" %arr[i]),

# This code is contributed by Mohit Kumra

```

Output:

```

Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13

```

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \Theta(n)$$

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\Theta(n \log n)$.

Time complexity of Merge Sort is $\Theta(n \log n)$ in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: $O(n)$

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: No in a typical implementation

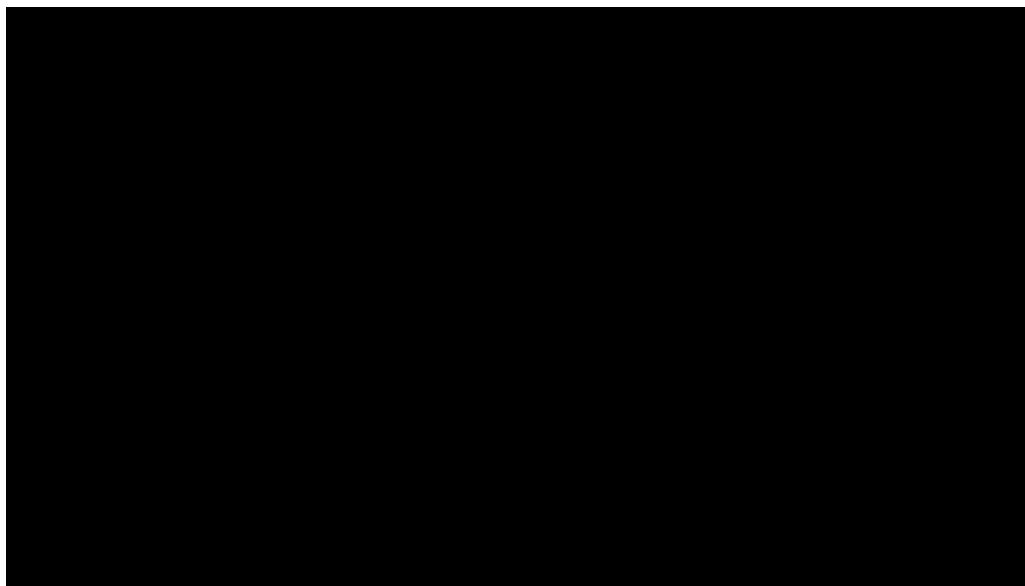
Stable: Yes

Applications of Merge Sort

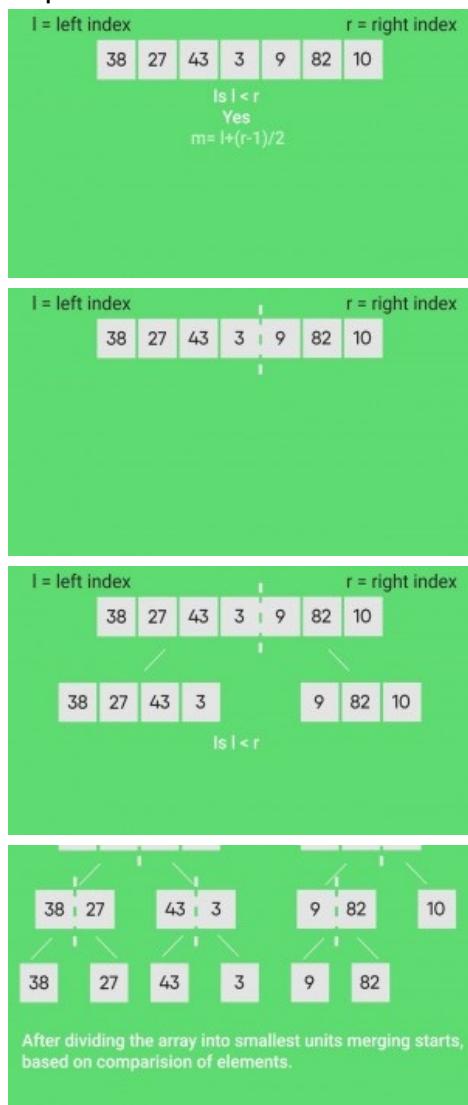
1. Merge Sort is useful for sorting linked lists in $O(n \log n)$ time. In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in $O(1)$ extra space and $O(1)$ time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

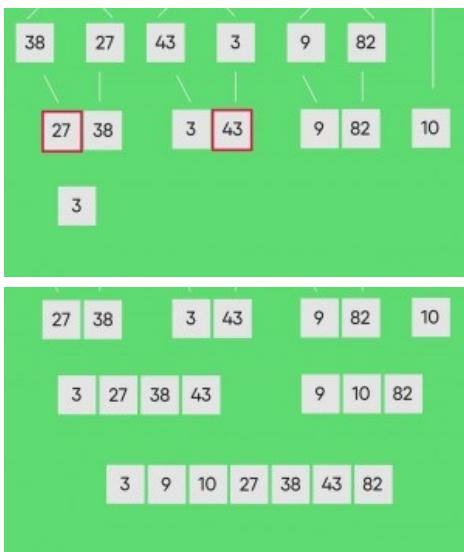
In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at $(x + i \cdot 4)$. Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i^{th} index, we have to travel each and every node from the head to i^{th} node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

2. Inversion Count Problem
3. Used in External Sorting



Snapshots:





Quiz on Merge Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort

Coding practice for sorting.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining elements.

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source [Wikipedia](#))

A [Binary Heap](#) is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index i , the left child can be calculated by $2 * i + 1$ and right child by $2 * i + 2$ (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Lets understand with the help of an example:

```
Input data: 4, 10, 3, 5, 1
        4(0)
       / \
      10(1) 3(2)
     / \
    5(3) 1(4)
```

The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:

```
        4(0)
       / \
      10(1) 3(2)
     / \
    5(3) 1(4)
```

Applying heapify procedure to index 0:

```
        10(0)
       / \
      5(1) 3(2)
     / \
    4(3) 1(4)
```

The heapify procedure calls itself recursively to build heap in top down manner.

C++

```
// C++ program for implementation of Heap Sort
```

```
#include <iostream>
using namespace std;
```

```
// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
```

```
void heapify(int arr[], int n, int i)
```

```
{
```

```
    int largest = i; // Initialize largest as root
```

```
    int l = 2*i + 1; // left = 2*i + 1
```

```
    int r = 2*i + 2; // right = 2*i + 2
```

```
    // If left child is larger than root
```

```
    if (l < n && arr[l] > arr[largest])
```

```
        largest = l;
```

```
    // If right child is larger than largest so far
```

```
    if (r < n && arr[r] > arr[largest])
```

```
        largest = r;
```

```
    // If largest is not root
```

```
    if (largest != i)
```

```
    {
```

```
        swap(arr[i], arr[largest]);
```

```
        // Recursively heapify the affected sub-tree
```

```
        heapify(arr, n, largest);
```

```
}
```

```
}
```

```
// main function to do heap sort
```

```
void heapSort(int arr[], int n)
```

```
{
```

```
    // Build heap (rearrange array)
```

```
    for (int i = n / 2 - 1; i >= 0; i--)
```

```
        heapify(arr, n, i);
```

```
    // One by one extract an element from heap
```

```
    for (int i=n-1; i>=0; i--)
```

```

{
    // Move current root to end
    swap(arr[0], arr[i]);

    // call max heapify on the reduced heap
    heapify(arr, i, 0);
}
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i=0; i<n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

// Driver program
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is \n";
    printArray(arr, n);
}

```

Java

```

// Java program for implementation of Heap Sort
public class HeapSort
{
    public void sort(int arr[])
    {
        int n = arr.length;

        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        // One by one extract an element from heap
        for (int i=n-1; i>=0; i--)
        {
            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }

    // To heapify a subtree rooted with node i which is
    // an index in arr[]. n is size of heap
    void heapify(int arr[], int n, int i)
    {
        int largest = i; // Initialize largest as root
        int l = 2*i + 1; // left = 2*i + 1
        int r = 2*i + 2; // right = 2*i + 2

        // If left child is larger than root
        if (l < n && arr[l] > arr[largest])
            largest = l;

        // If right child is larger than largest so far
        if (r < n && arr[r] > arr[largest])
            largest = r;

        // If largest is not root
        if (largest != i)

```

```

    {
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

/* A utility function to print array of size n */
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i]+" ");
    System.out.println();
}

// Driver program
public static void main(String args[])
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = arr.length;

    HeapSort ob = new HeapSort();
    ob.sort(arr);

    System.out.println("Sorted array is");
    printArray(arr);
}
}

```

Python

```

# Python program for implementation of heap Sort

# To heapify subtree rooted at index i.
# n is size of heap
def heapify(arr, n, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1 # left = 2*i + 1
    r = 2 * i + 2 # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[i] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i] # swap

        # Heapify the root.
        heapify(arr, n, largest)

# The main function to sort an array of given size
def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)

```

```
# Driver code to test above
arr = [ 12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print ("Sorted array is")
for i in range(n):
    print ("%d" %arr[i]),
# This code is contributed by Mohit Kumra
```

Output:

```
Sorted array is
5 6 7 11 12 13
```

[Here](#) is previous C code for reference.

Notes:

Heap sort is an in-place algorithm.

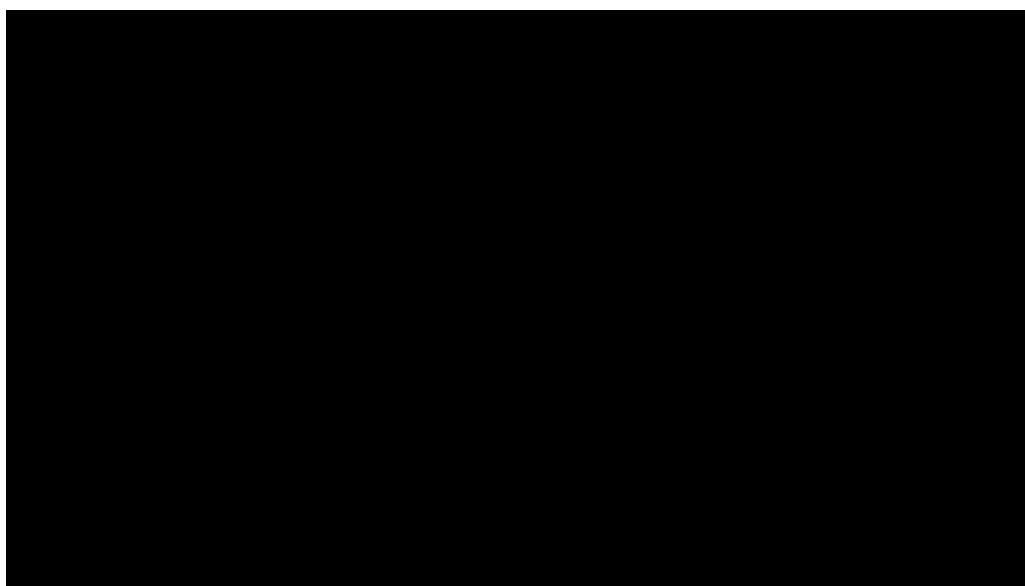
Its typical implementation is not stable, but can be made stable (See [this](#))

Time Complexity: Time complexity of heapify is $O(\log n)$. Time complexity of createAndBuildHeap() is $O(n)$ and overall time complexity of Heap Sort is $O(n\log n)$.

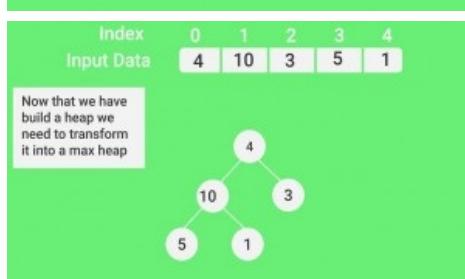
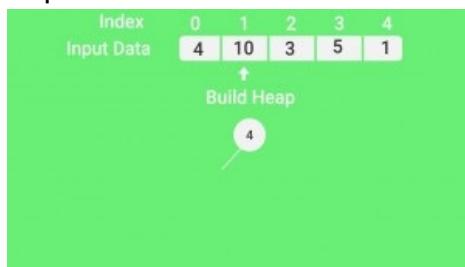
Applications of HeapSort

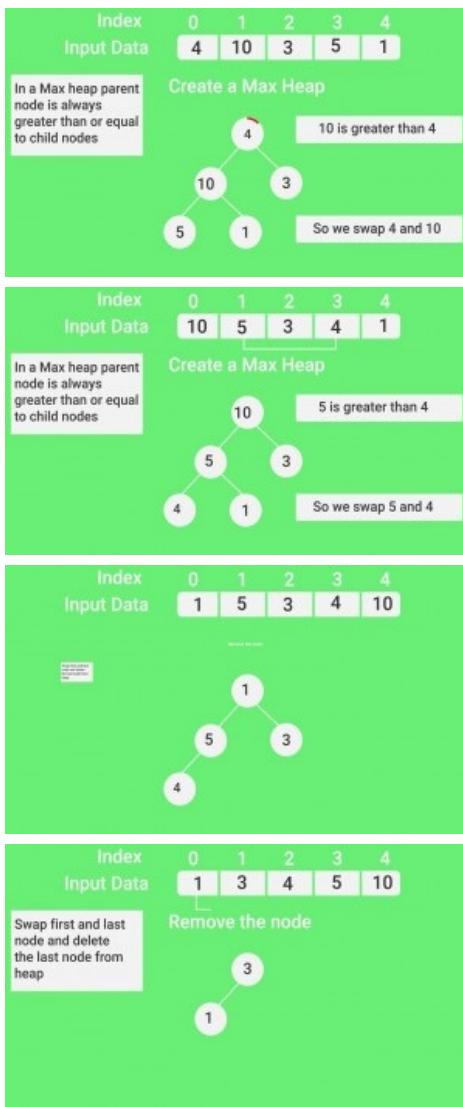
1. Sort a nearly sorted (or K sorted) array
2. k largest(or smallest) elements in an array

Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used. See [Applications of Heap Data Structure](#)



Snapshots:





Quiz on Heap Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

QuickSort, Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort, Pigeonhole Sort

Coding practice for sorting.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

QuickSort

Like [Merge 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.

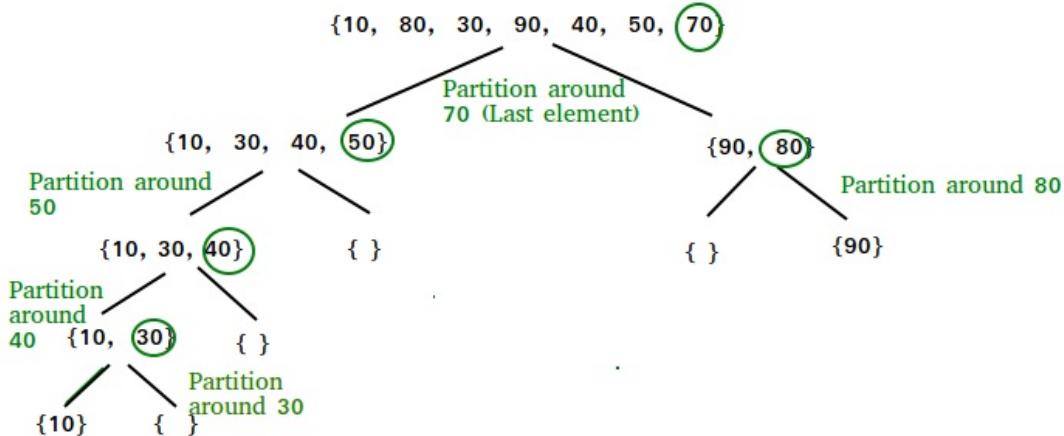
1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

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[pi] is
           now at right place */
        pi = partition(arr[], low, high);

        /* Recursively sort elements before
           partition and after partition */
        quickSort(arr[], low, pi - 1);
        quickSort(arr[], pi + 1, high);
    }
}
```



Partition Algorithm

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.

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

        /* Recursively sort elements before
           partition and after partition */
        quickSort(arr[], low, pi - 1);
        quickSort(arr[], pi + 1, high);
    }
}
```

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 (arr[j] <= pivot)
            swap(arr[i], arr[j]);
            i++;
}
```

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.

We strongly recommend that you click here and practice it, before moving on to the solution.

Implementation:

Following are C++, Java and Python implementations of QuickSort.

C/C++

```

/* C implementation QuickSort */
#include<stdio.h>

// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* 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 */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int 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], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

// A utility function to print array of size n
void printArray (int arr[], int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Function to implement QuickSort()
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

```

```

        swap(&arr[i], &arr[j]);
    }
}
swap(&arr[i + 1], &arr[high]);
return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
         at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}

```

Java

```

// Java program for implementation of QuickSort
class QuickSort
{
    /* 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 */
    int partition(int arr[], int low, int high)
    {
        int pivot = arr[high];
        int i = (low-1); // index of smaller element
        for (int j=low; j<=high-1; j++)
        {
            // If current element is smaller than or
            // equal to pivot
            if (arr[j] <= pivot)
            {
                i++;

```

```

        // swap arr[i] and arr[j]
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

}

// swap arr[i+1] and arr[high] (or pivot)
int temp = arr[i+1];
arr[i+1] = arr[high];
arr[high] = temp;

return i+1;
}

/* The main function that implements QuickSort()
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void sort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is
        now at right place */
        int pi = partition(arr, low, high);

        // Recursively sort elements before
        // partition and after partition
        sort(arr, low, pi-1);
        sort(arr, pi+1, high);
    }
}

/* A utility function to print array of size n */
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i]+" ");
    System.out.println();
}

// Driver program
public static void main(String args[])
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = arr.length;

    QuickSort ob = new QuickSort();
    ob.sort(arr, 0, n-1);

    System.out.println("sorted array");
    printArray(arr);
}
}

/*This code is contributed by Rajat Mishra */

```

Python

```

# Python program for implementation of Quicksort Sort

# 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
def partition(arr,low,high):
    i = ( low-1 )      # index of smaller element
    pivot = arr[high]   # pivot

    for j in range(low , high):

        # If current element is smaller than or
        # equal to pivot
        if  arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low --> Starting index,
# high --> Ending index

# Function to do Quick sort
def quickSort(arr,low,high):
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)

# Driver code to test above
arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
quickSort(arr,0,n-1)
print ("Sorted array is:")
for i in range(n):
    print ("%d" %arr[i]),

# This code is contributed by Mohit Kumra

```

Output:

Sorted array:
1 5 7 8 9 10

Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

$$T(n) = T(k) + T(n-k-1) + \theta(n)$$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.

The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy,

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \theta(n) \\ \text{which is equivalent to} \\ T(n) &= T(n-1) + \theta(n) \end{aligned}$$

The solution of above recurrence is $\theta(n^2)$.

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

$$T(n) = 2T(n/2) + \theta(n)$$

The solution of above recurrence is $\theta(n\log n)$. It can be solved using case 2 of [Master Theorem](#).

Average Case:

To do average case analysis, we need to [consider all possible permutation of array and calculate time taken by every permutation which doesn't look ea](#)

We can get an idea of average case by considering the case when partition puts $O(n/9)$ elements in one set and $O(9n/10)$ elements in other set. Followin

$$T(n) = T(n/9) + T(9n/10) + \theta(n)$$

Solution of above recurrence is also $O(n\log n)$

Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like [Merge Sort](#) and [Heap Sort](#), QuickSort is still a very efficient sorting algorithm.

What is 3-Way QuickSort?

In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for subarrays on left and right of pivot.

Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. If 4 is picked as pivot in Simple Qu

a) arr[i..i] elements less than pivot.

b) arr[i+1..j-1] elements equal to pivot.

c) arr[j..r] elements greater than pivot.

See [this](#) for implementation.

How to implement QuickSort for Linked Lists?

[QuickSort on Singly Linked List](#)

[QuickSort on Doubly Linked List](#)

Can we implement QuickSort Iteratively?

Yes, please refer [Iterative Quick Sort](#).

Why Quick Sort is preferred over MergeSort for sorting Arrays

Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires $O(N)$ extra storage, N denoting the number of elements to be sorted.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of $O(n\log n)$. The worst case time complexity is $O(n^2)$.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

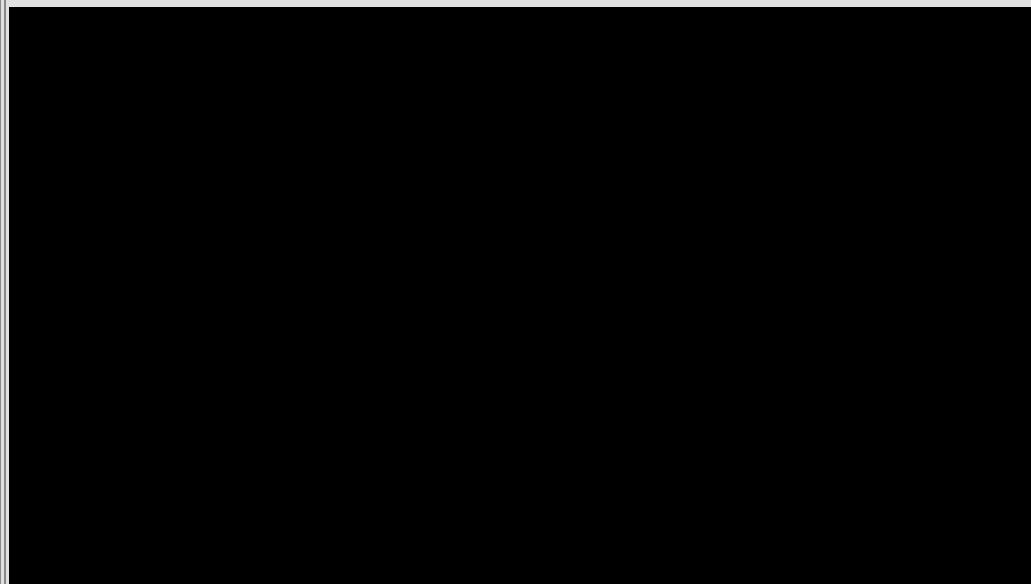
Why MergeSort is preferred over QuickSort for Linked Lists?

In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be contiguous in memory.

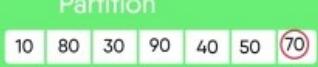
In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of $A[0]$ be 100 . Then we can calculate the address of $A[1]$ as $100 + 4$, $A[2]$ as $100 + 4 * 2$ and so on.

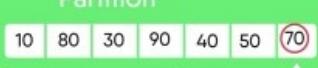
How to optimize QuickSort so that it takes $O(\log n)$ extra space in worst case?

Please see [QuickSort Tail Call Optimization \(Reducing worst case space to Log n \)](#)

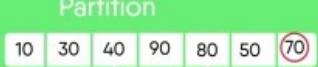


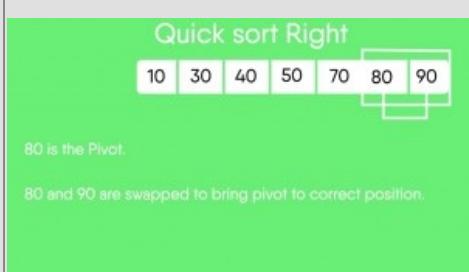
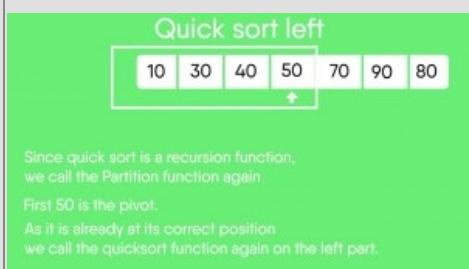
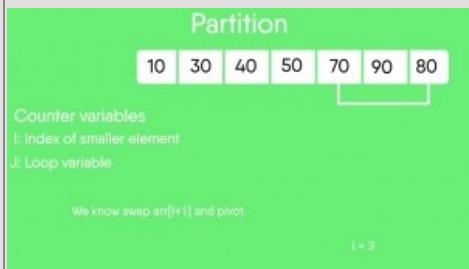
Snapshots:

Partition		
		
Counter variables i: Index of smaller element j: Loop variable We start the loop with initial values.		
Test condition	Actions	Value of variables
arr[j] <= pivot		i = -1 j = 0

Partition		
		
Counter variables i: Index of smaller element j: Loop variable Pass 2		
Test condition	Actions	Value of variables
arr[j] <= pivot 80 < 70 False	No action	i = 0 j = 1

Partition		
		
Counter variables i: Index of smaller element j: Loop variable		
Test condition	Actions	Value of variables
arr[j] <= pivot 30 < 70 True	i++ Swap(arr[i], arr[j])	i = 1 j = 2

Partition		
		
Counter variables i: Index of smaller element j: Loop variable Pass 5		
Test condition	Actions	Value of variables
arr[j] <= pivot 40 < 70 True	i++ Swap(arr[i], arr[j])	i = 2 j = 4



Quiz on QuickSort

References:

<http://en.wikipedia.org/wiki/Quicksort>

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

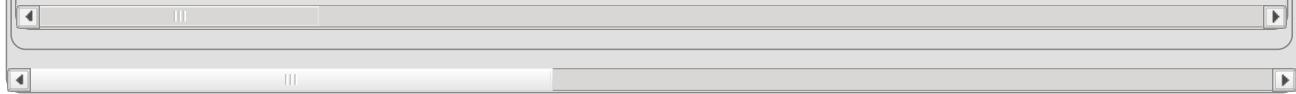
Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort, Pigeonhole Sort.

Coding practice for sorting.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner



See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

Interpolation Search

Given a sorted array of n uniformly distributed values arr[], write a function to search for a particular element x in the array.

Linear Search finds the element in O(n) time, Jump Search takes O(\sqrt{n}) time and Binary Search take O(Log n) time.

The Interpolation Search is an improvement over [Binary Search](#) for instances, where the values in a sorted array are uniformly distributed. Binary Search always goes to middle element to check. On the other hand interpolation search may go to different locations according the value of key being searched. For example if the value of key is closer to the last element, interpolation search is likely to start search toward the end side.

To find the position to be searched, it uses following formula.

```
// The idea of formula is to return higher value of pos
// when element to be searched is closer to arr[hi]. And
// smaller value when closer to arr[lo]
pos = lo + [ (x-arr[lo])*(hi-lo) / (arr[hi]-arr[Lo]) ]  
  
arr[] ==> Array where elements need to be searched
x ==> Element to be searched
lo ==> Starting index in arr[]
hi ==> Ending index in arr[]
```

Algorithm

Rest of the Interpolation algorithm is same except the above partition logic.

Step1: In a loop, calculate the value of "pos" using the probe position formula.

Step2: If it is a match, return the index of the item, and exit.

Step3: If the item is less than arr[pos], calculate the probe position of the left sub-array. Otherwise calculate the same in the right sub-array.

Step4: Repeat until a match is found or the sub-array reduces to zero.

Below is C implementation of algorithm.

```

// C program to implement interpolation search
#include<stdio.h>

// If x is present in arr[0..n-1], then returns
// index of it, else returns -1.
int interpolationSearch(int arr[], int n, int x)
{
    // Find indexes of two corners
    int lo = 0, hi = (n - 1);

    // Since array is sorted, an element present
    // in array must be in range defined by corner
    while (lo <= hi && x >= arr[lo] && x <= arr[hi])
    {
        // Probing the position with keeping
        // uniform distribution in mind.
        int pos = lo + (((double)(hi-lo) /
        (arr[hi]-arr[lo]))*(x - arr[lo]));

        // Condition of target found
        if (arr[pos] == x)
            return pos;

        // If x is larger, x is in upper part
        if (arr[pos] < x)
            lo = pos + 1;

        // If x is smaller, x is in lower part
        else
            hi = pos - 1;
    }
    return -1;
}

// Driver Code
int main()
{
    // Array of items on which search will
    // be conducted.
    int arr[] = {10, 12, 13, 16, 18, 19, 20, 21, 22, 23,
                24, 33, 35, 42, 47};
    int n = sizeof(arr)/sizeof(arr[0]);

    int x = 18; // Element to be searched
    int index = interpolationSearch(arr, n, x);

    // If element was found
    if (index != -1)
        printf("Element found at index %d", index);
    else
        printf("Element not found.");
    return 0;
}

```

Output :

Element found at index 4

Time Complexity : If elements are uniformly distributed, then **O ($\log \log n$)**. In worst case it can take upto O(n).

Auxiliary Space : O(1)

This article is contributed by **Aayu sachdev**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

K'th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

We recommend to read following post as a prerequisite of this post.

K'th Smallest/Largest Element in Unsorted Array | Set 1

Given an array and a number k where k is smaller than size of array, we need to find the k'th smallest element in the given array. It is given that all array elements are distinct.

Examples:

```
Input: arr[] = {7, 10, 4, 3, 20, 15}
```

```
k = 3
```

```
Output: 7
```

```
Input: arr[] = {7, 10, 4, 3, 20, 15}
```

```
k = 4
```

```
Output: 10
```

We have discussed three different solutions [here](#).

In this post method 4 is discussed which is mainly an extension of method 3 (QuickSelect) discussed in the [previous](#) post. The idea is to randomly pick a pivot element. To implement randomized partition, we use a random function, `rand()` to generate index between l and r, swap the element at randomly generated index with the last element, and finally call the standard partition process which uses last element as pivot.

Following is implementation of above Randomized QuickSelect.

C/C++

```
// C++ implementation of randomized quickSelect
#include<iostream>
#include<climits>
#include<cstdlib>
using namespace std;

int randomPartition(int arr[], int l, int r);

// This function returns k'th smallest element in arr[l..r] using
// QuickSort based method. ASSUMPTION: ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = randomPartition(arr, l, r);

        // If position is same as k
        if (pos - l == k - 1)
            return arr[pos];
        if (pos - l > k - 1) // If position is more, recur for left subarray
            return kthSmallest(arr, l, pos - 1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos + 1, r, k - pos + l - 1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Standard partition process of QuickSort(). It considers the last
// element as pivot and moves all smaller element to left of it and
// greater elements to right. This function is used by randomPartition()
```

```

int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

// Picks a random pivot element between l and r and partitions
// arr[l..r] around the randomly picked element using partition()
int randomPartition(int arr[], int l, int r)
{
    int n = r-l+1;
    int pivot = rand() % n;
    swap(&arr[l + pivot], &arr[r]);
    return partition(arr, l, r);
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is " << kthSmallest(arr, 0, n-1, k);
    return 0;
}

```

Java

```

// Java program to find k'th smallest element in expected
// linear time
class KthSmallest
{
    // This function returns k'th smallest element in arr[l..r]
    // using QuickSort based method. ASSUMPTION: ALL ELEMENTS
    // IN ARR[] ARE DISTINCT
    int kthSmallest(int arr[], int l, int r, int k)
    {
        // If k is smaller than number of elements in array
        if (k > 0 && k <= r - l + 1)
        {
            // Partition the array around a random element and
            // get position of pivot element in sorted array
            int pos = randomPartition(arr, l, r);

            // If position is same as k
            if (pos - l == k - 1)
                return arr[pos];

            // If position is more, recur for left subarray
            if (pos - l > k - 1)
                return kthSmallest(arr, l, pos - 1, k);

            // Else recur for right subarray
            return kthSmallest(arr, pos + 1, r, k - pos + l - 1);
        }

        // If k is more than number of elements in array
        return Integer.MAX_VALUE;
    }

    // Utility method to swap arr[i] and arr[j]
    void swap(int arr[], int i, int j)
    {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}

```

```

        arr[j] = temp;
    }

// Standard partition process of QuickSort(). It considers
// the last element as pivot and moves all smaller element
// to left of it and greater elements to right. This function
// is used by randomPartition()
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(arr, i, j);
            i++;
        }
    }
    swap(arr, i, r);
    return i;
}

// Picks a random pivot element between l and r and
// partitions arr[l..r] around the randomly picked
// element using partition()
int randomPartition(int arr[], int l, int r)
{
    int n = r-l+1;
    int pivot = (int)(Math.random()) % n;
    swap(arr, l + pivot, r);
    return partition(arr, l, r);
}

// Driver method to test above
public static void main(String args[])
{
    KthSmallest ob = new KthSmallest();
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = arr.length, k = 3;
    System.out.println("K'th smallest element is "+
        ob.kthSmallest(arr, 0, n-1, k));
}
}
/*This code is contributed by Rajat Mishra*/

```

Output:

K'th smallest element is 5

Time Complexity:

The worst case time complexity of the above solution is still $O(n^2)$. In worst case, the randomized function may always pick a corner element.

The expected time complexity of above randomized QuickSelect is $\Theta(n)$, see [CLRS book](#) or [MIT video lecture](#) for proof. The assumption in the analysis is, random number generator is equally likely to generate any number in the input range.

Sources:

[MIT Video Lecture on Order Statistics, Median](#)

[Introduction to Algorithms](#) by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Randomized
Searching
Order-Statistics
Quick Sort

Given a sorted array and a number x, find the pair in array whose sum is closest to x

Given a sorted array and a number x, find a pair in array whose sum is closest to x.

Examples:

```
Input: arr[] = {10, 22, 28, 29, 30, 40}, x = 54
Output: 22 and 30
```

```
Input: arr[] = {1, 3, 4, 7, 10}, x = 15
Output: 4 and 10
```

A simple solution is to consider every pair and keep track of closest pair (absolute difference between pair sum and x is minimum). Finally print the closest pair. Time complexity of this solution is $O(n^2)$

An efficient solution can find the pair in $O(n)$ time. The idea is similar to method 2 of [this post](#). Following is detailed algorithm.

- 1) Initialize a variable diff as infinite (Diff is used to store the difference between pair and x). We need to find the minimum diff.
- 2) Initialize two index variables l and r in the given sorted array.
 - (a) Initialize first to the leftmost index: $l = 0$
 - (b) Initialize second the rightmost index: $r = n-1$
- 3) Loop while l

Following is C++ implementation of above algorithm.

C++

```
// Simple C++ program to find the pair with sum closest to a given no.
#include <iostream>
#include <climits>
#include <cstdlib>
using namespace std;

// Prints the pair with sum closest to x
void printClosest(int arr[], int n, int x)
{
    int res_l, res_r; // To store indexes of result pair

    // Initialize left and right indexes and difference between
    // pair sum and x
    int l = 0, r = n-1, diff = INT_MAX;

    // While there are elements between l and r
    while (r > l)
    {
        // Check if this pair is closer than the closest pair so far
        if (abs(arr[l] + arr[r] - x) < diff)
        {
            res_l = l;
            res_r = r;
            diff = abs(arr[l] + arr[r] - x);
        }

        // If this pair has more sum, move to smaller values.
        if (arr[l] + arr[r] > x)
            r--;
        else // Move to larger values
            l++;
    }

    cout << "The closest pair is " << arr[res_l] << " and " << arr[res_r];
}

// Driver program to test above functions
int main()
{
    int arr[] = {10, 22, 28, 29, 30, 40}, x = 54;
    int n = sizeof(arr)/sizeof(arr[0]);
    printClosest(arr, n, x);
    return 0;
}
```

```
}
```

Java

```
// Java program to find pair with sum closest to x
import java.io.*;
import java.util.*;
import java.lang.Math;

class CloseSum {

    // Prints the pair with sum closest to x
    static void printClosest(int arr[], int n, int x)
    {
        int res_l=0, res_r=0; // To store indexes of result pair

        // Initialize left and right indexes and difference between
        // pair sum and x
        int l = 0, r = n-1, diff = Integer.MAX_VALUE;

        // While there are elements between l and r
        while (r > l)
        {
            // Check if this pair is closer than the closest pair so far
            if (Math.abs(arr[l] + arr[r] - x) < diff)
            {
                res_l = l;
                res_r = r;
                diff = Math.abs(arr[l] + arr[r] - x);
            }

            // If this pair has more sum, move to smaller values.
            if (arr[l] + arr[r] > x)
                r--;
            else // Move to larger values
                l++;
        }

        System.out.println(" The closest pair is "+arr[res_l]+" and "+ arr[res_r]);
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int arr[] = {10, 22, 28, 29, 30, 40}, x = 54;
        int n = arr.length;
        printClosest(arr, n, x);
    }
}
/*This code is contributed by Devesh Agrawal*/
```

Output:

```
The closest pair is 22 and 30
```

This article is contributed by **Harsh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed

GATE CS Corner

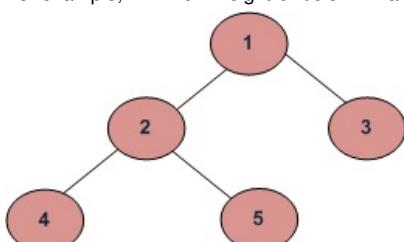
See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.

Category: Algorithms

Find Minimum Depth of a Binary Tree

Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from root node down to the nearest leaf node.

For example, minimum height of below Binary Tree is 2.



Note that the path must end on a leaf node. For example, minimum height of below Binary Tree is also 2.

```
10  
/  
5
```

We strongly recommend you to minimize your browser and try this yourself first.

The idea is to traverse the given Binary Tree. For every node, check if it is a leaf node. If yes, then return 1. If not leaf node then if left subtree is NULL, then recur for right subtree. And if right subtree is NULL, then recur for left subtree. If both left and right subtrees are not NULL, then take the minimum of two heights.

Below is implementation of the above idea.

C++

```
// C++ program to find minimum depth of a given Binary Tree  
#include<bits/stdc++.h>  
using namespace std;  
  
// A BT Node  
struct Node
```

```

{
    int data;
    struct Node* left, *right;
};

int minDepth(Node *root)
{
    // Corner case. Should never be hit unless the code is
    // called on root = NULL
    if (root == NULL)
        return 0;

    // Base case : Leaf Node. This accounts for height = 1.
    if (root->left == NULL && root->right == NULL)
        return 1;

    // If left subtree is NULL, recur for right subtree
    if (root->left)
        return minDepth(root->right) + 1;

    // If right subtree is NULL, recur for left subtree
    if (root->right)
        return minDepth(root->left) + 1;

    return min(minDepth(root->left), minDepth(root->right)) + 1;
}

// Utility function to create new Node
Node *newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return (temp);
}

// Driver program
int main()
{
    // Let us construct the Tree shown in the above figure
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    cout << minDepth(root);
    return 0;
}

```

Java

```

/* Java implementation to find minimum depth
of a given Binary tree */

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;
    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}
public class BinaryTree
{
    //Root of the Binary Tree
    Node root;

    int minimumDepth()
    {

```

```

        return minimumDepth(root);
    }

/* Function to calculate the minimum depth of the tree */
int minimumDepth(Node root)
{
    // Corner case. Should never be hit unless the code is
    // called on root = NULL
    if (root == null)
        return 0;

    // Base case : Leaf Node. This accounts for height = 1.
    if (root.left == null && root.right == null)
        return 1;

    // If left subtree is NULL, recur for right subtree
    if (root.left == null)
        return minimumDepth(root.right) + 1;

    // If right subtree is NULL, recur for right subtree
    if (root.right == null)
        return minimumDepth(root.left) + 1;

    return Math.min(minimumDepth(root.left),
                   minimumDepth(root.right)) + 1;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    System.out.println("The minimum depth of "+
                       "binary tree is : " + tree.minimumDepth());
}
}

```

Python

```

# Python program to find minimum depth of a given Binary Tree

# Tree node
class Node:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

def minDepth(root):
    # Corner Case. Should never be hit unless the code is
    # called on root = NULL
    if root is None:
        return 0

    # Base Case : Leaf node. This accounts for height = 1
    if root.left is None and root.right is None:
        return 1

    # If left subtree is Null, recur for right subtree
    if root.left is None:
        return minDepth(root.right)+1

    # If right subtree is Null , recur for left subtree
    if root.right is None:
        return minDepth(root.left) +1

    return min(minDepth(root.left), minDepth(root.right))+1

```

```

# Driver Program
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print minDepth(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

2

Time complexity of above solution is $O(n)$ as it traverses the tree only once.

Thanks to [Gaurav Ahirwar](#) for providing above solution.

The above method may end up with complete traversal of Binary Tree even when the topmost leaf is close to root. A **Better Solution** is to do Level Order Traversal. While doing traversal, returns depth of the first encountered leaf node. Below is implementation of this solution.

C

```

// C++ program to find minimum depth of a given Binary Tree
#include<bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A queue item (Stores pointer to node and an integer)
struct qItem
{
    Node *node;
    int depth;
};

// Iterative method to find minimum depth of Binary Tree
int minDepth(Node *root)
{
    // Corner Case
    if (root == NULL)
        return 0;

    // Create an empty queue for level order traversal
    queue<qItem> q;

    // Enqueue Root and initialize depth as 1
    qItem qi = {root, 1};
    q.push(qi);

    // Do level order traversal
    while (q.empty() == false)
    {
        // Remove the front queue item
        qi = q.front();
        q.pop();

        // Get details of the remove item
        Node *node = qi.node;
        int depth = qi.depth;

        // If this is the first leaf node seen so far
        // Then return its depth as answer
        if (node->left == NULL && node->right == NULL)
            return depth;

        // If left subtree is not NULL, add it to queue
    }
}

```

```

if (node->left != NULL)
{
    qi.node = node->left;
    qi.depth = depth + 1;
    q.push(qi);
}

// If right subtree is not NULL, add it to queue
if (node->right != NULL)
{
    qi.node = node->right;
    qi.depth = depth+1;
    q.push(qi);
}
}

return 0;
}

// Utility function to create a new tree Node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << minDepth(root);
    return 0;
}

```

Python

```

# Python program to find minimum depth of a given Binary Tree

# A Binary Tree node
class Node:
    # Utility to create new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def minDepth(root):
    # Corner Case
    if root is None:
        return 0

    # Create an empty queue for level order traversal
    q = []

    # Enqueue root and initialize depth as 1
    q.append({'node': root, 'depth': 1})

    # Do level order traversal
    while(len(q)>0):
        # Remove the front queue item
        queueItem = q.pop(0)

        # Get details of the removed item
        node = queueItem['node']
        depth = queueItem['depth']

```

```

# If this is the first leaf node seen so far
# then return its depth as answer
if node.left is None and node.right is None:
    return depth

# If left subtree is not None, add it to queue
if node.left is not None:
    q.append({'node': node.left, 'depth': depth+1})

# if right subtree is not None, add it to queue
if node.right is not None:
    q.append({'node': node.right, 'depth': depth+1})

# Driver program to test above function
# Lets construct a binary tree shown in above diagram
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print minDepth(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

2

Thanks to Manish Chauhan for suggesting above idea and Ravi for providing implementation.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Trees

Maximum Path Sum in a Binary Tree

Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree.

Example:

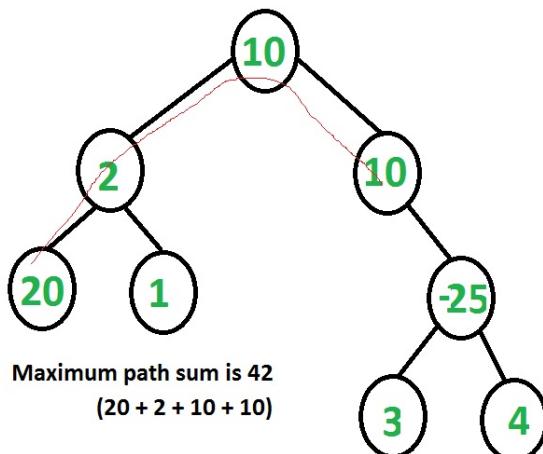
Input: Root of below tree

1
/\
2 3

Output: 6

See below diagram for another example.

1+2+3



We strongly recommend you to minimize your browser and try this yourself first.

For each node there can be four ways that the max path goes through the node:

1. Node only

2. Max path through Left Child + Node
3. Max path through Right Child + Node
4. Max path through Left Child + Node + Max path through Right Child

The idea is to keep trace of four paths and pick up the max one in the end. An important thing to note is, root of every subtree need to return maximum path sum such that at most one child of root is involved. This is needed for parent function call. In below code, this sum is stored in 'max_single' and returned by the recursive function.

C++

```
// C/C++ program to find maximum path sum in Binary Tree
#include<bits/stdc++.h>
using namespace std;

// A binary tree node
struct Node
{
    int data;
    struct Node* left, *right;
};

// A utility function to allocate a new node
struct Node* newNode(int data)
{
    struct Node* newNode = new Node;
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return (newNode);
}

// This function returns overall maximum path sum in 'res'
// And returns max path sum going through root.
int findMaxUtil(Node* root, int &res)
{
    //Base Case
    if (root == NULL)
        return 0;

    // l and r store maximum path sum going through left and
    // right child of root respectively
    int l = findMaxUtil(root->left,res);
    int r = findMaxUtil(root->right,res);

    // Max path for parent call of root. This path must
    // include at-most one child of root
    int max_single = max(max(l, r) + root->data, root->data);

    // Max Top represents the sum when the Node under
    // consideration is the root of the maxsum path and no
    // ancestors of root are there in max sum path
    int max_top = max(max_single, l + r + root->data);

    res = max(res, max_top); // Store the Maximum Result.

    return max_single;
}

// Returns maximum path sum in tree with given root
int findMaxSum(Node *root)
{
    // Initialize result
    int res = INT_MIN;

    // Compute and return result
    findMaxUtil(root, res);
    return res;
}

// Driver program
int main(void)
{
    struct Node *root = newNode(10);
```

```

root->left    = newNode(2);
root->right   = newNode(10);
root->left->left = newNode(20);
root->left->right = newNode(1);
root->right->right = newNode(-25);
root->right->right->left  = newNode(3);
root->right->right->right = newNode(4);
cout << "Max path sum is " << findMaxSum(root);
return 0;
}

```

Java

```

// Java program to find maximum path sum in Binary Tree

/* Class containing left and right child of current
node and key value*/
class Node {

    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

// An object of Res is passed around so that the
// same value can be used by multiple recursive calls.
class Res {
    public int val;
}

class BinaryTree {

    // Root of the Binary Tree
    Node root;

    // This function returns overall maximum path sum in 'res'
    // And returns max path sum going through root.
    int findMaxUtil(Node node, Res res)
    {

        // Base Case
        if (node == null)
            return 0;

        // l and r store maximum path sum going through left and
        // right child of root respectively
        int l = findMaxUtil(node.left, res);
        int r = findMaxUtil(node.right, res);

        // Max path for parent call of root. This path must
        // include at-most one child of root
        int max_single = Math.max(Math.max(l, r) + node.data,
                               node.data);

        // Max Top represents the sum when the Node under
        // consideration is the root of the maxsum path and no
        // ancestors of root are there in max sum path
        int max_top = Math.max(max_single, l + r + node.data);

        // Store the Maximum Result.
        res.val = Math.max(res.val, max_top);

        return max_single;
    }

    int findMaxSum() {
        return findMaxSum(root);
    }
}

```

```

// Returns maximum path sum in tree with given root
int findMaxSum(Node node) {

    // Initialize result
    // int res2 = Integer.MIN_VALUE;
    Res res = new Res();
    res.val = Integer.MIN_VALUE;

    // Compute and return result
    findMaxUtil(node, res);
    return res.val;
}

/* Driver program to test above functions */
public static void main(String args[]) {
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(10);
    tree.root.left = new Node(2);
    tree.root.right = new Node(10);
    tree.root.left.left = new Node(20);
    tree.root.left.right = new Node(1);
    tree.root.right.right = new Node(-25);
    tree.root.right.right.left = new Node(3);
    tree.root.right.right.right = new Node(4);
    System.out.println("maximum path sum is : " +
        tree.findMaxSum());
}
}

```

Python

```

# Python program to find maximum path sum in Binary Tree

# A Binary Tree Node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # This function returns overall maximum path sum in 'res'
    # And returns max path sum going through root
    def findMaxUtil(root):

        # Base Case
        if root is None:
            return 0

        # l and r store maximum path sum going through left
        # and right child of root respectively
        l = findMaxUtil(root.left)
        r = findMaxUtil(root.right)

        # Max path for parent call of root. This path
        # must include at most one child of root
        max_single = max(max(l, r) + root.data, root.data)

        # Max top represents the sum when the node under
        # consideration is the root of the maxSum path and
        # no ancestor of root are there in max sum path
        max_top = max(max_single, l+r+root.data)

        # Static variable to store the changes
        # Store the maximum result
        findMaxUtil.res = max(findMaxUtil.res, max_top)

        return max_single

    # Return maximum path sum in tree with given root
    def findMaxSum(root):

```

```

# Initialize result
findMaxUtil.res = float("-inf")

# Compute and return result
findMaxUtil(root)
return findMaxUtil.res

# Driver program
root = Node(10)
root.left = Node(2)
root.right = Node(10);
root.left.left = Node(20);
root.left.right = Node(1);
root.right.right = Node(-25);
root.right.right.left = Node(3);
root.right.right.right = Node(4);
print "Max path sum is ",findMaxSum(root);

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

Max path sum is 42

Time Complexity: O(n) where n is number of nodes in Binary Tree.

This article is contributed by **Anmol Varshney** (FB Profile: <https://www.facebook.com/anmolvarshney695>). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Trees

Check if a given array can represent Preorder Traversal of Binary Search Tree

Given an array of numbers, return true if given array can represent preorder traversal of a Binary Search Tree, else return false. Expected time complexity is O(n).

Examples:

Input: pre[] = {2, 4, 3}
 Output: true
 Given array can represent preorder traversal
 of below tree
 2
 \
 4
 /
 3

Input: pre[] = {2, 4, 1}
 Output: false
 Given array cannot represent preorder traversal
 of a Binary Search Tree.

Input: pre[] = {40, 30, 35, 80, 100}
 Output: true
 Given array can represent preorder traversal
 of below tree
 40
 / \/
 30 80
 \ \/
 35 100

Input: pre[] = {40, 30, 35, 20, 80, 100}
 Output: false
 Given array cannot represent preorder traversal

We strongly recommend that you click here and practice it, before moving on to the solution.

A Simple Solution is to do following for every node $\text{pre}[i]$ starting from first one.

- 1) Find the first greater value on right side of current node.

Let the index of this node be j . Return true if following conditions hold. Else return false

- (i) All values after the above found greater value are greater than current node.
- (ii) Recursive calls for the subarrays $\text{pre}[i+1..j-1]$ and $\text{pre}[j+1..n-1]$ also return true.

Time Complexity of the above solution is $O(n^2)$

An Efficient Solution can solve this problem in $O(n)$ time. The idea is to use a stack. This problem is similar to Next (or closest) Greater Element problem. Here we find next greater element and after finding next greater, if we find a smaller element, then return false.

- 1) Create an empty stack.
- 2) Initialize root as INT_MIN.
- 3) Do following for every element $\text{pre}[i]$
 - a) If $\text{pre}[i]$ is smaller than current root, return false.
 - b) Keep removing elements from stack while $\text{pre}[i]$ is greater than stack top. Make the last removed item as new root (to be compared next).
At this point, $\text{pre}[i]$ is greater than the removed root
(That is why if we see a smaller element in step a), we return false)
 - c) push $\text{pre}[i]$ to stack (All elements in stack are in decreasing order)

Below is implementation of above idea.

C++

```
// C++ program for an efficient solution to check if
// a given array can represent Preorder traversal of
// a Binary Search Tree
#include<bits/stdc++.h>
using namespace std;

bool canRepresentBST(int pre[], int n)
{
    // Create an empty stack
    stack<int> s;

    // Initialize current root as minimum possible
    // value
    int root = INT_MIN;

    // Traverse given array
    for (int i=0; i<n; i++)
    {
        // If we find a node who is on right side
        // and smaller than root, return false
        if (pre[i] < root)
            return false;

        // If pre[i] is in right subtree of stack top,
        // Keep removing items smaller than pre[i]
        // and make the last removed item as new
        // root.
        while (!s.empty() && s.top()<pre[i])
        {
            root = s.top();
            s.pop();
        }

        // At this point either stack is empty or
        // pre[i] is smaller than root, push pre[i]
        s.push(pre[i]);
    }
}
```

```

        s.push(pre[i]);
    }
    return true;
}

// Driver program
int main()
{
    int pre1[] = {40, 30, 35, 80, 100};
    int n = sizeof(pre1)/sizeof(pre1[0]);
    canRepresentBST(pre1, n)? cout << "true\n":
        cout << "false\n";

    int pre2[] = {40, 30, 35, 20, 80, 100};
    n = sizeof(pre2)/sizeof(pre2[0]);
    canRepresentBST(pre2, n)? cout << "true\n":
        cout << "false\n";

    return 0;
}

```

Java

```

// Java program for an efficient solution to check if
// a given array can represent Preorder traversal of
// a Binary Search Tree
import java.util.Stack;

class BinarySearchTree {

    boolean canRepresentBST(int pre[], int n) {
        // Create an empty stack
        Stack<Integer> s = new Stack<Integer>();

        // Initialize current root as minimum possible
        // value
        int root = Integer.MIN_VALUE;

        // Traverse given array
        for (int i = 0; i < n; i++) {
            // If we find a node who is on right side
            // and smaller than root, return false
            if (pre[i] < root) {
                return false;
            }

            // If pre[i] is in right subtree of stack top,
            // Keep removing items smaller than pre[i]
            // and make the last removed item as new
            // root.
            while (!s.empty() && s.peek() < pre[i]) {
                root = s.peek();
                s.pop();
            }

            // At this point either stack is empty or
            // pre[i] is smaller than root, push pre[i]
            s.push(pre[i]);
        }
        return true;
    }

    public static void main(String args[]) {
        BinarySearchTree bst = new BinarySearchTree();
        int[] pre1 = new int[]{40, 30, 35, 80, 100};
        int n = pre1.length;
        if (bst.canRepresentBST(pre1, n) == true) {
            System.out.println("true");
        } else {
            System.out.println("false");
        }
        int[] pre2 = new int[]{40, 30, 35, 20, 80, 100};
        int n1 = pre2.length;
    }
}

```

```

        if (bst.canRepresentBST(pre2, n) == true) {
            System.out.println("true");
        } else {
            System.out.println("false");
        }
    }
}

//This code is contributed by Mayank Jaiswal

```

Python

```

# Python program for an efficient solution to check if
# a given array can represent Preorder traversal of
# a Binary Search Tree

INT_MIN = -2**32

def canRepresentBST(pre):
    # Create an empty stack
    s = []

    # Initialize current root as minimum possible value
    root = INT_MIN

    # Traverse given array
    for value in pre:
        #NOTE:value is equal to pre[i] according to the
        #given algo

        # If we find a node who is on the right side
        # and smaller than root, return False
        if value < root :
            return False

        # If value(pre[i]) is in right subtree of stack top,
        # Keep removing items smaller than value
        # and make the last removed items as new root
        while(len(s) > 0 and s[-1] < value) :
            root = s.pop()

        # At this point either stack is empty or value
        # is smaller than root, push value
        s.append(value)

    return True

# Driver Program
pre1 = [40 , 30 ,35 , 80 , 100]
print "true" if canRepresentBST(pre1) == True else "false"
pre2 = [40 , 30 ,35 , 20 , 80 , 100]
print "true" if canRepresentBST(pre2) == True else "false"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

true
false

```

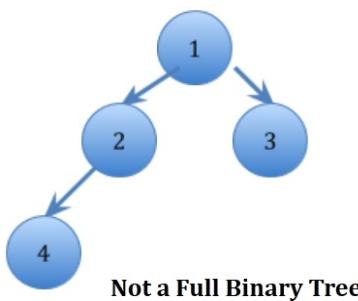
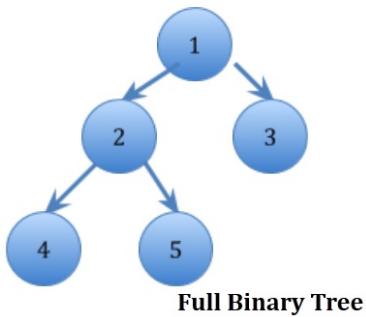
This article is contributed by **Romil Punetha**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Check whether a binary tree is a full binary tree or not

A full binary tree is defined as a binary tree in which all nodes have either zero or two child nodes. Conversely, there is no node in a full binary tree, which has one child node. More information about full binary trees can be found [here](#).

For Example:



We strongly recommend to minimize your browser and try this yourself first.

To check whether a binary tree is a full binary tree we need to test the following cases:-

- 1) If a binary tree node is NULL then it is a full binary tree.
- 2) If a binary tree node does have empty left and right sub-trees, then it is a full binary tree by definition
- 3) If a binary tree node has left and right sub-trees, then it is a part of a full binary tree by definition. In this case recursively check if the left and right sub-trees are also binary trees themselves.
- 4) In all other combinations of right and left sub-trees, the binary tree is not a full binary tree.

Following is the implementation for checking if a binary tree is a full binary tree.

C

```
// C program to check whether a given Binary Tree is full or not
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Tree node structure */
struct Node
{
    int key;
    struct Node *left, *right;
};

/* Helper function that allocates a new node with the
   given key and NULL left and right pointer. */
struct Node *newNode(char k)
{
    struct Node *node = (struct Node*)malloc(sizeof(struct Node));
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

/* This function tests if a binary tree is a full binary tree. */
bool isFullTree (struct Node* root)
{
    // If empty tree
}
```

```

if (root == NULL)
    return true;

// If leaf node
if (root->left == NULL && root->right == NULL)
    return true;

// If both left and right are not NULL, and left & right subtrees
// are full
if ((root->left) && (root->right))
    return (isFullTree(root->left) && isFullTree(root->right));

// We reach here when none of the above if conditions work
return false;
}

// Driver Program
int main()
{
    struct Node* root = NULL;
    root = newNode(10);
    root->left = newNode(20);
    root->right = newNode(30);

    root->left->right = newNode(40);
    root->left->left = newNode(50);
    root->right->left = newNode(60);
    root->right->right = newNode(70);

    root->left->left->left = newNode(80);
    root->left->left->right = newNode(90);
    root->left->right->left = newNode(80);
    root->left->right->right = newNode(90);
    root->right->left->left = newNode(80);
    root->right->left->right = newNode(90);
    root->right->right->left = newNode(80);
    root->right->right->right = newNode(90);

    if (isFullTree(root))
        printf("The Binary Tree is full\n");
    else
        printf("The Binary Tree is not full\n");

    return(0);
}

```

Java

```

// Java program to check if binay tree is full or not

/* Tree node structure */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* this function checks if a binary tree is full or not */
    boolean isFullTree(Node node)
    {
        // if empty tree
        if(node == null)
            return true;

```

```

// if leaf node
if(node.left == null && node.right == null )
    return true;

// if both left and right subtrees are not null
// they are full
if((node.left!=null) && (node.right!=null))
    return (isFullTree(node.left) && isFullTree(node.right));

// if none work
return false;
}

// Driver program
public static void main(String args[])
{
    BinaryTreeNode tree = new BinaryTreeNode();
    tree.root = new Node(10);
    tree.root.left = new Node(20);
    tree.root.right = new Node(30);
    tree.root.left.right = new Node(40);
    tree.root.left.left = new Node(50);
    tree.root.right.left = new Node(60);
    tree.root.left.left.left = new Node(80);
    tree.root.right.right = new Node(70);
    tree.root.left.left.right = new Node(90);
    tree.root.left.right.left = new Node(80);
    tree.root.left.right.right = new Node(90);
    tree.root.right.left.left = new Node(80);
    tree.root.right.left.right = new Node(90);
    tree.root.right.right.left = new Node(80);
    tree.root.right.right.right = new Node(90);

    if(tree.isFullTree(tree.root))
        System.out.print("The binary tree is full");
    else
        System.out.print("The binary tree is not full");
}
}

// This code is contributed by Mayank Jaiswal

```

Python

```

# Python program to check whether given Binary tree is full or not

# Tree node structure
class Node:

    # Constructor of the node class for creating the node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    # Checks if the binary tree is full or not
    def isFullTree(root):

        # If empty tree
        if root is None:
            return True

        # If leaf node
        if root.left is None and root.right is None:
            return True

        # If both left and right subtress are not None and
        # left and right subtress are full
        if root.left is not None and root.right is not None:
            return (isFullTree(root.left) and isFullTree(root.right))


```

```

# We reach here when none of the above if conditions work
return False

# Driver Program
root = Node(10);
root.left = Node(20);
root.right = Node(30);

root.left.right = Node(40);
root.left.left = Node(50);
root.right.left = Node(60);
root.right.right = Node(70);

root.left.left.left = Node(80);
root.left.left.right = Node(90);
root.left.right.left = Node(80);
root.left.right.right = Node(90);
root.right.left.left = Node(80);
root.right.left.right = Node(90);
root.right.right.left = Node(80);
root.right.right.right = Node(90);

if isFullTree(root):
    print "The Binary tree is full"
else:
    print "Binary tree is not full"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

The Binary Tree is full

Time complexity of the above code is O(n) where n is number of nodes in given binary tree.

This article is contributed by **Gaurav Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Trees

Bottom View of a Binary Tree

Given a Binary Tree, we need to print the bottom view from left to right. A node x is there in output if x is the bottommost node at its horizontal distance. Horizontal distance of left child of a node x is equal to horizontal distance of x minus 1, and that of right child is horizontal distance of x plus 1.

Examples:

```

20
 / \
8   22
 / \  \
5   3   25
  /\
 10  14

```

For the above tree the output should be 5, 10, 3, 14, 25.

If there are multiple bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottom-most nodes at horizontal distance 0, we need to print 4.

```

20
 / \
8   22
 / \ / \
5   3 4  25

```

For the above tree the output should be 5, 10, 4, 14, 25.

We strongly recommend to minimize your browser and try this yourself first.

The following are steps to print Bottom View of Binary Tree.

1. We put tree nodes in a queue for the level order traversal.

2. Start with the horizontal distance(hd) 0 of the root node, keep on adding left child to queue along with the horizontal distance as hd-1 and right child as hd+1.

3. Also, use a TreeMap which stores key value pair sorted on key.

4. Every time, we encounter a new horizontal distance or an existing horizontal distance put the node data for the horizontal distance as key.

For the first time it will add to the map, next time it will replace the value. This will make sure that the bottom most element for that horizontal distance is present in the map and if you see the tree from beneath that you will see that element.

A Java based implementation is below :

C++

```
// C++ Program to print Bottom View of Binary Tree
#include<bits/stdc++.h>
using namespace std;

// Tree node class
struct Node
{
    int data; //data of the node
    int hd; //horizontal distance of the node
    Node *left, *right; //left and right references

    // Constructor of tree node
    Node(int key)
    {
        data = key;
        hd = INT_MAX;
        left = right = NULL;
    }
};

// Method that prints the bottom view.
void bottomView(Node *root)
{
    if (root == NULL)
        return;

    // Initialize a variable 'hd' with 0
    // for the root element.
    int hd = 0;

    // TreeMap which stores key value pair
    // sorted on key value
    map<int, int> m;

    // Queue to store tree nodes in level
    // order traversal
    queue<Node *> q;

    // Assign initialized horizontal distance
    // value to root node and add it to the queue.
    root->hd = hd;
    q.push(root);

    // Loop until the queue is empty (standard
    // level order loop)
    while (!q.empty())
    {
        Node *temp = q.front();
        q.pop();

        // Extract the horizontal distance value
        // from the dequeued tree node.
        m[hd] = temp->data;
        if (temp->left)
            q.push(temp->left);
        if (temp->right)
            q.push(temp->right);
    }
}

// Function to print the bottom view
void printBottomView(Node *root)
{
    map<int, int> m;
    bottomView(root);
    for (auto i : m)
        cout << i.second << " ";
}
```

```

hd = temp->hd;

// Put the dequeued tree node to TreeMap
// having key as horizontal distance. Every
// time we find a node having same horizontal
// distance we need to replace the data in
// the map.
m[hd] = temp->data;

// If the dequeued node has a left child add
// it to the queue with a horizontal distance hd-1.
if (temp->left != NULL)
{
    temp->left->hd = hd-1;
    q.push(temp->left);
}

// If the dequeued node has a left child add
// it to the queue with a horizontal distance
// hd+1.
if (temp->right != NULL)
{
    temp->right->hd = hd+1;
    q.push(temp->right);
}

// Traverse the map elements using the iterator.
for (auto i = m.begin(); i != m.end(); ++i)
    cout << i->second << " ";
}

// Driver Code
int main()
{
    Node *root = new Node(20);
    root->left = new Node(8);
    root->right = new Node(22);
    root->left->left = new Node(5);
    root->left->right = new Node(3);
    root->right->left = new Node(4);
    root->right->right = new Node(25);
    root->left->right->left = new Node(10);
    root->left->right->right = new Node(14);
    cout << "Bottom view of the given binary tree :\n";
    bottomView(root);
    return 0;
}

```

Java

```

// Java Program to print Bottom View of Binary Tree
import java.util.*;
import java.util.Map.Entry;

// Tree node class
class Node
{
    int data; //data of the node
    int hd; //horizontal distance of the node
    Node left, right; //left and right references

    // Constructor of tree node
    public Node(int key)
    {
        data = key;
        hd = Integer.MAX_VALUE;
        left = right = null;
    }
}

//Tree class
class Tree

```

```
{
    Node root; //root node of tree

    // Default constructor
    public Tree() {}

    // Parameterized tree constructor
    public Tree(Node node)
    {
        root = node;
    }

    // Method that prints the bottom view.
    public void bottomView()
    {
        if (root == null)
            return;

        // Initialize a variable 'hd' with 0 for the root element.
        int hd = 0;

        // TreeMap which stores key value pair sorted on key value
        Map<Integer, Integer> map = new TreeMap<>();

        // Queue to store tree nodes in level order traversal
        Queue<Node> queue = new LinkedList<Node>();

        // Assign initialized horizontal distance value to root
        // node and add it to the queue.
        root.hd = hd;
        queue.add(root);

        // Loop until the queue is empty (standard level order loop)
        while (!queue.isEmpty())
        {
            Node temp = queue.remove();

            // Extract the horizontal distance value from the
            // dequeued tree node.
            hd = temp.hd;

            // Put the dequeued tree node to TreeMap having key
            // as horizontal distance. Every time we find a node
            // having same horizontal distance we need to replace
            // the data in the map.
            map.put(hd, temp.data);

            // If the dequeued node has a left child add it to the
            // queue with a horizontal distance hd-1.
            if (temp.left != null)
            {
                temp.left.hd = hd-1;
                queue.add(temp.left);
            }
            // If the dequeued node has a left child add it to the
            // queue with a horizontal distance hd+1.
            if (temp.right != null)
            {
                temp.right.hd = hd+1;
                queue.add(temp.right);
            }
        }

        // Extract the entries of map into a set to traverse
        // an iterator over that.
        Set<Entry<Integer, Integer>> set = map.entrySet();

        // Make an iterator
        Iterator<Entry<Integer, Integer>> iterator = set.iterator();

        // Traverse the map elements using the iterator.
        while (iterator.hasNext())
        {
            Map.Entry<Integer, Integer> me = iterator.next();
            System.out.print(me.getValue()+" ");
        }
    }
}
```

```

        }
    }

// Main driver class
public class BottomView
{
    public static void main(String[] args)
    {
        Node root = new Node(20);
        root.left = new Node(8);
        root.right = new Node(22);
        root.left.left = new Node(5);
        root.left.right = new Node(3);
        root.right.left = new Node(4);
        root.right.right = new Node(25);
        root.left.right.left = new Node(10);
        root.left.right.right = new Node(14);
        Tree tree = new Tree(root);
        System.out.println("Bottom view of the given binary tree:");
        tree.bottomView();
    }
}

```

Output:

Bottom view of the given binary tree:
5 10 4 14 25

Exercise: Extend the above solution to print all bottommost nodes at a horizontal distance if there are multiple bottommost nodes. For the above second example, the output should be 5 10 3 4 14 25.

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Trees

Print Nodes in Top View of Binary Tree

Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. Given a binary tree, print the top view of it. The output nodes can be printed in any order. Expected time complexity is O(n)

A node x is there in output if x is the topmost node at its horizontal distance. Horizontal distance of left child of a node x is equal to horizontal distance of x minus 1, and that of right child is horizontal distance of x plus 1.

```

1
/ \
2   3
/ \ / \
4 5 6 7
Top view of the above binary tree is
4 2 1 3 7

1
/ \
2   3
 \
4
 \
5
 \
6
Top view of the above binary tree is
2 1 3 6

```

We strongly recommend to minimize your browser and try this yourself first.

The idea is to do something similar to [vertical Order Traversal](#). Like vertical Order Traversal, we need to nodes of same horizontal distance together. We do a level order traversal so that the topmost node at a horizontal node is visited before any other node of same horizontal distance below it. Hashing is used to check if a node at given horizontal distance is seen or not.

```

// Java program to print top view of Binary tree
import java.util.*;

// Class for a tree node
class TreeNode
{
    // Members
    int key;
    TreeNode left, right;

    // Constructor
    public TreeNode(int key)
    {
        this.key = key;
        left = right = null;
    }
}

// A class to represent a queue item. The queue is used to do Level
// order traversal. Every Queue item contains node and horizontal
// distance of node from root
class QItem
{
    TreeNode node;
    int hd;
    public QItem(TreeNode n, int h)
    {
        node = n;
        hd = h;
    }
}

// Class for a Binary Tree
class Tree
{
    TreeNode root;

    // Constructors
    public Tree() { root = null; }
    public Tree(TreeNode n) { root = n; }

    // This method prints nodes in top view of binary tree
    public void printTopView()
    {
        // base case
        if (root == null) { return; }

        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();

        // Create a queue and add root to it
        Queue<QItem> Q = new LinkedList<QItem>();
        Q.add(new QItem(root, 0)); // Horizontal distance of root is 0

        // Standard BFS or level order traversal loop
        while (!Q.isEmpty())
        {
            // Remove the front item and get its details
            QItem qi = Q.remove();
            int hd = qi.hd;
            TreeNode n = qi.node;

            // If this is the first node at its horizontal distance,
            // then this node is in top view
            if (!set.contains(hd))
            {
                set.add(hd);
                System.out.print(n.key + " ");
            }

            // Enqueue left and right children of current node
            if (n.left != null)
                Q.add(new QItem(n.left, hd-1));
            if (n.right != null)
                Q.add(new QItem(n.right, hd+1));
        }
    }
}

```

```

        }
    }

// Driver class to test above methods
public class Main
{
    public static void main(String[] args)
    {
        /* Create following Binary Tree
           1
         / \
        2   3
         \
        4
         \
        5
         \
        6*/
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.right = new TreeNode(4);
        root.left.right.right = new TreeNode(5);
        root.left.right.right.right = new TreeNode(6);
        Tree t = new Tree(root);
        System.out.println("Following are nodes in top view of Binary Tree");
        t.printTopView();
    }
}

```

Output:

```

Following are nodes in top view of Binary Tree
1 2 3 6

```

Time Complexity of the above implementation is $O(n)$ where n is number of nodes in given binary tree. The assumption here is that `add()` and `contains()` methods of `HashSet` work in $O(1)$ time.

This article is contributed by **Rohan**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Trees

Remove nodes on root to leaf paths of length

Given a Binary Tree and a number k , remove all nodes that lie only on root to leaf path(s) of length smaller than k . If a node X lies on multiple root-to-leaf paths and if any of the paths has path length $\geq k$, then X is not deleted from Binary Tree. In other words a node is deleted if all paths going through it have lengths smaller than k .

Consider the following example Binary Tree

```

        1
       / \
      2   3
     / \   \
    4   5   6
   /       /
  7       8

```

Input: Root of above Binary Tree
 $k = 4$

Output: The tree should be changed to following

```

        1
       / \
      2   3
     /   \
    4     6
   /   /
  7   8

```

- There are 3 paths
- 1->2->4->7 path length = 4
 - 1->2->5 path length = 3
 - 1->3->6->8 path length = 4

There is only one path " 1->2->5 " of length smaller than 4.

The node 5 is the only node that lies only on this path, so node 5 is removed.

Nodes 2 and 1 are not removed as they are parts of other paths of length 4 as well.

If k is 5 or greater than 5, then whole tree is deleted.

If k is 3 or less than 3, then nothing is deleted.

We strongly recommend to minimize your browser and try this yourself first

The idea here is to use post order traversal of the tree. Before removing a node we need to check that all the children of that node in the shorter path are already removed.

There are 2 cases:

- This node becomes a leaf node in which case it needs to be deleted.
- This node has other child on a path with path length $\geq k$. In that case it needs not to be deleted.

The implementation of above approach is as below :

C/C++

```
// C++ program to remove nodes on root to leaf paths of length < K
#include<iostream>
using namespace std;

struct Node
{
    int data;
    Node *left, *right;
};

//New node of a tree
Node *newNode(int data)
{
    Node *node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Utility method that actually removes the nodes which are not
// on the pathLen  $\geq k$ . This method can change the root as well.
Node *removeShortPathNodesUtil(Node *root, int level, int k)
{
    //Base condition
    if (root == NULL)
        return NULL;

    // Traverse the tree in postorder fashion so that if a leaf
    // node path length is shorter than k, then that node and
    // all of its descendants till the node which are not
    // on some other path are removed.
    root->left = removeShortPathNodesUtil(root->left, level + 1, k);
    root->right = removeShortPathNodesUtil(root->right, level + 1, k);

    // If root is a leaf node and it's level is less than k then
    // remove this node.
    // This goes up and check for the ancestor nodes also for the
    // same condition till it finds a node which is a part of other
    // path(s) too.
    if (root->left == NULL && root->right == NULL && level < k)
    {
        delete root;
        return NULL;
    }

    // Return root;
    return root;
}
```

```

}

// Method which calls the utility method to remove the short path
// nodes.
Node *removeShortPathNodes(Node *root, int k)
{
    int pathLen = 0;
    return removeShortPathNodesUtil(root, 1, k);
}

//Method to print the tree in inorder fashion.
void printInorder(Node *root)
{
    if (root)
    {
        printInorder(root->left);
        cout << root->data << " ";
        printInorder(root->right);
    }
}

// Driver method.
int main()
{
    int k = 4;
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->left->left = newNode(7);
    root->right->right = newNode(6);
    root->right->right->left = newNode(8);
    cout << "Inorder Traversal of Original tree" << endl;
    printInorder(root);
    cout << endl;
    cout << "Inorder Traversal of Modified tree" << endl;
    Node *res = removeShortPathNodes(root, k);
    printInorder(res);
    return 0;
}

```

Java

```

// Java program to remove nodes on root to leaf paths of length < k

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // Utility method that actually removes the nodes which are not
    // on the pathLen >= k. This method can change the root as well.
    Node removeShortPathNodesUtil(Node node, int level, int k)
    {
        //Base condition
        if (node == null)
            return null;

        // Traverse the tree in postorder fashion so that if a leaf

```

```

// node path length is shorter than k, then that node and
// all of its descendants till the node which are not
// on some other path are removed.
node.left = removeShortPathNodesUtil(node.left, level + 1, k);
node.right = removeShortPathNodesUtil(node.right, level + 1, k);

// If root is a leaf node and it's level is less than k then
// remove this node.
// This goes up and check for the ancestor nodes also for the
// same condition till it finds a node which is a part of other
// path(s) too.
if (node.left == null && node.right == null && level < k)
    return null;

// Return root;
return node;
}

// Method which calls the utility method to remove the short path
// nodes.
Node removeShortPathNodes(Node node, int k)
{
    int pathLen = 0;
    return removeShortPathNodesUtil(node, 1, k);
}

//Method to print the tree in inorder fashion.
void printInorder(Node node)
{
    if (node != null)
    {
        printInorder(node.left);
        System.out.print(node.data + " ");
        printInorder(node.right);
    }
}

// Driver program to test for samples
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    int k = 4;
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.left.left.left = new Node(7);
    tree.root.right.right = new Node(6);
    tree.root.right.right.left = new Node(8);
    System.out.println("The inorder traversal of original tree is :");
    tree.printInorder(tree.root);
    Node res = tree.removeShortPathNodes(tree.root, k);
    System.out.println("");
    System.out.println("The inorder traversal of modified tree is :");
    tree.printInorder(res);
}
}

// This code has been contributed by Mayank Jaiswal

```

Output:

```

Inorder Traversal of Original tree
7 4 2 5 1 3 8 6
Inorder Traversal of Modified tree
7 4 2 1 3 8 6

```

Time complexity of the above solution is O(n) where n is number of nodes in given Binary Tree.

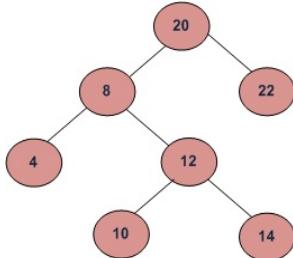
This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Lowest Common Ancestor in a Binary Search Tree.

Given values of two nodes in a Binary Search Tree, write a c program to find the Lowest Common Ancestor (LCA). You may assume that both the values exist in the tree.

The function prototype should be as follows:

```
struct node *lca(node* root, int n1, int n2)
n1 and n2 are two given values in the tree with given root.
```



For example, consider the BST in diagram, LCA of 10 and 14 is 12 and LCA of 8 and 14 is 8.

Following is definition of LCA from [Wikipedia](#):

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source [Wiki](#))

Solutions:

If we are given a BST where every node has **parent pointer**, then LCA can be easily determined by traversing up using parent pointer and printing the first intersecting node.

We can solve this problem using BST properties. We can **recursively traverse** the BST from root. The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2, i.e., $n1 < n < n2$ or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that $n1 < n2$). So just recursively traverse the BST in, if node's value is greater than both n1 and n2 then our LCA lies in left side of the node, if it's is smaller than both n1 and n2, then LCA lies on right side. Otherwise root is LCA (assuming that both n1 and n2 are present in BST)

C

```
// A recursive C program to find LCA of two nodes n1 and n2.
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node* left, *right;
};

/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    if (root == NULL) return NULL;

    // If both n1 and n2 are smaller than root, then LCA lies in left
    if (root->data > n1 && root->data > n2)
        return lca(root->left, n1, n2);

    // If both n1 and n2 are greater than root, then LCA lies in right
    if (root->data < n1 && root->data < n2)
        return lca(root->right, n1, n2);

    return root;
}
```

```

    return lca(root->right, n1, n2);

    return root;
}

/* Helper function that allocates a new node with the given data.*/
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

/* Driver program to test lca() */
int main()
{
    // Let us construct the BST shown in the above figure
    struct node *root      = newNode(20);
    root->left           = newNode(8);
    root->right          = newNode(22);
    root->left->left    = newNode(4);
    root->left->right   = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);

    int n1 = 10, n2 = 14;
    struct node *t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    n1 = 14, n2 = 8;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    n1 = 10, n2 = 22;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    getchar();
    return 0;
}

```

Java

```

// Recursive Java program to print lca of two nodes

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Function to find LCA of n1 and n2. The function assumes that both
       n1 and n2 are present in BST */
    Node lca(Node node, int n1, int n2)
    {
        if (node == null)
            return null;

        // If both n1 and n2 are smaller than root, then LCA lies in left
        if (node.data > n1 && node.data > n2)
            return lca(node.left, n1, n2);
    }
}

```

```

// If both n1 and n2 are greater than root, then LCA lies in right
if (node.data < n1 && node.data < n2)
    return lca(node.right, n1, n2);

return node;
}

/* Driver program to test lca() */
public static void main(String args[])
{
    // Let us construct the BST shown in the above figure
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(20);
    tree.root.left = new Node(8);
    tree.root.right = new Node(22);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(12);
    tree.root.left.right.left = new Node(10);
    tree.root.left.right.right = new Node(14);

    int n1 = 10, n2 = 14;
    Node t = tree.lca(tree.root, n1, n2);
    System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

    n1 = 14;
    n2 = 8;
    t = tree.lca(tree.root, n1, n2);
    System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

    n1 = 10;
    n2 = 22;
    t = tree.lca(tree.root, n1, n2);
    System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# A recursive python program to find LCA of two nodes
# n1 and n2

# A Binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Function to find LCA of n1 and n2. The function assumes
    # that both n1 and n2 are present in BST
    def lca(root, n1, n2):

        # Base Case
        if root is None:
            return None

        # If both n1 and n2 are smaller than root, then LCA
        # lies in left
        if(root.data > n1 and root.data > n2):
            return lca(root.left, n1, n2)

        # If both n1 and n2 are greater than root, then LCA
        # lies in right
        if(root.data < n1 and root.data < n2):
            return lca(root.right, n1, n2)

        # If one node is smaller and other is greater than root
        # then root is LCA
        return root

```

```

return root

# Driver program to test above function

# Let us construct the BST shown in the figure
root = Node(20)
root.left = Node(8)
root.right = Node(22)
root.left.left = Node(4)
root.left.right = Node(12)
root.left.right.left = Node(10)
root.left.right.right = Node(14)

n1 = 10 ; n2 = 14
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

n1 = 14 ; n2 = 8
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

n1 = 10 ; n2 = 22
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20

```

Time complexity of above solution is $O(h)$ where h is height of tree. Also, the above solution requires $O(h)$ extra space in function call stack for recursive function calls. We can avoid extra space using **iterative solution**.

```

/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    while (root != NULL)
    {
        // If both n1 and n2 are smaller than root, then LCA lies in left
        if (root->data > n1 && root->data > n2)
            root = root->left;

        // If both n1 and n2 are greater than root, then LCA lies in right
        else if (root->data < n1 && root->data < n2)
            root = root->right;

        else break;
    }
    return root;
}

```

See [this](#) for complete program.

You may like to see below articles as well :

[Lowest Common Ancestor in a Binary Tree](#)

[LCA using Parent Pointer](#)

[Find LCA in Binary Tree using RMQ](#)

Exercise

The above functions assume that $n1$ and $n2$ both are in BST. If $n1$ and $n2$ are not present, then they may return incorrect result. Extend the above solutions to return NULL if $n1$ or $n2$ or both not present in BST.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Check if a binary tree is subtree of another binary tree | Set 2

Given two binary trees, check if the first tree is subtree of the second one. A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T.

The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

For example, in the following case, Tree1 is a subtree of Tree2.

```
Tree1
x
/ \
a   b
\
c

Tree2
z
/ \
x   e
/ \  \
a   b   k
\
c
```

We have discussed a $O(n^2)$ solution for this problem. In this post a $O(n)$ solution is discussed. The idea is based on the fact that **inorder and preorder/postorder uniquely identify a binary tree**. Tree S is a subtree of T if both inorder and preorder traversals of S are substrings of inorder and preorder traversals of T respectively.

Following are detailed steps.

- 1) Find inorder and preorder traversals of T, store them in two auxiliary arrays $inT[]$ and $preT[]$.
- 2) Find inorder and preorder traversals of S, store them in two auxiliary arrays $inS[]$ and $preS[]$.
- 3) If $inS[]$ is a subarray of $inT[]$ and $preS[]$ is a subarray of $preT[]$, then S is a subtree of T. Else not.

We can also use postorder traversal in place of preorder in the above algorithm.

Let us consider the above example

Inorder and Preorder traversals of the big tree are.

$inT[] = \{a, c, x, b, z, e, k\}$
 $preT[] = \{z, x, a, c, b, e, k\}$

Inorder and Preorder traversals of small tree are

$inS[] = \{a, c, x, b\}$
 $preS[] = \{x, a, c, b\}$

We can easily figure out that $inS[]$ is a subarray of $inT[]$ and $preS[]$ is a subarray of $preT[]$.

EDIT

The above algorithm doesn't work for cases where a tree is present in another tree, but not as a subtree. Consider the following example.

```
Tree1
x
/ \
a   b
/
c

Tree2
x
/ \
a   b
/   \
c   d
```

Inorder and Preorder traversals of the big tree or Tree2 are.

Inorder and Preorder traversals of small tree or Tree1 are

The Tree2 is not a subtree of Tree1, but inS[] and preS[] are subarrays of inT[] and preT[] respectively.

The above algorithm can be extended to handle such cases by adding a special character whenever we encounter NULL in inorder and preorder traversals. Thanks to Shivam Goel for suggesting this extension.

Following is the implementation of above algorithm.

C

```
#include <iostream>
#include <cstring>
using namespace std;
#define MAX 100

// Structure of a tree node
struct Node
{
    char key;
    struct Node *left, *right;
};

// A utility function to create a new BST node
Node *newNode(char item)
{
    Node *temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to store inorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storeInorder(Node *root, char arr[], int &i)
{
    if (root == NULL)
    {
        arr[i++] = '$';
        return;
    }
    storeInorder(root->left, arr, i);
    arr[i++] = root->key;
    storeInorder(root->right, arr, i);
}

// A utility function to store preorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storePreOrder(Node *root, char arr[], int &i)
{
    if (root == NULL)
    {
        arr[i++] = '$';
        return;
    }
    arr[i++] = root->key;
    storePreOrder(root->left, arr, i);
    storePreOrder(root->right, arr, i);
}

/* This function returns true if S is a subtree of T, otherwise false */
bool isSubtree(Node *T, Node *S)
{
    /* base cases */
    if (S == NULL) return true;
    if (T == NULL) return false;

    // Store Inorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
```

```

int m = 0, n = 0;
char inT[MAX], inS[MAX];
storeInorder(T, inT, m);
storeInorder(S, inS, n);
inT[m] = '\0', inS[n] = '\0';

// If inS[] is not a substring of preS[], return false
if (strstr(inT, inS) == NULL)
    return false;

// Store Preorder traversals of T and S in inT[0..m-1]
// and inS[0..n-1] respectively
m = 0, n = 0;
char preT[MAX], preS[MAX];
storePreOrder(T, preT, m);
storePreOrder(S, preS, n);
preT[m] = '\0'; preS[n] = '\0';

// If inS[] is not a substring of preS[], return false
// Else return true
return (strstr(preT, preS) != NULL);
}

// Driver program to test above function
int main()
{
    Node *T = newNode('a');
    T->left = newNode('b');
    T->right = newNode('d');
    T->left->left = newNode('c');
    T->right->right = newNode('e');

    Node *S = newNode('a');
    S->left = newNode('b');
    S->left->left = newNode('c');
    S->right = newNode('d');

    if (isSubtree(T, S))
        cout << "Yes: S is a subtree of T";
    else
        cout << "No: S is NOT a subtree of T";

    return 0;
}

```

Java

```

// Java program to check if binary tree is subtree of another binary tree
class Node {

    char data;
    Node left, right;

    Node(char item) {
        data = item;
        left = right = null;
    }
}

class Passing {

    int i;
    int m = 0;
    int n = 0;
}

class BinaryTree {

    static Node root;
    Passing p = new Passing();

    String strstr(String haystack, String needle) {

```

```

if (haystack == null || needle == null) {
    return null;
}
int hLength = haystack.length();
int nLength = needle.length();
if (hLength < nLength) {
    return null;
}
if (nLength == 0) {
    return haystack;
}
for (int i = 0; i <= hLength - nLength; i++) {
    if (haystack.charAt(i) == needle.charAt(0)) {
        int j = 0;
        for (; j < nLength; j++) {
            if (haystack.charAt(i + j) != needle.charAt(j)) {
                break;
            }
        }
        if (j == nLength) {
            return haystack.substring(i);
        }
    }
}
return null;
}

// A utility function to store inorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storeInorder(Node node, char arr[], Passing i) {
    if (node == null) {
        arr[i++] = '$';
        return;
    }
    storeInorder(node.left, arr, i);
    arr[i++] = node.data;
    storeInorder(node.right, arr, i);
}

// A utility function to store preorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storePreOrder(Node node, char arr[], Passing i) {
    if (node == null) {
        arr[i++] = '$';
        return;
    }
    arr[i++] = node.data;
    storePreOrder(node.left, arr, i);
    storePreOrder(node.right, arr, i);
}

/* This function returns true if S is a subtree of T, otherwise false */
boolean isSubtree(Node T, Node S) {
    /* base cases */
    if (S == null) {
        return true;
    }
    if (T == null) {
        return false;
    }

    // Store Inorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    char inT[] = new char[100];
    String op1 = String.valueOf(inT);
    char inS[] = new char[100];
    String op2 = String.valueOf(inS);
    storeInorder(T, inT, p);
    storeInorder(S, inS, p);
    inT[p.m] = '\0';
    inS[p.m] = '\0';

    // If inS[] is not a substring of preS[], return false
    if (strstr(op1, op2) != null) {
        return false;
    }
}

```

```

}

// Store Preorder traversals of T and S in inT[0..m-1]
// and inS[0..n-1] respectively
p.m = 0;
p.n = 0;
char preT[] = new char[100];
char preS[] = new char[100];
String op3 = String.valueOf(preT);
String op4 = String.valueOf(preS);
storePreOrder(T, preT, p);
storePreOrder(S, preS, p);
preT[p.m] = '\0';
preS[p.n] = '\0';

// If inS[] is not a substring of preS[], return false
// Else return true
return (strstr(op3, op4) != null);
}

//Driver program to test above functions
public static void main(String args[]) {
    BinaryTree tree = new BinaryTree();
    Node T = new Node('a');
    T.left = new Node('b');
    T.right = new Node('d');
    T.left.left = new Node('c');
    T.right.right = new Node('e');

    Node S = new Node('a');
    S.left = new Node('b');
    S.right = new Node('d');
    S.left.left = new Node('c');

    if (tree.isSubtree(T, S)) {
        System.out.println("Yes , S is a subtree of T");
    } else {
        System.out.println("No, S is not a subtree of T");
    }
}
}

// This code is contributed by Mayank Jaiswal

```

Output:

No: S is NOT a subtree of T

Time Complexity: Inorder and Preorder traversals of Binary Tree take O(n) time. The function [strstr\(\)](#) can also be implemented in O(n) time using [KMP string matching algorithm](#).

Auxiliary Space: O(n)

Thanks to **Ashwini Singh** for suggesting this method. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Trees

Reverse alternate levels of a perfect binary tree

Given a [Perfect Binary Tree](#), reverse the alternate level nodes of the binary tree.

Given tree:



```
h i j k l m n o
```

Modified tree:

```
      a
      /   \
     c     b
    / \   / \
   d   e   f   g
  / \ / \ / \ / \
o   n   m   l   k   j   i   h
```

We strongly recommend to minimize the browser and try this yourself first.

Method 1 (Simple)

A **simple solution** is to do following steps.

- 1) Access nodes level by level.
- 2) If current level is odd, then store nodes of this level in an array.
- 3) Reverse the array and store elements back in tree.

Method 2 (Using Two Traversals)

Another is to do two inorder traversals. Following are steps to be followed.

- 1) Traverse the given tree in inorder fashion and store all odd level nodes in an auxiliary array. For the above example given tree, contents of array become {h, i, b, j, k, l, m, c, n, o}

2) Reverse the array. The array now becomes {o, n, c, m, l, k, j, b, i, h}

3) Traverse the tree again inorder fashion. While traversing the tree, one by one take elements from array and store elements from array to every odd level traversed node.

For the above example, we traverse 'h' first in above array and replace 'h' with 'o'. Then we traverse 'i' and replace it with n.

Following is the implementation of the above algorithm.

C++

```
// C++ program to reverse alternate levels of a binary tree
#include<iostream>
#define MAX 100
using namespace std;

// A Binary Tree node
struct Node
{
    char data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree Node
struct Node *newNode(char item)
{
    struct Node *temp = new Node;
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to store nodes of alternate levels in an array
void storeAlternate(Node *root, char arr[], int *index, int l)
{
    // Base case
    if (root == NULL) return;

    // Store elements of left subtree
    storeAlternate(root->left, arr, index, l+1);

    // Store this node only if this is a odd level node
    if (l%2 != 0)
    {
        arr[*index] = root->data;
        (*index)++;
    }
}
```

```

// Store elements of right subtree
storeAlternate(root->right, arr, index, l+1);
}

// Function to modify Binary Tree (All odd level nodes are
// updated by taking elements from array in inorder fashion)
void modifyTree(Node *root, char arr[], int *index, int l)
{
    // Base case
    if (root == NULL) return;

    // Update nodes in left subtree
    modifyTree(root->left, arr, index, l+1);

    // Update this node only if this is an odd level node
    if (l%2 != 0)
    {
        root->data = arr[*index];
        (*index)++;
    }

    // Update nodes in right subtree
    modifyTree(root->right, arr, index, l+1);
}

// A utility function to reverse an array from index
// 0 to n-1
void reverse(char arr[], int n)
{
    int l = 0, r = n-1;
    while (l < r)
    {
        int temp = arr[l];
        arr[l] = arr[r];
        arr[r] = temp;
        l++; r--;
    }
}

// The main function to reverse alternate nodes of a binary tree
void reverseAlternate(struct Node *root)
{
    // Create an auxiliary array to store nodes of alternate levels
    char *arr = new char[MAX];
    int index = 0;

    // First store nodes of alternate levels
    storeAlternate(root, arr, &index, 0);

    // Reverse the array
    reverse(arr, index);

    // Update tree by taking elements from array
    index = 0;
    modifyTree(root, arr, &index, 0);
}

// A utility function to print inorder traversal of a
// binary tree
void printlnorder(struct Node *root)
{
    if (root == NULL) return;
    printlnorder(root->left);
    cout << root->data << " ";
    printlnorder(root->right);
}

// Driver Program to test above functions
int main()
{
    struct Node *root = newNode('a');
    root->left = newNode('b');
    root->right = newNode('c');
    root->left->left = newNode('d');
}

```

```

root->left->right = newNode('e');
root->right->left = newNode('f');
root->right->right = newNode('g');
root->left->left->left = newNode('h');
root->left->left->right = newNode('i');
root->left->right->left = newNode('j');
root->right->left->right = newNode('k');
root->right->left->left = newNode('l');
root->right->right->right = newNode('m');
root->right->right->left = newNode('n');
root->right->right->right = newNode('o');

cout << "Inorder Traversal of given tree\n";
printInorder(root);

reverseAlternate(root);

cout << "\n\nInorder Traversal of modified tree\n";
printInorder(root);

return 0;
}

```

Java

```

// Java program to reverse alternate levels of perfect binary tree
// A binary tree node
class Node {

    char data;
    Node left, right;

    Node(char item) {
        data = item;
        left = right = null;
    }
}

// class to access index value by reference
class Index {

    int index;
}

class BinaryTree {

    Node root;
    Index index_obj = new Index();

    // function to store alternate levels in a tree
    void storeAlternate(Node node, char arr[], Index index, int l) {
        // base case
        if (node == null) {
            return;
        }
        // store elements of left subtree
        storeAlternate(node.left, arr, index, l + 1);

        // store this node only if level is odd
        if (l % 2 != 0) {
            arr[index.index] = node.data;
            index.index++;
        }

        storeAlternate(node.right, arr, index, l + 1);
    }

    // Function to modify Binary Tree (All odd level nodes are
    // updated by taking elements from array in inorder fashion)
    void modifyTree(Node node, char arr[], Index index, int l) {

        // Base case

```

```

if (node == null) {
    return;
}

// Update nodes in left subtree
modifyTree(node.left, arr, index, l + 1);

// Update this node only if this is an odd level node
if (l % 2 != 0) {
    node.data = arr[index.index];
    (index.index)++;
}

// Update nodes in right subtree
modifyTree(node.right, arr, index, l + 1);
}

// A utility function to reverse an array from index
// 0 to n-1
void reverse(char arr[], int n) {
    int l = 0, r = n - 1;
    while (l < r) {
        char temp = arr[l];
        arr[l] = arr[r];
        arr[r] = temp;
        l++;
        r--;
    }
}

void reverseAlternate() {
    reverseAlternate(root);
}

// The main function to reverse alternate nodes of a binary tree
void reverseAlternate(Node node) {

    // Create an auxiliary array to store nodes of alternate levels
    char[] arr = new char[100];

    // First store nodes of alternate levels
    storeAlternate(node, arr, index_obj, 0);

    //index_obj.index = 0;

    // Reverse the array
    reverse(arr, index_obj.index);

    // Update tree by taking elements from array
    index_obj.index = 0;
    modifyTree(node, arr, index_obj, 0);
}

void printlnorder() {
    printlnorder(root);
}

// A utility function to print inorder traversal of a
// binary tree
void printlnorder(Node node) {
    if (node == null) {
        return;
    }
    printlnorder(node.left);
    System.out.print(node.data + " ");
    printlnorder(node.right);
}

// Driver program to test the above functions
public static void main(String args[]) {
    BinaryTree tree = new BinaryTree();
    tree.root = new Node('a');
    tree.root.left = new Node('b');
    tree.root.right = new Node('c');
    tree.root.left.left = new Node('d');
}

```

```

tree.root.left.right = new Node('e');
tree.root.right.left = new Node('f');
tree.root.right.right = new Node('g');
tree.root.left.left.left = new Node('h');
tree.root.left.left.right = new Node('i');
tree.root.left.right.left = new Node('j');
tree.root.left.right.right = new Node('k');
tree.root.right.left.left = new Node('l');
tree.root.right.left.right = new Node('m');
tree.root.right.right.left = new Node('n');
tree.root.right.right.right = new Node('o');
System.out.println("Inorder Traversal of given tree");
tree.printInorder();

tree.reverseAlternate();
System.out.println("");
System.out.println("");
System.out.println("Inorder Traversal of modified tree");
tree.printInorder();
}
}

// This code has been contributed by Mayank Jaiswal

```

Output:

```
Inorder Traversal of given tree
h d i b j e k a l f m c n g o
```

```
Inorder Traversal of modified tree
o d n c m e l a k f j b i g h
```

Time complexity of the above solution is O(n) as it does two inorder traversals of binary tree.

Method 3 (Using One Traversal)

```

// C++ program to reverse alternate levels of a tree
#include <bits/stdc++.h>
using namespace std;

struct Node
{
    char key;
    Node *left, *right;
};

void preorder(struct Node *root1, struct Node* root2, int lvl)
{
    // Base cases
    if (root1 == NULL || root2==NULL)
        return;

    // Swap subtrees if level is even
    if (lvl%2 == 0)
        swap(root1->key, root2->key);

    // Recur for left and right subtrees (Note : left of root1
    // is passed and right of root2 in first call and opposite
    // in second call.
    preorder(root1->left, root2->right, lvl+1);
    preorder(root1->right, root2->left, lvl+1);
}

// This function calls preorder() for left and right children
// of root
void reverseAlternate(struct Node *root)
{
    preorder(root->left, root->right, 0);
}

// Inorder traversal (used to print initial and
// modified trees)
void printInorder(struct Node *root)
{

```

```

if (root == NULL)
    return;
printInorder(root->left);
cout << root->key << " ";
printInorder(root->right);
}

// A utility function to create a new node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->left = temp->right = NULL;
    temp->key = key;
    return temp;
}

// Driver program to test above functions
int main()
{
    struct Node *root = newNode('a');
    root->left = newNode('b');
    root->right = newNode('c');
    root->left->left = newNode('d');
    root->left->right = newNode('e');
    root->right->left = newNode('f');
    root->right->right = newNode('g');
    root->left->left->left = newNode('h');
    root->left->left->right = newNode('i');
    root->left->right->left = newNode('j');
    root->left->right->right = newNode('k');
    root->right->left->left = newNode('l');
    root->right->left->right = newNode('m');
    root->right->right->left = newNode('n');
    root->right->right->right = newNode('o');

    cout << "Inorder Traversal of given tree\n";
    printInorder(root);

    reverseAlternate(root);

    cout << "\n\nInorder Traversal of modified tree\n";
    printInorder(root);
    return 0;
}

```

Output :

```
Inorder Traversal of given tree
h d i b j e k a l f m c n g o
```

```
Inorder Traversal of modified tree
o d n c m e l a k f j b i g h
```

Thanks Soumyajit Bhattacharyay for suggesting above solution.

This article is contributed by **Kripal Gaurav**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Trees
Reverse

Modular Exponentiation (Power in Modular Arithmetic)

Given three numbers x, y and p, compute $(x^y) \% p$.

Examples

Input: x = 2, y = 3, p = 5

Output: 3

Explanation: $2^3 \% 5 = 8 \% 5 = 3$.

```

Input: x = 2, y = 5, p = 13
Output: 6
Explanation: 2^5 % 13 = 32 % 13 = 6.

```

We have discussed **recursive** and **iterative** solutions for power.

Below is discussed iterative solution.

```

/* Iterative Function to calculate (x^y) in O(log y) */
int power(int x, unsigned int y)
{
    int res = 1; // Initialize result

    while (y > 0)
    {
        // If y is odd, multiply x with result
        if (y & 1)
            res = res*x;

        // n must be even now
        y = y>>1; // y = y/2
        x = x*x; // Change x to x^2
    }
    return res;
}

```

The problem with above solutions is, overflow may occur for large value of n or x. Therefore, power is generally evaluated under modulo of a large number.

Below is the fundamental modular property that is used for efficiently computing power under modular arithmetic.

```

(a mod p) (b mod p) ≡ (ab) mod p

or equivalently

( (a mod p) (b mod p) ) mod p = (ab) mod p

For example a = 50, b = 100, p = 13
50 mod 13 = 11
100 mod 13 = 9

11*9 ≡ 1500 mod 13
or
11*9 mod 13 = 1500 mod 13

```

Below is C implementation based on above property.

```

// Iterative C program to compute modular power
#include <stdio.h>

/* Iterative Function to calculate (x^y)%p in O(log y) */
int power(int x, unsigned int y, int p)
{
    int res = 1; // Initialize result

    x = x % p; // Update x if it is more than or
               // equal to p

    while (y > 0)
    {
        // If y is odd, multiply x with result
        if (y & 1)
            res = (res*x) % p;

        // y must be even now
        y = y>>1; // y = y/2
        x = (x*x) % p;
    }
    return res;
}

// Driver program to test above functions
int main()
{
    int x = 2;

```

```

int y = 5;
int p = 13;
printf("Power is %u", power(x, y, p));
return 0;
}

```

Output:

Power is 6

Time Complexity of above solution is O(Log y).

This article is contributed by **Shivam Agrawal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Mathematical
large-numbers
modular-arithmetic

Modular multiplicative inverse

Given two integers 'a' and 'm', find modular multiplicative inverse of 'a' under modulo 'm'.

The modular multiplicative inverse is an integer 'x' such that.

$a \cdot x \equiv 1 \pmod{m}$

The value of x should be in {0, 1, 2, ... m-1}, i.e., in the ring of integer modulo m.

The multiplicative inverse of "a modulo m" exists if and only if a and m are relatively prime (i.e., if $\gcd(a, m) = 1$).

Examples:

```

Input: a = 3, m = 11
Output: 4
Since (4*3) mod 11 = 1, 4 is modulo inverse of 3
One might think, 15 also as a valid output as "(15*3) mod 11"
is also 1, but 15 is not in ring {0, 1, 2, ... 10}, so not
valid.

Input: a = 10, m = 17
Output: 12
Since (10*12) mod 17 = 1, 12 is modulo inverse of 3

```

We strongly recommend you to minimize your browser and try this yourself first.

Method 1 (Naive)

A Naive method is to try all numbers from 1 to m. For every number x, check if $(a \cdot x) \% m$ is 1. Below is C++ implementation of this method.

```

// C++ program to find modular inverse of a under modulo m
#include<iostream>
using namespace std;

// A naive method to find modular multiplicative inverse of
// 'a' under modulo 'm'
int modInverse(int a, int m)
{
    a = a%m;
    for (int x=1; x<m; x++)
        if ((a*x) % m == 1)
            return x;
}

// Driver Program
int main()
{
    int a = 3, m = 11;
    cout << modInverse(a, m);
    return 0;
}

```

Output:

Time Complexity of this method is O(m).

Method 2 (Works when m and a are coprime)

The idea is to use **Extended Euclidean algorithms** that takes two integers 'a' and 'b', finds their gcd and also find 'x' and 'y' such that

$$ax + by = \text{gcd}(a, b)$$

To find multiplicative inverse of 'a' under 'm', we put $b = m$ in above formula. Since we know that a and m are relatively prime, we can put value of gcd as 1.

$$ax + my = 1$$

If we take modulo m on both sides, we get

$$ax + my \equiv 1 \pmod{m}$$

We can remove the second term on left side as ' $my \pmod{m}$ ' would always be 0 for an integer y .

$$ax \equiv 1 \pmod{m}$$

So the ' x ' that we can find using **Extended Euclid Algorithm** is multiplicative inverse of ' a '.

Below is C++ implementation of above algorithm.

```
// C++ program to find multiplicative modulo inverse using
// Extended Euclid algorithm.
#include<iostream>
using namespace std;

// C function for extended Euclidean Algorithm
int gcdExtended(int a, int b, int *x, int *y);

// Function to find modulo inverse of a
void modInverse(int a, int m)
{
    int x, y;
    int g = gcdExtended(a, m, &x, &y);
    if (g != 1)
        cout << "Inverse doesn't exist";
    else
    {
        // m is added to handle negative x
        int res = (x%m + m) % m;
        cout << "Modular multiplicative inverse is " << res;
    }
}

// C function for extended Euclidean Algorithm
int gcdExtended(int a, int b, int *x, int *y)
{
    // Base Case
    if (a == 0)
    {
        *x = 0, *y = 1;
        return b;
    }

    int x1, y1; // To store results of recursive call
    int gcd = gcdExtended(b%a, a, &x1, &y1);

    // Update x and y using results of recursive
    // call
    *x = y1 - (b/a) * x1;
    *y = x1;

    return gcd;
}

// Driver Program
int main()
{
    int a = 3, m = 11;
    modInverse(a, m);
    return 0;
}
```

```
}
```

Output:

```
Modular multiplicative inverse is 4
```

Iterative Implementation:

```
// Iterative C++ program to find modular inverse using
// extended Euclid algorithm
#include <stdio.h>

// Returns modulo inverse of a with respect to m using
// extended Euclid Algorithm
// Assumption: a and m are coprimes, i.e., gcd(a, m) = 1
int modInverse(int a, int m)
{
    int m0 = m, t, q;
    int x0 = 0, x1 = 1;

    if (m == 1)
        return 0;

    while (a > 1)
    {
        // q is quotient
        q = a / m;

        t = m;
        m = a % m, a = t;

        t = x0;
        x0 = x1 - q * x0;

        x1 = t;
    }

    // Make x1 positive
    if (x1 < 0)
        x1 += m0;

    return x1;
}

// Driver program to test above function
int main()
{
    int a = 3, m = 11;

    printf("Modular multiplicative inverse is %d\n",
          modInverse(a, m));
    return 0;
}
```

Output:

```
Modular multiplicative inverse is 4
```

Time Complexity of this method is O(Log m)

Method 3 (Works when m is prime)

If we know m is prime, then we can also use **Fermat's little theorem** to find the inverse.

$$a^{m-1} \equiv 1 \pmod{m}$$

If we multiply both sides with a^{-1} , we get

$$a^{-1} \equiv a^{m-2} \pmod{m}$$

Below is C++ implementation of above idea.

```

// C++ program to find modular inverse of a under modulo m
// This program works only if m is prime.
#include<iostream>
using namespace std;

// To find GCD of a and b
int gcd(int a, int b);

// To compute x raised to power y under modulo m
int power(int x, unsigned int y, unsigned int m);

// Function to find modular inverse of a under modulo m
// Assumption: m is prime
void modInverse(int a, int m)
{
    int g = gcd(a, m);
    if (g != 1)
        cout << "Inverse doesn't exist";
    else
    {
        // If a and m are relatively prime, then modulo inverse
        // is a^(m-2) mode m
        cout << "Modular multiplicative inverse is "
             << power(a, m-2, m);
    }
}

// To compute x^y under modulo m
int power(int x, unsigned int y, unsigned int m)
{
    if (y == 0)
        return 1;
    int p = power(x, y/2, m) % m;
    p = (p * p) % m;

    return (y%2 == 0)? p : (x * p) % m;
}

// Function to return gcd of a and b
int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b%a, a);
}

// Driver Program
int main()
{
    int a = 3, m = 11;
    modInverse(a, m);
    return 0;
}

```

Output:

Modular multiplicative inverse is 4

Time Complexity of this method is O(Log m)

We have discussed three methods to find multiplicative inverse modulo m.

- 1) Naive Method, O(m)
- 2) Extended Euler's GCD algorithm, O(Log m) [Works when a and m are coprime]
- 3) Fermat's Little theorem, O(Log m) [Works when 'm' is prime]

Applications:

Computation of the modular multiplicative inverse is an essential step in RSA public-key encryption method.

References:

https://en.wikipedia.org/wiki/Modular_multiplicative_inverse
http://e-maxx.ru/algo/reverse_element

This article is contributed by **Ankur**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Primality Test | Set 2 (Fermat Method)

Given a number n, check if it is prime or not. We have introduced and discussed School method for primality testing in Set 1.

Primality Test | Set 1 (Introduction and School Method)

In this post, Fermat's method is discussed. This method is a probabilistic method and is based on below Fermat's Little Theorem.

Fermat's Little Theorem:

If n is a prime number, then for every a, $a^{n-1} \equiv 1 \pmod{n}$

OR

$a^{n-1} \% n = 1$

Example: Since 5 is prime, $2^4 \equiv 1 \pmod{5}$ [or $2^4 \% 5 = 1$],

$3^4 \equiv 1 \pmod{5}$ and $4^4 \equiv 1 \pmod{5}$

Since 7 is prime, $2^6 \equiv 1 \pmod{7}$,

$3^6 \equiv 1 \pmod{7}$, $4^6 \equiv 1 \pmod{7}$

$5^6 \equiv 1 \pmod{7}$ and $6^6 \equiv 1 \pmod{7}$

Refer [this](#) for different proofs.

If a given number is prime, then this method always returns true. If given number is composite (or non-prime), then it may return true or false, but the probability of producing incorrect result for composite is low and can be reduced by doing more iterations.

Below is algorithm:

```
// Higher value of k indicates probability of correct
// results for composite inputs become higher. For prime
// inputs, result is always correct
1) Repeat following k times:
   a) Pick a randomly in the range [2, n - 2]
   b) If  $a^{n-1} \not\equiv 1 \pmod{n}$ , then return false
2) Return true [probably prime].
```

Below is C++ implementation of above algorithm. The code uses power function from [Modular Exponentiation](#)

```
// C++ program to find the smallest twin in given range
#include <bits/stdc++.h>
using namespace std;

/* Iterative Function to calculate (a^n)%p in O(logn) */
int power(int a, unsigned int n, int p)
{
    int res = 1; // Initialize result
    a = a % p; // Update 'a' if 'a' >= p

    while (n > 0)
    {
        // If n is odd, multiply 'a' with result
        if (n & 1)
            res = (res*a) % p;

        // n must be even now
        n = n>>1; // n = n/2
        a = (a*a) % p;
    }
    return res;
}

// If n is prime, then always returns true, If n is
// composite than returns false with high probability
// Higher value of k increases probability of correct
// result.
bool isPrime(unsigned int n, int k)
{
    // Corner cases
    if (n <= 1 || n == 4) return false;
```

```

if (n <= 3) return true;

// Try k times
while (k>0)
{
    // Pick a random number in [2..n-2]
    // Above corner cases make sure that n > 4
    int a = 2 + rand()% (n-4);

    // Fermat's little theorem
    if (power(a, n-1, n) != 1)
        return false;

    k--;
}

return true;
}

// Driver Program to test above function
int main()
{
    int k = 3;
    isPrime(11, k)? cout << " true\n": cout << " false\n";
    isPrime(15, k)? cout << " true\n": cout << " false\n";
    return 0;
}

```

Output:

```

true
false

```

Time complexity of this solution is $O(k \log n)$. Note that power function takes $O(\log n)$ time.

Note that the above method may fail even if we increase number of iterations (higher k). There exist sum composite numbers with the property that for every $a^{n-1} \equiv 1 \pmod{n}$. Such numbers are called **Carmichael numbers**. Fermat's primality test is often used if a rapid method is needed for filtering, for example in key generation phase of the RSA public key cryptographic algorithm.

We will soon be discussing more methods for Primality Testing.

References:

https://en.wikipedia.org/wiki/Fermat_primality_test
https://en.wikipedia.org/wiki/Prime_number
<http://www.cse.iitk.ac.in/users/manindra/presentations/FLTBasedTests.pdf>
https://en.wikipedia.org/wiki/Primality_test

This article is contributed by **Ajay**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
Randomized
modular-arithmetic
prime-number

Euler's Totient Function

Euler's Totient function $\Phi(n)$ for an input n is count of numbers in $\{1, 2, 3, \dots, n\}$ that are relatively prime to n , i.e., the numbers whose GCD (Greatest Common Divisor) with n is 1.

Examples:

```

Φ(1) = 1
gcd(1, 1) is 1

Φ(2) = 1
gcd(1, 2) is 1, but gcd(2, 2) is 2.

Φ(3) = 2
gcd(1, 3) is 1 and gcd(2, 3) is 1

```

$\Phi(4) = 2$

gcd(1, 4) is 1 and gcd(3, 4) is 1

$\Phi(5) = 4$

gcd(1, 5) is 1, gcd(2, 5) is 1,

gcd(3, 5) is 1 and gcd(4, 5) is 1

$\Phi(6) = 2$

gcd(1, 6) is 1 and gcd(5, 6) is 1,

How to compute $\Phi(n)$ for an input n?

A **simple solution** is to iterate through all numbers from 1 to $n-1$ and count numbers with gcd with n as 1. Below is C implementation of the simple method to compute Euler's Totient function for an input integer n .

```
// A simple C program to calculate Euler's Totient Function
#include <stdio.h>

// Function to return gcd of a and b
int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b%a, a);
}

// A simple method to evaluate Euler Totient Function
int phi(unsigned int n)
{
    unsigned int result = 1;
    for (int i=2; i < n; i++)
        if (gcd(i, n) == 1)
            result++;
    return result;
}

// Driver program to test above function
int main()
{
    int n;
    for (n=1; n<=10; n++)
        printf("phi(%d) = %d\n", n, phi(n));
    return 0;
}
```

Output:

```
phi(1) = 1
phi(2) = 1
phi(3) = 2
phi(4) = 2
phi(5) = 4
phi(6) = 2
phi(7) = 6
phi(8) = 4
phi(9) = 6
phi(10) = 4
```

The above code calls gcd function $O(n)$ times. Time complexity of the gcd function is $O(h)$ where h is number of digits in smaller number of given two numbers. Therefore, an upper bound on time complexity of above solution is $O(n \log n)$ [How? there can be at most $\log_{10} n$ digits in all numbers from 1 to n]

Below is a **Better Solution**. The idea is based on Euler's product formula which states that value of totient functions is below product over all prime factors p of n .

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right),$$

The formula basically says that the value of $\Phi(n)$ is equal to n multiplied by product of $(1 - 1/p)$ for all prime factors p of n . For example value of $\Phi(6) = 6 * (1-1/2) * (1 - 1/3) = 2$.

We can find all prime factors using the idea used in [this post](#).

1) Initialize : result = n

2) Run a loop from 'p' = 2 to \sqrt{n} , do following for every 'p'.

- a) If p divides n, then
Set: result = result * (1.0 - (1.0 / (float) p));
Divide all occurrences of p in n.
- 3) Return result

Below is C implementation of Euler's product formula.

```
// C program to calculate Euler's Totient Function
// using Euler's product formula
#include <stdio.h>

int phi(int n)
{
    float result = n; // Initialize result as n

    // Consider all prime factors of n and for every prime
    // factor p, multiply result with (1 - 1/p)
    for (int p=2; p*p<=n; ++p)
    {
        // Check if p is a prime factor.
        if (n % p == 0)
        {
            // If yes, then update n and result
            while (n % p == 0)
                n /= p;
            result *= (1.0 - (1.0 / (float) p));
        }
    }

    // If n has a prime factor greater than sqrt(n)
    // (There can be at-most one such prime factor)
    if (n > 1)
        result *= (1.0 - (1.0 / (float) n));

    return (int)result;
}

// Driver program to test above function
int main()
{
    int n;
    for (n=1; n<=10; n++)
        printf("phi(%d) = %d\n", n, phi(n));
    return 0;
}
```

Output:

```
phi(1) = 1
phi(2) = 1
phi(3) = 2
phi(4) = 2
phi(5) = 4
phi(6) = 2
phi(7) = 6
phi(8) = 4
phi(9) = 6
phi(10) = 4
```

We can avoid floating point calculations in above method. The idea is to count all prime factors and their multiples and subtract this count from n to get the totient function value (Prime factors and multiples of prime factors won't have gcd as 1)

- 1) Initialize result as n
- 2) Consider every number 'p' (where 'p' varies from 2 to \sqrt{n}).
If p divides n, then do following
 - a) Subtract all multiples of p from 1 to n [all multiples of p will have gcd more than 1 (at least p) with n]
 - b) Update n by repeatedly dividing it by p.
- 3) If the reduced n is more than 1, then remove all multiples of n from result.

Below is C implementation of above algorithm.

```
// C program to calculate Euler's Totient Function
#include <stdio.h>
```

```

int phi(int n)
{
    int result = n; // Initialize result as n

    // Consider all prime factors of n and subtract their
    // multiples from result
    for (int p=2; p*p<=n; ++p)
    {
        // Check if p is a prime factor.
        if (n % p == 0)
        {
            // If yes, then update n and result
            while (n % p == 0)
                n /= p;
            result -= result / p;
        }
    }

    // If n has a prime factor greater than sqrt(n)
    // (There can be at-most one such prime factor)
    if (n > 1)
        result -= result / n;
    return result;
}

// Driver program to test above function
int main()
{
    int n;
    for (n=1; n<=10; n++)
        printf("phi(%d) = %d\n", n, phi(n));
    return 0;
}

```

Output:

```

phi(1) = 1
phi(2) = 1
phi(3) = 2
phi(4) = 2
phi(5) = 4
phi(6) = 2
phi(7) = 6
phi(8) = 4
phi(9) = 6
phi(10) = 4

```

Let us take an example to understand the above algorithm.

```

n = 10.
Initialize: result = 10

2 is a prime factor, so n = n/i = 5, result = 5
3 is not a prime factor.

The for loop stops after 3 as 4*4 is not less than or equal
to 10.

After for loop, result = 5, n = 5
Since n > 1, result = result - result/n = 4

```

Some Interesting Properties of Euler's Totient Function

- 1) For a prime number p , $\Phi(p)$ is $p-1$. For example $\Phi(5)$ is 4, $\Phi(7)$ is 6 and $\Phi(13)$ is 12. This is obvious, gcd of all numbers from 1 to $p-1$ will be 1 because p is a prime.
 - 2) For two numbers a and b , if $\gcd(a, b) = 1$, then $\Phi(ab) = \Phi(a) * \Phi(b)$. For example $\Phi(5)$ is 4 and $\Phi(6)$ is 2, so $\Phi(30)$ must be 8 as 5 and 6 are relatively prime.
 - 3) For any two prime numbers p and q , $\Phi(pq) = (p-1)*(q-1)$. This property is used in RSA algorithm.
 - 4) If p is a prime number, then $\Phi(p^k) = p^k - p^{k-1}$. This can be proved using Euler's product formula.
 - 5) Sum of values of totient functions of all divisors of n is equal to n .
- $$\sum_{d|n} \varphi(d) = n,$$

For example, $n = 6$, the divisors of n are 1, 2, 3 and 6. According to Gauss, sum of $\Phi(1) + \Phi(2) + \Phi(3) + \Phi(6)$ should be 6. We can verify the same by putting values, we get $(1 + 1 + 2 + 2) = 6$.

6) The most famous and important feature is expressed in **Euler's theorem** :

The theorem states that if n and a are coprime (or relatively prime) positive integers, then

$$a^{\Phi(n)} \equiv 1 \pmod{n}$$

The **RSA cryptosystem** is based on this theorem:

In the particular case when m is prime say p , Euler's theorem turns into the so-called **Fermat's little theorem** :

$$a^{p-1} \equiv 1 \pmod{p}$$

Euler's Totient function for all numbers smaller than or equal to n

References:

http://e-maxx.ru/algo/euler_function

http://en.wikipedia.org/wiki/Euler%27s_totient_function

This article is contributed by **Ankur**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
GCD-LCM
MathematicalAlgo
modular-arithmetic
prime-number

Sieve of Eratosthenes

Given a number n , print all primes smaller than or equal to n . It is also given that n is a small number.

For example, if n is 10, the output should be "2, 3, 5, 7". If n is 20, the output should be "2, 3, 5, 7, 11, 13, 17, 19".

The sieve of Eratosthenes is one of the most efficient ways to find all primes smaller than n when n is smaller than 10 million or so (Ref [Wiki](#)).

We strongly recommend that you click here and practice it, before moving on to the solution.

Following is the algorithm to find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

1. Create a list of consecutive integers from 2 to n : (2, 3, 4, ..., n).
2. Initially, let p equal 2, the first prime number.
3. Starting from p , count up in increments of p and mark each of these numbers greater than p itself in the list. These numbers will be $2p$, $3p$, $4p$, etc.; note that some of them may have already been marked.
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this number (which is the next prime), and repeat from step 3.

When the algorithm terminates, all the numbers in the list that are not marked are prime.

Explanation with Example:

Let us take an example when $n = 50$. So we need to print all print numbers smaller than or equal to 50.

We create a list of all numbers from 2 to 50.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

According to the algorithm we will mark all the numbers which are divisible by 2.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Now we move to our next unmarked number 3 and mark all the numbers which are multiples of 3.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

We move to our next unmarked number 5 and mark all multiples of 5.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

We continue this process and our final table will look like below:

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

So the prime numbers are the unmarked ones: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47.

Thanks to **Krishan Kumar** for providing above explanation.

Implementation:

Following is C++ implementation of the above algorithm. In the following implementation, a boolean array arr[] of size n is used to mark multiples of prime numbers.

C/C++

```
// C++ program to print all primes smaller than or equal to
// n using Sieve of Eratosthenes
#include <bits/stdc++.h>
using namespace std;

void SieveOfEratosthenes(int n)
{
    // Create a boolean array "prime[0..n]" and initialize
    // all entries it as true. A value in prime[i] will
    // finally be false if i is Not a prime, else true.
    bool prime[n+1];
    memset(prime, true, sizeof(prime));

    for (int p=2; p*p<=n; p++)
    {
        // If prime[p] is not changed, then it is a prime
        if (prime[p] == true)
        {
            // Update all multiples of p
            for (int i=p*2; i<=n; i += p)
                prime[i] = false;
        }
    }

    // Print all prime numbers
    for (int p=2; p<=n; p++)
        if (prime[p])
            cout << p << " ";
}

// Driver Program to test above function
int main()
{
    int n = 30;
    cout << "Following are the prime numbers smaller "
        << " than or equal to " << n << endl;
    SieveOfEratosthenes(n);
    return 0;
}
```

Java

```

// Java program to print all primes smaller than or equal to
// n using Sieve of Eratosthenes

class SieveOfEratosthenes
{
void sieveOfEratosthenes(int n)
{
    // Create a boolean array "prime[0..n]" and initialize
    // all entries it as true. A value in prime[i] will
    // finally be false if i is Not a prime, else true.
    boolean prime[] = new boolean[n+1];
    for(int i=0;i<n;i++)
        prime[i] = true;

    for(int p = 2; p*p <=n; p++)
    {
        // If prime[p] is not changed, then it is a prime
        if(prime[p] == true)
        {
            // Update all multiples of p
            for(int i = p*2; i <= n; i += p)
                prime[i] = false;
        }
    }

    // Print all prime numbers
    for(int i = 2; i <= n; i++)
    {
        if(prime[i] == true)
            System.out.print(i + " ");
    }
}

// Driver Program to test above function
public static void main(String args[])
{
    int n = 30;
    System.out.print("Following are the prime numbers ");
    System.out.println("smaller than or equal to " + n);
    SieveOfEratosthenes g = new SieveOfEratosthenes();
    g.sieveOfEratosthenes(n);
}
}

```

// This code has been contributed by Amit Khandelwal.

Python

```

# Python program to print all primes smaller than or equal to
# n using Sieve of Eratosthenes

def SieveOfEratosthenes(n):

    # Create a boolean array "prime[0..n]" and initialize
    # all entries it as true. A value in prime[i] will
    # finally be false if i is Not a prime, else true.
    prime = [True for i in range(n+1)]
    p=2
    while(p * p <= n):

        # If prime[p] is not changed, then it is a prime
        if (prime[p] == True):

            # Update all multiples of p
            for i in range(p * 2, n+1, p):
                prime[i] = False
            p+=1
    lis =[]

    # Print all prime numbers
    for p in range(2, n):
        if prime[p]:
            print p,

```

```
# driver program
if __name__=='__main__':
    n = 30
    print "Following are the prime numbers smaller",
    print "than or equal to", n
    SieveOfEratosthenes(n)
```

Output:

```
Following are the prime numbers below 30
2 3 5 7 11 13 17 19 23 29
```

You may also like to see :

[Segmented Sieve.](#)

[Sieve of Eratosthenes in O\(n\) time complexity](#)

References:

http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

This article is compiled by **Abhinav Priyadarshi** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo
prime-number
sieve

Convex Hull | Set 1 (Jarvis's Algorithm or Wrapping)

Given a set of points in the plane. the convex hull of the set is the smallest convex polygon that contains all the points of it.



We strongly recommend to see the following post first.

[How to check if two given line segments intersect?](#)

The idea of Jarvis's Algorithm is simple, we start from the leftmost point (or point with minimum x coordinate value) and we keep wrapping points in counterclockwise direction. The big question is, given a point p as current point, how to find the next point in output? The idea is to use [orientation\(\)](#) here. Next point is selected as the point that beats all other points at counterclockwise orientation, i.e., next point is q if for any other point r, we have "orientation(p, r, q) = counterclockwise". Following is the detailed algorithm.

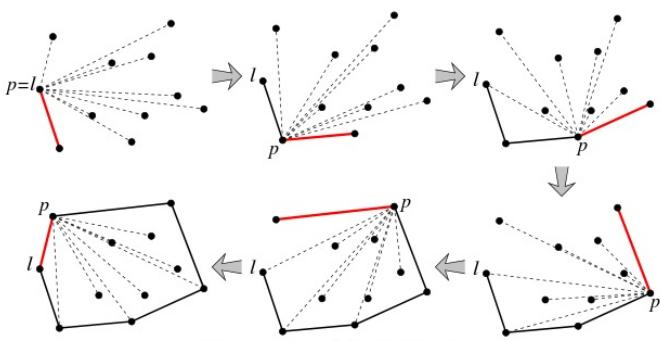
1) Initialize p as leftmost point.

2) Do following while we don't come back to the first (or leftmost) point.

.....**a) The next point q is the point such that the triplet (p, q, r) is counterclockwise for any other point r.**

.....**b) next[p] = q (Store q as next of p in the output convex hull).**

.....**c) p = q (Set p as q for next iteration).**



Below is C++ implementation of above algorithm.

```
// A C++ program to find convex hull of a set of points. Refer
```

```

// http://www.geeksforgeeks.org/orientation-3-ordered-points/
// for explanation of orientation()
#include <bits/stdc++.h>
using namespace std;

struct Point
{
    int x, y;
};

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0; // colinear
    return (val > 0)? 1: 2; // clock or counterclock wise
}

// Prints convex hull of a set of n points.
void convexHull(Point points[], int n)
{
    // There must be at least 3 points
    if (n < 3) return;

    // Initialize Result
    vector<Point> hull;

    // Find the leftmost point
    int l = 0;
    for (int i = 1; i < n; i++)
        if (points[i].x < points[l].x)
            l = i;

    // Start from leftmost point, keep moving counterclockwise
    // until reach the start point again. This loop runs O(h)
    // times where h is number of points in result or output.
    int p = l, q;
    do
    {
        // Add current point to result
        hull.push_back(points[p]);

        // Search for a point 'q' such that orientation(p, x,
        // q) is counterclockwise for all points 'x'. The idea
        // is to keep track of last visited most counterclock-
        // wise point in q. If any point 'i' is more counterclock-
        // wise than q, then update q.
        q = (p+1)%n;
        for (int i = 0; i < n; i++)
        {
            // If i is more counterclockwise than current q, then
            // update q
            if (orientation(points[p], points[i], points[q]) == 2)
                q = i;
        }

        // Now q is the most counterclockwise with respect to p
        // Set p as q for next iteration, so that q is added to
        // result 'hull'
        p = q;
    } while (p != l); // While we don't come to first point

    // Print Result
    for (int i = 0; i < hull.size(); i++)
        cout << "(" << hull[i].x << ", "
              << hull[i].y << ")\n";
}

```

```

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 3}, {2, 2}, {1, 1}, {2, 1},
                      {3, 0}, {0, 0}, {3, 3}};
    int n = sizeof(points)/sizeof(points[0]);
    convexHull(points, n);
    return 0;
}

```

Output: The output is points of the convex hull.

```

(0, 3)
(0, 0)
(3, 0)
(3, 3)

```

Time Complexity: For every point on the hull we examine all the other points to determine the next point. Time complexity is $(m * n)$ where n is number of input points and m is number of output or hull points ($m \leq n$). In worst case, time complexity is $O(n^2)$. The worst case occurs when all the points are on the hull ($m = n$)

Set 2- Convex Hull (Graham Scan)

Sources:

<http://www.cs.uiuc.edu/~jeffe/teaching/373/notes/x05-convexhull.pdf>
<http://www.dcs.gla.ac.uk/~pat/52233/slides/Hull1x1.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Geometric
geometric algorithms
MathematicalAlgo

Basic and Extended Euclidean algorithms

Basic Euclidean Algorithm is used to find GCD of two numbers say a and b . Below is a recursive C function to evaluate gcd using Euclid's algorithm.

```

// C program to demonstrate Basic Euclidean Algorithm
#include <stdio.h>

// Function to return gcd of a and b
int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b%a, a);
}

// Driver program to test above function
int main()
{
    int a = 10, b = 15;
    printf("GCD(%d, %d) = %d\n", a, b, gcd(a, b));
    a = 35, b = 10;
    printf("GCD(%d, %d) = %d\n", a, b, gcd(a, b));
    a = 31, b = 2;
    printf("GCD(%d, %d) = %d\n", a, b, gcd(a, b));
    return 0;
}

```

Output:

```

GCD(10, 15) = 5
GCD(35, 10) = 5
GCD(31, 2) = 1

```

Time Complexity: $O(\log \min(a, b))$

Extended Euclidean Algorithm:

Extended Euclidean algorithm also finds integer coefficients x and y such that:

$$ax + by = \text{gcd}(a, b)$$

Examples:

```
Input: a = 30, b = 20
Output: gcd = 10
        x = 1, y = -1
(Note that 30*1 + 20*(-1) = 10)
```

```
Input: a = 35, b = 15
Output: gcd = 5
        x = 1, y = -2
(Note that 10*0 + 5*1 = 5)
```

The extended Euclidean algorithm updates results of $\text{gcd}(a, b)$ using the results calculated by recursive call $\text{gcd}(b \% a, a)$. Let values of x and y calculated by the recursive call be x_1 and y_1 . x and y are updated using below expressions.

$$\begin{aligned}x &= y_1 - \lfloor b/a \rfloor * x_1 \\y &= x_1\end{aligned}$$

Below is C implementation based on above formulas.

```
// C program to demonstrate working of extended
// Euclidean Algorithm
#include <stdio.h>

// C function for extended Euclidean Algorithm
int gcdExtended(int a, int b, int *x, int *y)
{
    // Base Case
    if (a == 0)
    {
        *x = 0;
        *y = 1;
        return b;
    }

    int x1, y1; // To store results of recursive call
    int gcd = gcdExtended(b % a, a, &x1, &y1);

    // Update x and y using results of recursive
    // call
    *x = y1 - (b/a) * x1;
    *y = x1;

    return gcd;
}

// Driver Program
int main()
{
    int x, y;
    int a = 35, b = 15;
    int g = gcdExtended(a, b, &x, &y);
    printf("gcd(%d, %d) = %d, x = %d, y = %d",
          a, b, g, x, y);
    return 0;
}
```

Output:

$$\text{gcd}(35, 15) = 5, x = 1, y = -2$$

How does Extended Algorithm Work?

As seen above, x and y are results for inputs a and b,
 $a.x + b.y = \text{gcd}$ ----(1)

And x_1 and y_1 are results for inputs $b \% a$ and a
 $(b \% a).x_1 + a.y_1 = \text{gcd}$

When we put $b \% a = (b - (\lfloor b/a \rfloor) \cdot a)$ in above, we get following. Note that $\lfloor b/a \rfloor$ is floor(a/b)

$$(b - (\lfloor b/a \rfloor) \cdot a) \cdot x_1 + a \cdot y_1 = \gcd$$

Above equation can also be written as below

$$b \cdot x_1 + a \cdot (y_1 - (\lfloor b/a \rfloor) \cdot x_1) = \gcd \quad \text{---(2)}$$

After comparing coefficients of 'a' and 'b' in (1) and

(2), we get following

$$x = y_1 - \lfloor b/a \rfloor \cdot x_1$$

$$y = x_1$$

How is Extended Algorithm Useful?

The extended Euclidean algorithm is particularly useful when a and b are coprime (or gcd is 1). Since x is the modular multiplicative inverse of "a modulo b", and y is the modular multiplicative inverse of "b modulo a". In particular, the computation of the modular multiplicative inverse is an essential step in RSA public-key encryption method.

References:

http://e-maxx.ru/algo/extended_euclid_algorithm

http://en.wikipedia.org/wiki/Euclidean_algorithm

http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

This article is contributed by **Ankur**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
GCD-LCM
MathematicalAlgo

Segmented Sieve

Given a number n, print all primes smaller than n. For example, if the given number is 10, output 2, 3, 5, 7.

A Naive approach is to run a loop from 0 to n-1 and check each number for primeness. A Better Approach is use [Simple Sieve of Eratosthenes](#).

```
// This functions finds all primes smaller than 'limit'  
// using simple sieve of eratosthenes.  
void simpleSieve(int limit)  
{  
    // Create a boolean array "mark[0..limit-1]" and  
    // initialize all entries of it as true. A value  
    // in mark[p] will finally be false if 'p' is Not  
    // a prime, else true.  
    bool mark[limit];  
    memset(mark, true, sizeof(mark));  
  
    // One by one traverse all numbers so that their  
    // multiples can be marked as composite.  
    for (int p=2; p*p<limit; p++)  
    {  
        // If p is not changed, then it is a prime  
        if (mark[p] == true)  
        {  
            // Update all multiples of p  
            for (int i=p*p; i<limit; i+=p)  
                mark[i] = false;  
        }  
    }  
  
    // Print all prime numbers and store them in prime  
    for (int p=2; p<limit; p++)  
        if (mark[p] == true)  
            cout << p << " ";  
}
```

Problems with Simple Sieve:

The Sieve of Eratosthenes looks good, but consider the situation when n is large, the Simple Sieve faces following issues.

- An array of size $\Theta(n)$ may not fit in memory
- The simple Sieve is not cache friendly even for slightly bigger n. The algorithm traverses the array without locality of reference

Segmented Sieve

The idea of segmented sieve is to divide the range [0..n-1] in different segments and compute primes in all segments one by one. This algorithm first uses Simple Sieve to find primes smaller than or equal to \sqrt{n} . Below are steps used in Segmented Sieve.

1. Use Simple Sieve to find all primes upto square root of 'n' and store these primes in an array "prime[]". Store the found primes in an array 'prime[]'.
2. We need all primes in range [0..n-1]. We divide this range in different segments such that size of every segment is at-most \sqrt{n} .
3. Do following for every segment [low..high]
 - Create an array mark[high-low+1]. Here we need only $O(x)$ space where x is number of elements in given range.
 - Iterate through all primes found in step 1. For every prime, mark its multiples in given range [low..high].

In Simple Sieve, we needed $O(n)$ space which may not be feasible for large n. Here we need $O(\sqrt{n})$ space and we process smaller ranges at a time (locality of reference)

Below is C++ implementation of above idea.

```
// C++ program to print all primes smaller than
// n using segmented sieve
#include <bits/stdc++.h>
using namespace std;

// This function finds all primes smaller than 'limit'
// using simple sieve of eratosthenes. It also stores
// found primes in vector prime[]
void simpleSieve(int limit, vector<int> &prime)
{
    // Create a boolean array "mark[0..n-1]" and initialize
    // all entries of it as true. A value in mark[p] will
    // finally be false if 'p' is Not a prime, else true.
    bool mark[limit+1];
    memset(mark, true, sizeof(mark));

    for (int p=2; p*p<limit; p++)
    {
        // If p is not changed, then it is a prime
        if (mark[p] == true)
        {
            // Update all multiples of p
            for (int i=p*2; i<limit; i+=p)
                mark[i] = false;
        }
    }

    // Print all prime numbers and store them in prime
    for (int p=2; p<limit; p++)
    {
        if (mark[p] == true)
        {
            prime.push_back(p);
            cout << p << " ";
        }
    }
}

// Prints all prime numbers smaller than 'n'
void segmentedSieve(int n)
{
    // Compute all primes smaller than or equal
    // to square root of n using simple sieve
    int limit = floor(sqrt(n))+1;
    vector<int> prime;
    simpleSieve(limit, prime);

    // Divide the range [0..n-1] in different segments
    // We have chosen segment size as sqrt(n).
    int low = limit;
    int high = 2*limit;
```

```

// While all segments of range [0..n-1] are not processed,
// process one segment at a time
while (low < n)
{
    // To mark primes in current range. A value in mark[i]
    // will finally be false if 'i-low' is Not a prime,
    // else true.
    bool mark[limit+1];
    memset(mark, true, sizeof(mark));

    // Use the found primes by simpleSieve() to find
    // primes in current range
    for (int i = 0; i < prime.size(); i++)
    {
        // Find the minimum number in [low..high] that is
        // a multiple of prime[i] (divisible by prime[i])
        // For example, if low is 31 and prime[i] is 3,
        // we start with 33.
        int loLim = floor(low/prime[i]) * prime[i];
        if (loLim < low)
            loLim += prime[i];

        /* Mark multiples of prime[i] in [low..high]:
         We are marking j - low for j, i.e. each number
         in range [low, high] is mapped to [0, high-low]
         so if range is [50, 100] marking 50 corresponds
         to marking 0, marking 51 corresponds to 1 and
         so on. In this way we need to allocate space only
         for range */
        for (int j=loLim; j<high; j+=prime[i])
            mark[j-low] = false;
    }

    // Numbers which are not marked as false are prime
    for (int i = low; i<high; i++)
        if (mark[i - low] == true)
            cout << i << " ";
    }

    // Update low and high for next segment
    low = low + limit;
    high = high + limit;
    if (high >= n) high = n;
}
}

// Driver program to test above function
int main()
{
    int n = 100;
    cout << "Primes smaller than " << n << "\n";
    segmentedSieve(n);
    return 0;
}

```

Output:

```

Primes smaller than 100:
2 3 5 7 11 13 17 19 23 29 31 37 41
43 47 53 59 61 67 71 73 79 83 89 97

```

Note that time complexity (or number of operations) by Segmented Sieve is same as Simple Sieve. It has advantages for large 'n' as it has better locality of reference and requires

This article is contributed by [Utkarsh Trivedi](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Find the maximum subarray XOR in a given array

Given an array of integers. find the maximum XOR subarray value in given array. Expected time complexity O(n).

Examples:

Input: arr[] = {1, 2, 3, 4}

Output: 7

The subarray {3, 4} has maximum XOR value

Input: arr[] = {8, 1, 2, 12, 7, 6}

Output: 15

The subarray {1, 2, 12} has maximum XOR value

Input: arr[] = {4, 6}

Output: 6

The subarray {6} has maximum XOR value

We strongly recommend you to minimize your browser and try this yourself first.

A **Simple Solution** is to use two loops to find XOR of all subarrays and return the maximum.

```
// A simple C++ program to find max subarray XOR
#include<bits/stdc++.h>
using namespace std;

int maxSubarrayXOR(int arr[], int n)
{
    int ans = INT_MIN; // Initialize result

    // Pick starting points of subarrays
    for (int i=0; i<n; i++)
    {
        int curr_xor = 0; // to store xor of current subarray

        // Pick ending points of subarrays starting with i
        for (int j=i; j<n; j++)
        {
            curr_xor = curr_xor ^ arr[j];
            ans = max(ans, curr_xor);
        }
    }
    return ans;
}

// Driver program to test above functions
int main()
{
    int arr[] = {8, 1, 2, 12};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Max subarray XOR is " << maxSubarrayXOR(arr, n);
    return 0;
}
```

Output:

Max subarray XOR is 15

Time Complexity of above solution is O(n^2).

An **Efficient Solution** can solve the above problem in O(n) time under the assumption that integers take fixed number of bits to store. The idea is to use Trie Data Structure. Below is algorithm.

- 1) Create an empty Trie. Every node of Trie is going to contain two children, for 0 and 1 value of bit.
- 2) Initialize pre_xor = 0 and insert into the Trie.
- 3) Initialize result = minus infinite
- 4) Traverse the given array and do following for every array element arr[i].
 - a) pre_xor = pre_xor ^ arr[i]
pre_xor now contains xor of elements from arr[0] to arr[i].
 - b) Query the maximum xor value ending with arr[i] from Trie.
 - c) Update result if the value obtained in step 4.b is more than current value of result.

How does 4.b work?

We can observe from above algorithm that we build a Trie that contains XOR of all prefixes of given array. To find the maximum XOR subarray ending with arr[i], there may be two cases.

i) The prefix itself has the maximum XOR value ending with arr[i]. For example if i=2 in {8, 2, 1, 12}, then the maximum subarray xor ending with arr[2] is the whole prefix.

ii) We need to remove some prefix (ending at index from 0 to i-1). For example if i=3 in {8, 2, 1, 12}, then the maximum subarray xor ending with arr[3] starts with arr[1] and we need to remove arr[0].

To find the prefix to be removed, we find the entry in Trie that has maximum XOR value with current prefix. If we do XOR of such previous prefix with current prefix, we get the maximum XOR value ending with arr[i].

If there is no prefix to be removed (case i), then we return 0 (that's why we inserted 0 in Trie).

Below is C++ implementation of above algorithm.

```
// C++ program for a Trie based O(n) solution to find max
// subarray XOR
#include<bits/stdc++.h>
using namespace std;

// Assumed int size
#define INT_SIZE 32

// A Trie Node
struct TrieNode
{
    int value; // Only used in leaf nodes
    TrieNode *arr[2];
};

// Utility function tp create a Trie node
TrieNode *newNode()
{
    TrieNode *temp = new TrieNode;
    temp->value = 0;
    temp->arr[0] = temp->arr[1] = NULL;
    return temp;
}

// Inserts pre_xor to trie with given root
void insert(TrieNode *root, int pre_xor)
{
    TrieNode *temp = root;

    // Start from the msb, insert all bits of
    // pre_xor into Trie
    for (int i=INT_SIZE-1; i>=0; i--)
    {
        // Find current bit in given prefix
        bool val = pre_xor & (1<<i);

        // Create a new node if needed
        if (temp->arr[val] == NULL)
            temp->arr[val] = newNode();

        temp = temp->arr[val];
    }

    // Store value at leaf node
    temp->value = pre_xor;
}

// Finds the maximum XOR ending with last number in
// prefix XOR 'pre_xor' and returns the XOR of this maximum
// with pre_xor which is maximum XOR ending with last element
// of pre_xor.
int query(TrieNode *root, int pre_xor)
{
    TrieNode *temp = root;
    for (int i=INT_SIZE-1; i>=0; i--)
    {
        // Find current bit in given prefix
        bool val = pre_xor & (1<<i);
```

```

// Traverse Trie, first look for a
// prefix that has opposite bit
if (temp->arr[1-val]==NULL)
    temp = temp->arr[1-val];

// If there is no prefix with opposite
// bit, then look for same bit.
else if (temp->arr[val] != NULL)
    temp = temp->arr[val];
}

return pre_xor^(temp->value);
}

// Returns maximum XOR value of a subarray in arr[0..n-1]
int maxSubarrayXOR(int arr[], int n)
{
    // Create a Trie and insert 0 into it
    TrieNode *root = newNode();
    insert(root, 0);

    // Initialize answer and xor of current prefix
    int result = INT_MIN, pre_xor = 0;

    // Traverse all input array element
    for (int i=0; i<n; i++)
    {
        // update current prefix xor and insert it into Trie
        pre_xor = pre_xor^arr[i];
        insert(root, pre_xor);

        // Query for current prefix xor in Trie and update
        // result if required
        result = max(result, query(root, pre_xor));
    }
    return result;
}

// Driver program to test above functions
int main()
{
    int arr[] = {8, 1, 2, 12};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Max subarray XOR is " << maxSubarrayXOR(arr, n);
    return 0;
}

```

Output:

Max subarray XOR is 15

Exercise: Extend the above solution so that it also prints starting and ending indexes of subarray with maximum value (Hint: we can add one more field to Trie node to achieve this)

This article is contributed by **Romil Punetha**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Bit Magic
XOR

Find nth Magic Number

A magic number is defined as a number which can be expressed as a power of 5 or sum of unique powers of 5. First few magic numbers are 5, 25, 30(5 + 25), 125, 130(125 + 5),

Write a function to find the nth Magic number.

Example:

Input: n = 2
Output: 25

Input: n = 5
Output: 130

Source: Amazon written round question

We strongly recommend that you click here and practice it, before moving on to the solution.

If we notice carefully the magic numbers can be represented as 001, 010, 011, 100, 101, 110 etc, where 001 is $0^{\text{pow}(5,3)} + 0^{\text{pow}(5,2)} + 1^{\text{pow}(5,1)}$. So basically we need to add powers of 5 for each bit set in given integer n.

Below is C++ implementation based on this idea.

```
// C++ program to find nth magic number
#include <bits/stdc++.h>
using namespace std;

// Function to find nth magic number
int nthMagicNo(int n)
{
    int pow = 1, answer = 0;

    // Go through every bit of n
    while (n)
    {
        pow = pow*5;

        // If last bit of n is set
        if (n & 1)
            answer += pow;

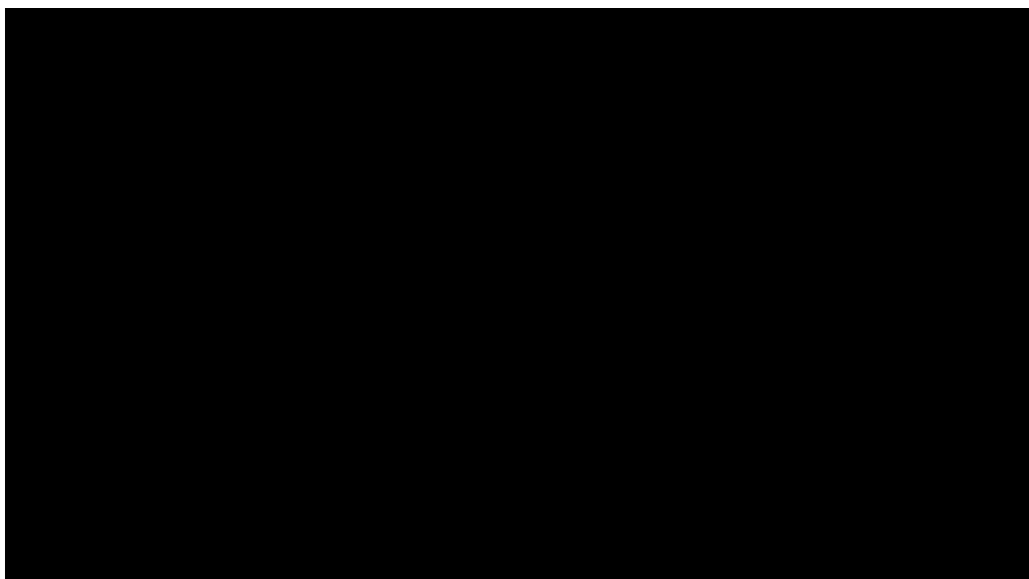
        // proceed to next bit
        n >>= 1; // or n = n/2
    }
    return answer;
}

// Driver program to test above function
int main()
{
    int n = 5;
    cout << "nth magic number is " << nthMagicNo(n) << endl;
    return 0;
}
```

Output:

nth magic number is 130

Thanks to manralsingh for suggesting above solution.



GATE CS Corner Company Wise Coding Practice

Bit Magic

Sum of bit differences among all pairs

Given an integer array of n integers, find sum of bit differences in all pairs that can be formed from array elements. Bit difference of a pair (x, y) is count of different bits at same positions in binary representations of x and y.

For example, bit difference for 2 and 7 is 2. Binary representation of 2 is 010 and 7 is 111 (first and last bits differ in two numbers).

Examples:

```
Input: arr[] = {1, 2}
Output: 4
All pairs in array are (1, 1), (1, 2)
          (2, 1), (2, 2)
Sum of bit differences = 0 + 2 +
                        2 + 0
                        = 4

Input: arr[] = {1, 3, 5}
Output: 8
All pairs in array are (1, 1), (1, 3), (1, 5)
          (3, 1), (3, 3) (3, 5),
          (5, 1), (5, 3), (5, 5)
Sum of bit differences = 0 + 1 + 1 +
                        1 + 0 + 2 +
                        1 + 2 + 0
                        = 8
```

Source: Google Interview Question

We strongly recommend that you click here and practice it, before moving on to the solution.

A **Simple Solution** is to run two loops to consider all pairs one by one. For every pair, count bit differences. Finally return sum of counts. Time complexity of this solution is $O(n^2)$.

An **Efficient Solution** can solve this problem in $O(n)$ time using the fact that all numbers are represented using 32 bits (or some fixed number of bits). The idea is to count differences at individual bit positions. We traverse from 0 to 31 and count numbers with i'th bit set. Let this count be 'count'. There would be "n-count" numbers with i'th bit not set. So count of differences at i'th bit would be "count * (n-count) * 2".

Below is C++ implementation of above idea.

```
// C++ program to compute sum of pairwise bit differences
#include <bits/stdc++.h>
using namespace std;

int sumBitDifferences(int arr[], int n)
{
    int ans = 0; // Initialize result

    // traverse over all bits
    for (int i = 0; i < 32; i++)
    {
        // count number of elements with i'th bit set
        int count = 0;
        for (int j = 0; j < n; j++)
            if ((arr[j] & (1 << i)) )
                count++;

        // Add "count * (n - count) * 2" to the answer
        ans += (count * (n - count) * 2);
    }

    return ans;
```

```

}

// Driver program
int main()
{
    int arr[] = {1, 3, 5};
    int n = sizeof arr / sizeof arr[0];
    cout << sumBitDifferences(arr, n) << endl;
    return 0;
}

```

Output:

8

Thanks to [Gaurav Ahirwar](#) for suggesting this solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Bit Magic
Google

Swap all odd and even bits

Given an unsigned integer, swap all odd bits with even bits. For example, if the given number is 23 (**00010111**), it should be converted to 43 (**00101011**). Every even position bit is swapped with adjacent bit on right side (even position bits are highlighted in binary representation of 23), and every odd position bit is swapped with adjacent on left side.

If we take a closer look at the example, we can observe that we basically need to right shift (**>>**) all even bits (In the above example, even bits of 23 are highlighted) by 1 so that they become odd bits (highlighted in 43), and left shift (**<<**) all odd bits by 1 so that they become even bits. The following solution is based on this observation. The solution assumes that input number is stored using 32 bits.

Let the input number be x

- 1) Get all even bits of x by doing bitwise and of x with 0xAAAAAAA. The number 0xAAAAAAA is a 32 bit number with all even bits set as 1 and all odd bits as 0.
- 2) Get all odd bits of x by doing bitwise and of x with 0x5555555. The number 0x5555555 is a 32 bit number with all odd bits set as 1 and all even bits as 0.
- 3) Right shift all even bits.
- 4) Left shift all odd bits.
- 5) Combine new even and odd bits and return.

C/C++

```

// C program to swap even and odd bits of a given number
#include <stdio.h>

unsigned int swapBits(unsigned int x)
{
    // Get all even bits of x
    unsigned int even_bits = x & 0xAAAAAAA;

    // Get all odd bits of x
    unsigned int odd_bits = x & 0x5555555;

    even_bits >>= 1; // Right shift even bits
    odd_bits <<= 1; // Left shift odd bits

    return (even_bits | odd_bits); // Combine even and odd bits
}

// Driver program to test above function
int main()
{
    unsigned int x = 23; // 00010111

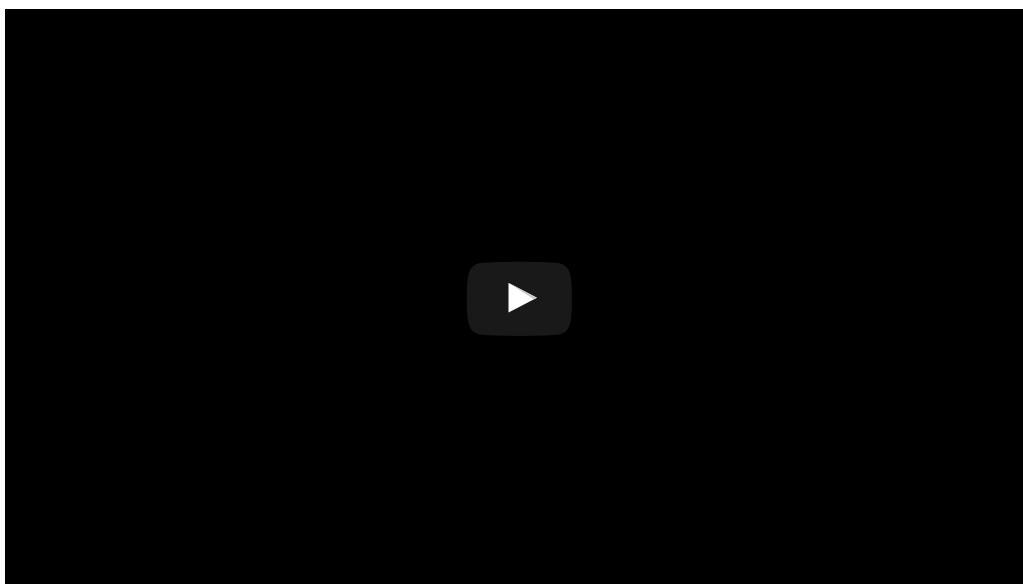
    // Output is 43 (00101011)
    printf("%u ", swapBits(x));
}

```

```
    return 0;  
}
```

Output:

```
43
```



Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Bit Magic

Find the element that appears once

Given an array where every element occurs three times, except one element which occurs only once. Find the element that occurs once. Expected time complexity is O(n) and O(1) extra space.

Examples:

```
Input: arr[] = {12, 1, 12, 3, 12, 1, 1, 2, 3, 3}  
Output: 2
```

We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

We can use sorting to do it in O(nLogn) time. We can also use hashing, but the worst case time complexity of but hashing requires extra space.

The idea is to use bitwise operators for a solution that is O(n) time and uses O(1) extra space. The solution is not easy like other XOR based solutions, because all elements appear odd number of times here. The idea is taken from [here](#).

Run a loop for all elements in array. At the end of every iteration, maintain following two values.

ones: The bits that have appeared 1st time or 4th time or 7th time .. etc.

twos: The bits that have appeared 2nd time or 5th time or 8th time .. etc.

Finally, we return the value of 'ones'

How to maintain the values of 'ones' and 'twos'?

'ones' and 'twos' are initialized as 0. For every new element in array, find out the common set bits in the new element and previous value of 'ones'. These common set bits are actually the bits that should be added to 'twos'. So do bitwise OR of the common set bits with 'twos'. 'twos' also gets some extra bits that appear third time. These extra bits are removed later.

Update 'ones' by doing XOR of new element with previous value of 'ones'. There may be some bits which appear 3rd time. These extra bits are also removed later.

Both 'ones' and 'twos' contain those extra bits which appear 3rd time. Remove these extra bits by finding out common set bits in 'ones' and 'twos'.

```

#include <stdio.h>

int getSingle(int arr[], int n)
{
    int ones = 0, twos = 0;

    int common_bit_mask;

    // Let us take the example of {3, 3, 2, 3} to understand this
    for( int i=0; i< n; i++ )
    {
        /* The expression "one & arr[i]" gives the bits that are
         there in both 'ones' and new element from arr[]. We
         add these bits to 'twos' using bitwise OR

         Value of 'twos' will be set as 0, 3, 3 and 1 after 1st,
         2nd, 3rd and 4th iterations respectively */
        twos = twos | (ones & arr[i]);
    }

    /* XOR the new bits with previous 'ones' to get all bits
     appearing odd number of times

     Value of 'ones' will be set as 3, 0, 2 and 3 after 1st,
     2nd, 3rd and 4th iterations respectively */
    ones = ones ^ arr[i];
}

/* The common bits are those bits which appear third time
 So these bits should not be there in both 'ones' and 'twos'.
 common_bit_mask contains all these bits as 0, so that the bits can
 be removed from 'ones' and 'twos'

 Value of 'common_bit_mask' will be set as 00, 00, 01 and 10
 after 1st, 2nd, 3rd and 4th iterations respectively */
common_bit_mask = ~(ones & twos);

/* Remove common bits (the bits that appear third time) from 'ones'

 Value of 'ones' will be set as 3, 0, 0 and 2 after 1st,
 2nd, 3rd and 4th iterations respectively */
ones &= common_bit_mask;

/* Remove common bits (the bits that appear third time) from 'twos'

 Value of 'twos' will be set as 0, 3, 1 and 0 after 1st,
 2nd, 3rd and 4th iterations respectively */
twos &= common_bit_mask;

// uncomment this code to see intermediate values
//printf ("%d %d \n", ones, twos);
}

return ones;
}

int main()
{
    int arr[] = {3, 3, 2, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("The element with single occurrence is %d ",
        getSingle(arr, n));
    return 0;
}

```

Output:

2

Time Complexity: O(n)

Auxiliary Space: O(1)

Following is another O(n) time complexity and O(1) extra space method suggested by *aj*. We can sum the bits in same positions for all the

numbers and take modulo with 3. The bits for which sum is not multiple of 3, are the bits of number with single occurrence.

Let us consider the example array {5, 5, 5, 8}. The 101, 101, 101, 1000

Sum of first bits%3 = $(1 + 1 + 1 + 0)\%3 = 0$;

Sum of second bits%3 = $(0 + 0 + 0 + 0)\%0 = 0$;

Sum of third bits%3 = $(1 + 1 + 1 + 0)\%3 = 0$;

Sum of fourth bits%3 = $(1)\%3 = 1$;

Hence number which appears once is 1000

```
#include <stdio.h>
#define INT_SIZE 32

int getSingle(int arr[], int n)
{
    // Initialize result
    int result = 0;

    int x, sum;

    // Iterate through every bit
    for (int i = 0; i < INT_SIZE; i++)
    {
        // Find sum of set bits at ith position in all
        // array elements
        sum = 0;
        x = (1 << i);
        for (int j=0; j< n; j++)
        {
            if (arr[j] & x)
                sum++;
        }

        // The bits with sum not multiple of 3, are the
        // bits of element with single occurrence.
        if (sum % 3)
            result |= x;
    }

    return result;
}

// Driver program to test above function
int main()
{
    int arr[] = {12, 1, 12, 3, 12, 1, 1, 2, 3, 2, 2, 3, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("The element with single occurrence is %d ",
          getSingle(arr, n));
    return 0;
}
```

7

This article is compiled by **Sumit Jain** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Bit Magic
XOR

Binary representation of a given number

Write a program to print Binary representation of a given number.

Source: Microsoft Interview Set-3

Method 1: Iterative

For any number, we can check whether its 'i'th bit is 0(OFF) or 1(ON) by bitwise ANDing it with " 2^i " (2 raise to i).

- 1) Let us take number 'NUM' and we want to check whether it's 0th bit is ON or OFF
 $bit = 2 ^ 0$ (0th bit)

```
if NUM & bit == 1 means 0th bit is ON else 0th bit is OFF
```

2) Similarly if we want to check whether 5th bit is ON or OFF

```
bit = 2 ^ 5 (5th bit)
```

```
if NUM & bit == 1 means its 5th bit is ON else 5th bit is OFF.
```

Let us take unsigned integer (32 bit), which consist of 0-31 bits. To print binary representation of unsigned integer, start from 31th bit, check whether 31th bit is ON or OFF, if it is ON print "1" else print "0". Now check whether 30th bit is ON or OFF, if it is ON print "1" else print "0", do this for all bits from 31 to 0, finally we will get binary representation of number.

```
void bin(unsigned n)
{
    unsigned i;
    for (i = 1 << 31; i > 0; i = i / 2)
        (n & i)? printf("1"): printf("0");
}

int main(void)
{
    bin(7);
    printf("\n");
    bin(4);
}
```

Method 2: Recursive

Following is recursive method to print binary representation of 'NUM'.

step 1) if NUM > 1

- a) push NUM on stack
- b) recursively call function with 'NUM / 2'

step 2)

- a) pop NUM from stack, divide it by 2 and print it's remainder.

```
void bin(unsigned n)
{
    /* step 1 */
    if (n > 1)
        bin(n/2);

    /* step 2 */
    printf("%d", n % 2);
}

int main(void)
{
    bin(7);
    printf("\n");
    bin(4);
}
```

This article is compiled by **Narendra Kangalkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Bit Magic

Count total set bits in all numbers from 1 to n

Given a positive integer n, count the total number of set bits in binary representation of all numbers from 1 to n.

Examples:

```
Input: n = 3
Output: 4
```

```
Input: n = 6
Output: 9
```

```
Input: n = 7
Output: 12
```

```
Input: n = 8
Output: 13
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Source: Amazon Interview Question

Method 1 (Simple)

A simple solution is to run a loop from 1 to n and sum the count of set bits in all numbers from 1 to n.

```
// A simple program to count set bits in all numbers from 1 to n.
#include <stdio.h>

// A utility function to count set bits in a number x
unsigned int countSetBitsUtil(unsigned int x);

// Returns count of set bits present in all numbers from 1 to n
unsigned int countSetBits(unsigned int n)
{
    int bitCount = 0; // initialize the result

    for(int i = 1; i <= n; i++)
        bitCount += countSetBitsUtil(i);

    return bitCount;
}

// A utility function to count set bits in a number x
unsigned int countSetBitsUtil(unsigned int x)
{
    if (x <= 0)
        return 0;
    return (x % 2 == 0? 0: 1) + countSetBitsUtil (x/2);
}

// Driver program to test above functions
int main()
{
    int n = 4;
    printf ("Total set bit count is %d", countSetBits(n));
    return 0;
}
```

Output:

```
Total set bit count is 5
```

Time Complexity: O(nLogn)

Method 2 (Tricky)

If the input number is of the form $2^b - 1$ e.g., 1,3,7,15.. etc, the number of set bits is $b * 2^{(b-1)}$. This is because for all the numbers 0 to $(2^b)-1$, if you complement and flip the list you end up with the same list (half the bits are on, half off).

If the number does not have all set bits, then some position m is the position of leftmost set bit. The number of set bits in that position is $n - (0|0\ 0|0\ 0|1\ 0|1\ 0\ 0|1\ 1\ |- 1|0\ 0\ 1|0\ 1\ 1|1\ 0)$

The leftmost set bit is in position 2 (positions are considered starting from 0). If we mask that off what remains is 2 (the "1 0" in the right part of the last row.) So the number of bits in the 2nd position (the lower left box) is 3 (that is, $2 + 1$). The set bits from 0-3 (the upper right box above) is $2 * 2^{(2-1)} = 4$. The box in the lower right is the remaining bits we haven't yet counted, and is the number of set bits for all the numbers up to 2 (the value of the last entry in the lower right box) which can be figured recursively.

```
// A O(Logn) complexity program to count set bits in all numbers from 1 to n
#include <stdio.h>

/* Returns position of leftmost set bit. The rightmost
position is considered as 0 */
unsigned int getLeftmostBit (int n)
{
```

```

int m = 0;
while (n > 1)
{
    n = n >> 1;
    m++;
}
return m;
}

/* Given the position of previous leftmost set bit in n (or an upper
bound on leftmost position) returns the new position of leftmost
set bit in n */
unsigned int getNextLeftmostBit (int n, int m)
{
    unsigned int temp = 1 << m;
    while (n < temp)
    {
        temp = temp >> 1;
        m--;
    }
    return m;
}

// The main recursive function used by countSetBits()
unsigned int _countSetBits(unsigned int n, int m);

// Returns count of set bits present in all numbers from 1 to n
unsigned int countSetBits(unsigned int n)
{
    // Get the position of leftmost set bit in n. This will be
    // used as an upper bound for next set bit function
    int m = getLeftmostBit (n);

    // Use the position
    return _countSetBits (n, m);
}

unsigned int _countSetBits(unsigned int n, int m)
{
    // Base Case: if n is 0, then set bit count is 0
    if (n == 0)
        return 0;

    /* get position of next leftmost set bit */
    m = getNextLeftmostBit(n, m);

    // If n is of the form 2^x-1, i.e., if n is like 1, 3, 7, 15, 31.. etc,
    // then we are done.
    // Since positions are considered starting from 0, 1 is added to m
    if (n == ((unsigned int)1<<(m+1))-1)
        return (unsigned int)(m+1)*(1<<m);

    // update n for next recursive call
    n = n - (1<<m);
    return (n+1) + countSetBits(n) + m*(1<<(m-1));
}

// Driver program to test above functions
int main()
{
    int n = 17;
    printf ("Total set bit count is %d", countSetBits(n));
    return 0;
}

```

Total set bit count is 35

Time Complexity: O(Logn). From the first look at the implementation, time complexity looks more. But if we take a closer look, statements inside while loop of getNextLeftmostBit() are executed for all 0 bits in n. And the number of times recursion is executed is less than or equal to set bits in n. In other words, if the control goes inside while loop of getNextLeftmostBit(), then it skips those many bits in recursion.

Thanks to [agatsu](#) and [IC](#) for suggesting this solution.

See [this](#) for another solution suggested by Piyush Kapoor.

GATE CS Corner Company Wise Coding Practice

Bit Magic
Amazon

Rotate bits of a number

Bit Rotation: A rotation (or circular shift) is an operation similar to shift except that the bits that fall off at one end are put back to the other end.

In left rotation, the bits that fall off at left end are put back at right end.

In right rotation, the bits that fall off at right end are put back at left end.

Example:

Let n is stored using 8 bits. Left rotation of n = 11100101 by 3 makes n = 00101111 (Left shifted by 3 and first 3 bits are put back in last). If n is stored using 16 bits or 32 bits then left rotation of n (000...11100101) becomes 00..00**11100101**000.

Right rotation of n = 11100101 by 3 makes n = 10111100 (Right shifted by 3 and last 3 bits are put back in first) if n is stored using 8 bits. If n is stored using 16 bits or 32 bits then right rotation of n (000...11100101) by 3 becomes **101000..0011100**.

```
#include<stdio.h>
#define INT_BITS 32

/*Function to left rotate n by d bits*/
int leftRotate(int n, unsigned int d)
{
    /* In n<<d, last d bits are 0. To put first 3 bits of n at
     * last, do bitwise or of n<<d with n >>(INT_BITS - d) */
    return (n << d)|(n >> (INT_BITS - d));
}

/*Function to right rotate n by d bits*/
int rightRotate(int n, unsigned int d)
{
    /* In n>>d, first d bits are 0. To put last 3 bits of at
     * first, do bitwise or of n>>d with n <<(INT_BITS - d) */
    return (n >> d)|(n << (INT_BITS - d));
}

/* Driver program to test above functions */
int main()
{
    int n = 16;
    int d = 2;
    printf("Left Rotation of %d by %d is ", n, d);
    printf("%d", leftRotate(n, d));
    printf("\nRight Rotation of %d by %d is ", n, d);
    printf("%d", rightRotate(n, d));
    getchar();
}
```

Please write comments if you find any bug in the above program or other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic
rotation

Count number of bits to be flipped to convert A to B

Suggested by Dheeraj

Question: You are given two numbers A and B. Write a program to count number of bits needed to be flipped to convert A to B.

Solution:

1. Calculate XOR of A and B.
 $a_xor_b = A \wedge B$

2. Count the set bits in the above calculated XOR result.
countSetBits(a_xor_b)

XOR of two number will have set bits only at those places where A differs from B.

Example:

```
A = 1001001  
B = 0010101  
a_xor_b = 1011100  
No of bits need to flipped = set bit count in a_xor_b i.e. 4
```

We strongly recommend that you click here and practice it, before moving on to the solution.

To get the set bit count please see another post on this portal <http://geeksforgeeks.org/?p=1176>

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic
XOR

Find Next Sparse Number

A number is Sparse if there are no two adjacent 1s in its binary representation. For example 5 (binary representation: 101) is sparse, but 6 (binary representation: 110) is not sparse.

Given a number x, find the smallest Sparse number which greater than or equal to x.

Examples:

```
Input: x = 6  
Output: Next Sparse Number is 8  
  
Input: x = 4  
Output: Next Sparse Number is 4  
  
Input: x = 38  
Output: Next Sparse Number is 40  
  
Input: x = 44  
Output: Next Sparse Number is 64
```

We strongly recommend that you click here and practice it, before moving on to the solution.

A **Simple Solution** is to do following:

- 1) Write a utility function isSparse(x) that takes a number and returns true if x is sparse, else false. This function can be easily written by traversing the bits of input number.
- 2) Start from x and do following

```
while(1)
{
    if (isSparse(x))
        return x;
    else
        x++;
}
```

Time complexity of isSparse() is O(Log x). Time complexity of this solution is O(x Log x). The next sparse number can be at most O(x) distance away.

Thanks to [kk_angel](#) for suggesting above solution.

An **Efficient Solution** can solve this problem without checking all numbers one by one. Below are steps.

- 1) Find binary of the given number and store it in a

- boolean array.
- 2) Initialize last_finalized bit position as 0.
 - 2) Start traversing the binary from least significant bit.
 - a) If we get two adjacent 1's such that next (or third) bit is not 1, then
 - (i) Make all bits after this 1 to last finalized bit (including last finalized) as 0.
 - (ii) Update last finalized bit as next bit.

For example, let binary representation be 010100010**11101**, we change it to 01010001**100000** (all bits after highlighted 11 are set to 0). Again two 1's are adjacent, so change binary representation to 010100**10000000**. This is our final answer.

Below is C++ implementation of above solution.

```
// C++ program to find next sparse number
#include<bits/stdc++.h>
using namespace std;

int nextSparse(int x)
{
    // Find binary representation of x and store it in bin[].
    // bin[0] contains least significant bit (LSB), next
    // bit is in bin[1], and so on.
    vector<bool> bin;
    while (x != 0)
    {
        bin.push_back(x&1);
        x >>= 1;
    }

    // There may be extra bit in result, so add one extra bit
    bin.push_back(0);
    int n = bin.size(); // Size of binary representation

    // The position till which all bits are finalized
    int last_final = 0;

    // Start from second bit (next to LSB)
    for (int i=1; i<n-1; i++)
    {
        // If current bit and its previous bit are 1, but next
        // bit is not 1.
        if (bin[i] == 1 && bin[i-1] == 1 && bin[i+1] != 1)
        {
            // Make the next bit 1
            bin[i+1] = 1;

            // Make all bits before current bit as 0 to make
            // sure that we get the smallest next number
            for (int j=i; j>=last_final; j--)
                bin[j] = 0;

            // Store position of the bit set so that this bit
            // and bits before it are not changed next time.
            last_final = i+1;
        }
    }

    // Find decimal equivalent of modified bin[]
    int ans = 0;
    for (int i =0; i<n; i++)
        ans += bin[i]*(1<<i);
    return ans;
}

// Driver program
int main()
{
    int x = 38;
    cout << "Next Sparse Number is " << nextSparse(x);
    return 0;
}
```

Output:

Next Sparse Number is 40

Time complexity of this solution is O(Log x).

Thanks to [gcccde](#) for suggesting above solution [here](#).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Bit Magic

Reverse an array without affecting special characters

Given a string, that contains special character together with alphabets ('a' to 'z' and 'A' to 'Z'), reverse the string in a way that special characters are not affected.

Examples:

```
Input: str = "a,b$c"
Output: str = "c,b$a"
Note that $ and , are not moved anywhere.
Only subsequence "abc" is reversed
```

```
Input: str = "Ab,c,de$"
Output: str = "ed,c,bA$"
```

We strongly recommend you to minimize your browser and try this yourself first.

Simple Solution:

- 1) Create a temporary character array say temp[].
- 2) Copy alphabetic characters from given array to temp[].
- 3) Reverse temp[] using standard [string reversal algorithm](#).
- 4) Now traverse input string and temp in a single loop. Wherever there is alphabetic character is input string, replace it with current character of temp[].

Efficient Solution:

Time complexity of above solution is O(n), but it requires extra space and it does two traversals of input string.

We can reverse with one traversal and without extra space. Below is algorithm.

- 1) Let input string be 'str[]' and length of string be 'n'
- 2) l = 0, r = n-1
- 3) While l is smaller than r, do following
 - a) If str[l] is not an alphabetic character, do l++
 - b) Else If str[r] is not an alphabetic character, do r--
 - c) Else swap str[l] and str[r]

Below is C++ implementation of above algorithm.

C++

```
// C++ program to reverse a string with special characters
#include<bits/stdc++.h>
using namespace std;

// Returns true if x is an aplhabetic character, false otherwise
bool isAlphabet(char x)
{
    return ( (x >= 'A' && x <= 'Z') ||
            (x >= 'a' && x <= 'z') );
}

void reverse(char str[])
{
    // Initialize left and right pointers
    int r = strlen(str) - 1, l = 0;

    // Traverse string from both ends until
    // 'l' and 'r'
    while (l < r)
```

```

{
    // Ignore special characters
    if (!isAlphabet(str[l]))
        l++;
    else if (!isAlphabet(str[r]))
        r--;

    else // Both str[l] and str[r] are not spacial
    {
        swap(str[l], str[r]);
        l++;
        r--;
    }
}

// Driver program
int main()
{
    char str[] = "a!!!b.c.d,e'f,ghi";
    cout << "Input string: " << str << endl;
    reverse(str);
    cout << "Output string: " << str << endl;
    return 0;
}

```

Python

```

# Python program to reverse a string with special characters

# Returns true if x is an aplhabetic character, false otherwise
def isAlphabet(x):
    return x.isalpha()

def reverse(string):
    LIST = toList(string)

    # Initialize left and right pointers
    r = len(LIST) - 1
    l = 0

    # Traverse LIST from both ends until
    # 'l' and 'r'
    while l < r:

        # Ignore special characters
        if not isAlphabet(LIST[l]):
            l += 1
        elif not isAlphabet(LIST[r]):
            r -= 1

        else: # Both LIST[l] and LIST[r] are not special
            LIST[l], LIST[r] = swap(LIST[l], LIST[r])
            l += 1
            r -= 1

    return toString(LIST)

# Utility functions
def toList(string):
    List = []
    for i in string:
        List.append(i)
    return List

def toString(List):
    return ''.join(List)

def swap(a, b):
    return b, a

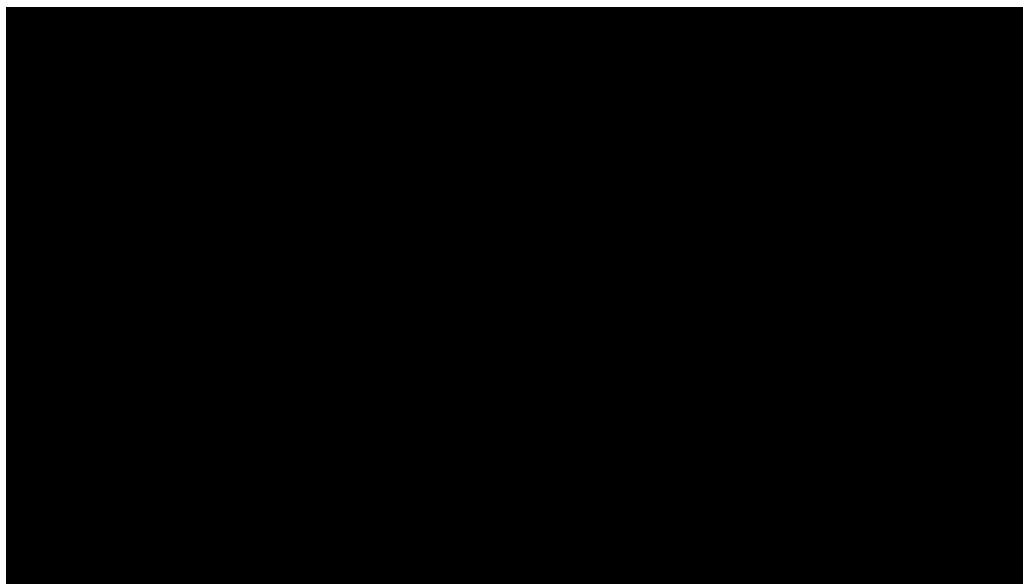
# Driver program
string = "a!!!b.c.d,e'f,ghi"

```

```
print "Input string: " + string
string = reverse(string)
print "Output string: " + string
# This code is contributed by Bhavya Jain
```

Output:

```
Input string: a!!!b.c.d,e'f,ghi
Output string: i!!!h.g.f,e'd,cba
```



Thanks to anonymous user for suggesting above solution here.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Strings
Reverse

Given a string, print all possible palindromic partitions

Given a string, find all possible palindromic partitions of given string.

Example:

```
Input: nitin
Output: n i t i n
       n i t i n
       nitin

Input: geeks
Output: g e e k s
       g ee k s
```

We strongly recommend you to minimize your browser and try this yourself first.

Note that this problem is different from [Palindrome Partitioning Problem](#), there the task was to find the partitioning with minimum cuts in input string. Here we need to print all possible partitions.

The idea is to go through every substring starting from first character, check if it is palindrome. If yes, then add the substring to solution and recur for remaining part. Below is complete algorithm.

Below is C++ implementation of above idea

C++

```
// C++ program to print all palindromic partitions of a given string
#include<bits/stdc++.h>
using namespace std;

// A utility function to check if str is palindromic
```

```

bool isPalindrome(string str, int low, int high)
{
    while (low < high)
    {
        if (str[low] != str[high])
            return false;
        low++;
        high--;
    }
    return true;
}

// Recursive function to find all palindromic partitions of str[start..n-1]
// allPart --> A vector of vector of strings. Every vector inside it stores
//   a partition
// currPart --> A vector of strings to store current partition
void allPalPartUtil(vector<vector<string>>&allPart, vector<string> &currPart,
                     int start, int n, string str)
{
    // If 'start' has reached len
    if (start >= n)
    {
        allPart.push_back(currPart);
        return;
    }

    // Pick all possible ending points for substrings
    for (int i=start; i<n; i++)
    {
        // If substring str[start..i] is palindrome
        if (isPalindrome(str, start, i))
        {
            // Add the substring to result
            currPart.push_back(str.substr(start, i-start+1));

            // Recur for remaining remaining substring
            allPalPartUtil(allPart, currPart, i+1, n, str);

            // Remove substring str[start..i] from current
            // partition
            currPart.pop_back();
        }
    }
}

// Function to print all possible palindromic partitions of
// str. It mainly creates vectors and calls allPalPartUtil()
void allPalPartitions(string str)
{
    int n = str.length();

    // To Store all palindromic partitions
    vector<vector<string>> allPart;

    // To store current palindromic partition
    vector<string> currPart;

    // Call recursive function to generate all partitions
    // and store in allPart
    allPalPartUtil(allPart, currPart, 0, n, str);

    // Print all partitions generated by above call
    for (int i=0; i< allPart.size(); i++ )
    {
        for (int j=0; j<allPart[i].size(); j++)
            cout << allPart[i][j] << " ";
        cout << "\n";
    }
}

// Driver program
int main()
{
    string str = "nitin";
    allPalPartitions(str);
}

```

```
return 0;  
}
```

Python

```
# A O(n^2) time and O(1) space program to find the  
#longest palindromic substring  
  
# This function prints the longest palindrome substring (LPS)  
# of str[]. It also returns the length of the longest palindrome  
def longestPalSubstr(string):  
    maxLength = 1  
  
    start = 0  
    length = len(string)  
  
    low = 0  
    high = 0  
  
    # One by one consider every character as center point of  
    # even and length palindromes  
    for i in xrange(1, length):  
        # Find the longest even length palindrome with center  
        # points as i-1 and i.  
        low = i - 1  
        high = i  
        while low >= 0 and high < length and string[low] == string[high]:  
            if high - low + 1 > maxLength:  
                start = low  
                maxLength = high - low + 1  
            low -= 1  
            high += 1  
  
        # Find the longest odd length palindrome with center  
        # point as i  
        low = i - 1  
        high = i + 1  
        while low >= 0 and high < length and string[low] == string[high]:  
            if high - low + 1 > maxLength:  
                start = low  
                maxLength = high - low + 1  
            low -= 1  
            high += 1  
  
    print "Longest palindrome substring is:",  
    print string[start:start + maxLength]  
  
    return maxLength  
  
# Driver program to test above functions  
string = "forgeeksskeegfor"  
print "Length is: " + str(longestPalSubstr(string))  
  
# This code is contributed by BHAVYA JAIN
```

Output:

```
n i t i n  
n i t i n  
nitin
```

This article is contributed by [Ekta Goel](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Count triplets with sum smaller than a given value

Given an array of distinct integers and a sum value. Find count of triplets with sum smaller than given sum value. Expected Time Complexity is $O(n^2)$.

Examples:

```
Input : arr[] = {-2, 0, 1, 3}
        sum = 2.
Output : 2
Explanation : Below are triplets with sum less than 2
              (-2, 0, 1) and (-2, 0, 3)

Input : arr[] = {5, 1, 3, 4, 7}
        sum = 12.
Output : 4
Explanation : Below are triplets with sum less than 4
              (1, 3, 4), (1, 3, 5), (1, 3, 7) and
              (1, 4, 5)
```

We strongly recommend you to minimize your browser and try this yourself first.

A **Simple Solution** is to run three loops to consider all triplets one by one. For every triplet, compare the sums and increment count if triplet sum is smaller than given sum.

```
// A Simple C++ program to count triplets with sum smaller
// than a given value
#include<bits/stdc++.h>
using namespace std;

int countTriplets(int arr[], int n, int sum)
{
    // Initialize result
    int ans = 0;

    // Fix the first element as A[i]
    for (int i = 0; i < n-2; i++)
    {
        // Fix the second element as A[j]
        for (int j = i+1; j < n-1; j++)
        {
            // Now look for the third number
            for (int k = j+1; k < n; k++)
                if (arr[i] + arr[j] + arr[k] < sum)
                    ans++;
        }
    }

    return ans;
}

// Driver program
int main()
{
    int arr[] = {5, 1, 3, 4, 7};
    int n = sizeof arr / sizeof arr[0];
    int sum = 12;
    cout << countTriplets(arr, n, sum) << endl;
    return 0;
}
```

Output:

```
4
```

Time complexity of above solution is $O(n^3)$. An **Efficient Solution** can count triplets in $O(n^2)$ by sorting the array first, and then using method 1 of [this](#) post in a loop.

- 1) Sort the input array in increasing order.
- 2) Initialize result as 0.
- 3) Run a loop from $i = 0$ to $n-2$. An iteration of this loop finds all triplets with $arr[i]$ as first element.
 - a) Initialize other two elements as corner elements of subarray $arr[i+1..n-1]$, i.e., $j = i+1$ and $k = n-1$
 - b) Move j and k toward each other until they meet, i.e., while ($j = sum$), then do $k--$

```

// Else for current i and j, there can (k-j) possible third elements
// that satisfy the constraint.
(ii) Else Do ans += (k - j) followed by j++

```

Below is C++ implementation of above idea.

```

// C++ program to count triplets with sum smaller than a given value
#include<bits/stdc++.h>
using namespace std;

int countTriplets(int arr[], int n, int sum)
{
    // Sort input array
    sort(arr, arr+n);

    // Initialize result
    int ans = 0;

    // Every iteration of loop counts triplet with
    // first element as arr[i].
    for (int i = 0; i < n - 2; i++)
    {
        // Initialize other two elements as corner elements
        // of subarray arr[j+1..k]
        int j = i + 1, k = n - 1;

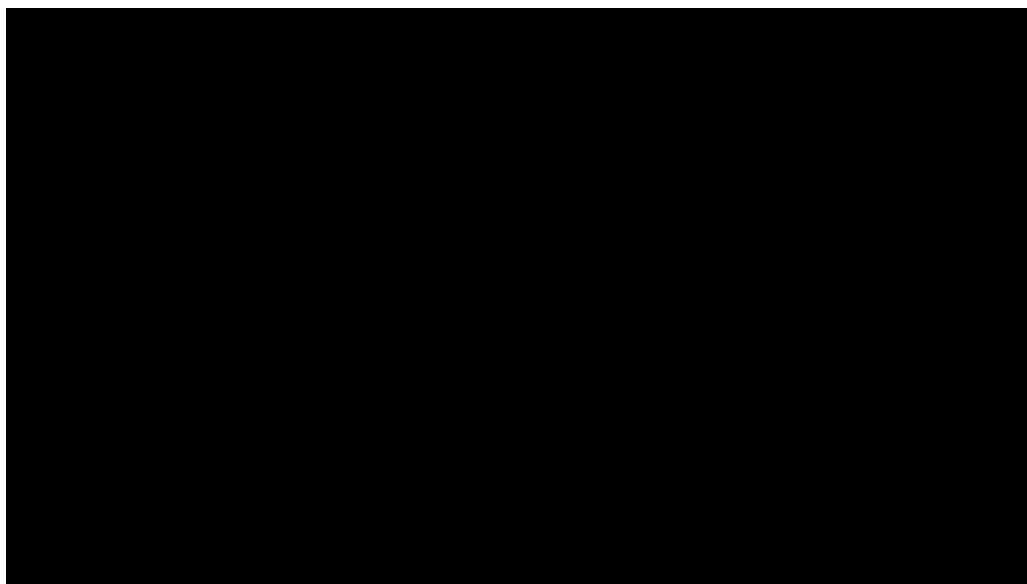
        // Use Meet in the Middle concept
        while (j < k)
        {
            // If sum of current triplet is more or equal,
            // move right corner to look for smaller values
            if (arr[i] + arr[j] + arr[k] >= sum)
                k--;

            // Else move left corner
            else
            {
                // This is important. For current i and j, there
                // can be total k-j third elements.
                ans += (k - j);
                j++;
            }
        }
    }
    return ans;
}

// Driver program
int main()
{
    int arr[] = {5, 1, 3, 4, 7};
    int n = sizeof arr / sizeof arr[0];
    int sum = 12;
    cout << countTriplets(arr, n, sum) << endl;
    return 0;
}

```

Output:



Thanks to [Gaurav Ahirwar](#) for suggesting this solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Arrays

Convert array into Zig-Zag fashion

Given an array of distinct elements, rearrange the elements of array in zig-zag fashion in O(n) time. The converted array should be in form a c e

Example:

Input: arr[] = {4, 3, 7, 8, 6, 2, 1}
Output: arr[] = {3, 7, 4, 8, 2, 6, 1}

Input: arr[] = {1, 4, 3, 2}
Output: arr[] = {1, 4, 2, 3}

A **Simple Solution** is to first sort the array. After sorting, exclude the first element, swap the remaining elements in pairs. (i.e. keep arr[0] as it is, swap arr[1] and arr[2], swap arr[3] and arr[4], and so on). Time complexity is O(nlogn) since we need to sort the array first.

We can convert in O(n) time using an **Efficient Approach**. The idea is to use modified one pass of bubble sort. Maintain a flag for representing which order(i.e.) currently we need. If the current two elements are not in that order then swap those elements otherwise not.

Let us see the main logic using three consecutive elements A, B, C. Suppose we are processing B and C currently and the current relation is 'C'. Since current relation is " i.e., A must be greater than B. So, the relation is A > B and B > C. We can deduce A > C. So if we swap B and C then the relation is A > C and C A C B

Refer [this](#) for more explanation.

Below is C++ implementation of above algorithm

```
// C++ program to sort an array in Zig-Zag form
#include <iostream>
using namespace std;

// Program for zig-zag conversion of array
void zigZag(int arr[], int n)
{
    // Flag true indicates relation "<" is expected,
    // else ">" is expected. The first expected relation
    // is "<"
    bool flag = true;

    for (int i=0; i<=n-2; i++)
    {
        if (flag) /* "<" relation expected */
        {
            /* If we have a situation like A > B > C,
               we get A > B < C by swapping B and C */
        }
    }
}
```

```

        if (arr[i] > arr[i+1])
            swap(arr[i], arr[i+1]);
    }
    else /* ">" relation expected */
    {
        /* If we have a situation like A < B < C,
           we get A < C > B by swapping B and C */
        if (arr[i] < arr[i+1])
            swap(arr[i], arr[i+1]);
    }
    flag = !flag; /* flip flag */
}
}

// Driver program
int main()
{
    int arr[] = {4, 3, 7, 8, 6, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    zigZag(arr, n);
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
    return 0;
}

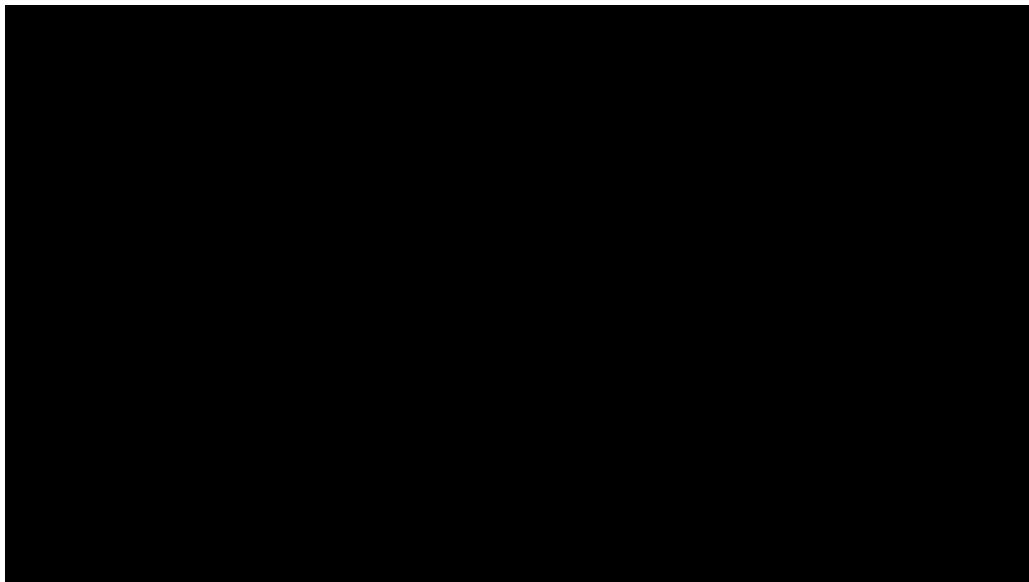
```

Output:

3 7 4 8 2 6 1

Time complexity: O(n)

Auxiliary Space: O(1)



This article is contributed by **Siva Krishna Aleti**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Arrays

Generate all possible sorted arrays from alternate elements of two given sorted arrays

Given two sorted arrays A and B, generate all possible arrays such that first element is taken from A then from B then from A and so on in increasing order till the arrays exhausted. The generated arrays should end with an element from B.

For Example

A = {10, 15, 25}
B = {1, 5, 20, 30}

The resulting arrays are:

```
10 20
10 20 25 30
10 30
15 20
15 20 25 30
15 30
25 30
```

Source: Microsoft Interview Question

We strongly recommend you to minimize your browser and try this yourself first.

The idea is to use recursion. In the recursive function, a flag is passed to indicate whether current element in output should be taken from 'A' or 'B'. Below is C++ implementation.

C

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

void printArr(int arr[], int n);

/* Function to generates and prints all sorted arrays from alternate elements of
'A[..m-1]' and 'B[..n-1]'
If 'flag' is true, then current element is to be included from A otherwise
from B.
'len' is the index in output array C[]. We print output array each time
before including a character from A only if length of output array is
greater than 0. We try than all possible combinations */
void generateUtil(int A[], int B[], int C[], int i, int j, int m, int n,
                  int len, bool flag)
{
    if (flag) // Include valid element from A
    {
        // Print output if there is at least one 'B' in output array 'C'
        if (len)
            printArr(C, len+1);

        // Recur for all elements of A after current index
        for (int k = i; k < m; k++)
        {
            if (!len)
            {
                /* this block works for the very first call to include
                 the first element in the output array */
                C[len] = A[k];

                // don't increment len as B is included yet
                generateUtil(A, B, C, k+1, j, m, n, len, !flag);
            }
            else /* include valid element from A and recur */
            {
                if (A[k] > C[len])
                {
                    C[len+1] = A[k];
                    generateUtil(A, B, C, k+1, j, m, n, len+1, !flag);
                }
            }
        }
    }
    else /* Include valid element from B and recur */
    {
        for (int l = j; l < n; l++)
        {
            if (B[l] > C[len])
            {
                C[len+1] = B[l];
                generateUtil(A, B, C, i, l+1, m, n, len+1, !flag);
            }
        }
    }
}

/* Wrapper function */
```

```

void generate(int A[], int B[], int m, int n)
{
    int C[m+n]; /* output array */
    generateUtil(A, B, C, 0, 0, m, n, 0, true);
}

// A utility function to print an array
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program
int main()
{
    int A[] = {10, 15, 25};
    int B[] = {5, 20, 30};
    int n = sizeof(A)/sizeof(A[0]);
    int m = sizeof(B)/sizeof(B[0]);
    generate(A, B, n, m);
    return 0;
}

```

Java

```

class GenerateArrays {

    /* Function to generates and prints all sorted arrays from alternate
     * elements of 'A[i..m-1]' and 'B[j..n-1]'
     * If 'flag' is true, then current element is to be included from A
     * otherwise from B.
     * 'len' is the index in output array C[]. We print output array
     * each time before including a character from A only if length of
     * output array is greater than 0. We try than all possible
     * combinations */

    void generateUtil(int A[], int B[], int C[], int i, int j, int m, int n,
                      int len, boolean flag)
    {
        if (flag) // Include valid element from A
        {
            // Print output if there is at least one 'B' in output array 'C'
            if (len != 0)
                printArr(C, len + 1);

            // Recur for all elements of A after current index
            for (int k = i; k < m; k++)
            {
                if (len == 0)
                {
                    /* this block works for the very first call to include
                     * the first element in the output array */
                    C[len] = A[k];

                    // don't increment len as B is included yet
                    generateUtil(A, B, C, k + 1, j, m, n, len, !flag);
                }

                /* include valid element from A and recur */
                else if (A[k] > C[len])
                {
                    C[len + 1] = A[k];
                    generateUtil(A, B, C, k + 1, j, m, n, len + 1, !flag);
                }
            }
        }

        /* Include valid element from B and recur */
        else
        {
            for (int l = j; l < n; l++)
            {

```

```

        if (B[i] > C[len])
        {
            C[len + 1] = B[i];
            generateUtil(A, B, C, i, i + 1, m, n, len + 1, !flag);
        }
    }
}

/* Wrapper function */
void generate(int A[], int B[], int m, int n)
{
    int C[] = new int[m + n];

    /* output array */
    generateUtil(A, B, C, 0, 0, m, n, 0, true);
}

// A utility function to print an array
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        System.out.print(arr[i] + " ");
    System.out.println("");
}

public static void main(String[] args)
{
    GenerateArrays generate = new GenerateArrays();
    int A[] = {10, 15, 25};
    int B[] = {5, 20, 30};
    int n = A.length;
    int m = B.length;
    generate.generate(A, B, n, m);
}
}

// This code has been contributed by Mayank Jaiswal

```

Output:

```

10 20
10 20 25 30
10 30
15 20
15 20 25 30
15 30
25 30

```

This article is contributed by [Gaurav Ahirwar](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above\

GATE CS Corner Company Wise Coding Practice

Arrays
Recursion

Pythagorean Triplet in an array

Given an array of integers, write a function that returns true if there is a triplet (a, b, c) that satisfies $a^2 + b^2 = c^2$.

Example:

```

Input: arr[] = {3, 1, 4, 6, 5}
Output: True
There is a Pythagorean triplet (3, 4, 5).

Input: arr[] = {10, 4, 6, 12, 5}
Output: False
There is no Pythagorean triplet.

```

We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

Method 1 (Naive)

A simple solution is to run three loops, three loops pick three array elements and check if current three elements form a Pythagorean Triplet.

Below is C++ implementation of simple solution.

C++

```
// A C++ program that returns true if there is a Pythagorean
// Triplet in a given array.
#include <iostream>
using namespace std;

// Returns true if there is Pythagorean triplet in ar[0..n-1]
bool isTriplet(int ar[], int n)
{
    for (int i=0; i<n; i++)
    {
        for (int j=i+1; j<n; j++)
        {
            for (int k=j+1; k<n; k++)
            {
                // Calculate square of array elements
                int x = ar[i]*ar[i], y = ar[j]*ar[j], z = ar[k]*ar[k];

                if (x == y + z || y == x + z || z == x + y)
                    return true;
            }
        }
    }

    // If we reach here, no triplet found
    return false;
}

/* Driver program to test above function */
int main()
{
    int ar[] = {3, 1, 4, 6, 5};
    int ar_size = sizeof(ar)/sizeof(ar[0]);
    isTriplet(ar, ar_size)? cout << "Yes": cout << "No";
    return 0;
}
```

Java

```
// A Java program that returns true if there is a Pythagorean
// Triplet in a given array.
import java.io.*;

class PythagoreanTriplet {

    // Returns true if there is Pythagorean triplet in ar[0..n-1]
    static boolean isTriplet(int ar[], int n)
    {
        for (int i=0; i<n; i++)
        {
            for (int j=i+1; j<n; j++)
            {
                for (int k=j+1; k<n; k++)
                {
                    // Calculate square of array elements
                    int x = ar[i]*ar[i], y = ar[j]*ar[j], z = ar[k]*ar[k];

                    if (x == y + z || y == x + z || z == x + y)
                        return true;
                }
            }
        }

        return false;
    }
}
```

```

    // If we reach here, no triplet found
    return false;
}

// Driver program to test above function
public static void main(String[] args)
{
    int ar[] = {3, 1, 4, 6, 5};
    int ar_size = ar.length;
    if(isTriplet(ar,ar_size)==true)
        System.out.println("Yes");
    else
        System.out.println("No");
}
/* This code is contributed by Devesh Agrawal */

```

Output:

Yes

Time Complexity of the above solution is $O(n^3)$.

Method 2 (Use Sorting)

We can solve this in $O(n^2)$ time by sorting the array first.

- 1) Do square of every element in input array. This step takes $O(n)$ time.
- 2) Sort the squared array in increasing order. This step takes $O(n \log n)$ time.
- 3) To find a triplet (a, b, c) such that $a = b + c$, do following.
 1. Fix 'a' as last element of sorted array.
 2. Now search for pair (b, c) in subarray between first element and 'a'. A pair (b, c) with given sum can be found in $O(n)$ time using meet in middle algorithm discussed in method 1 of [this](#) post.
 3. If no pair found for current 'a', then move 'a' one position back and repeat step 3.2.

Below is C++ implementation of above algorithm.

C++

```

// A C++ program that returns true if there is a Pythagorean
// Triplet in a given array.
#include <iostream>
#include <algorithm>
using namespace std;

// Returns true if there is a triplet with following property
// A[i]*A[i] = A[j]*A[j] + A[k]*A[k]
// Note that this function modifies given array
bool isTriplet(int arr[], int n)
{
    // Square array elements
    for (int i=0; i<n; i++)
        arr[i] = arr[i]*arr[i];

    // Sort array elements
    sort(arr, arr + n);

    // Now fix one element one by one and find the other two
    // elements
    for (int i = n-1; i >= 2; i--)
    {
        // To find the other two elements, start two index
        // variables from two corners of the array and move
        // them toward each other
        int l = 0; // index of the first element in arr[0..i-1]
        int r = i-1; // index of the last element in arr[0..i-1]
        while (l < r)

```

```

    {
        // A triplet found
        if (arr[l] + arr[r] == arr[i])
            return true;

        // Else either move 'l' or 'r'
        (arr[l] + arr[r] < arr[i])? l++: r--;
    }

    // If we reach here, then no triplet found
    return false;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {3, 1, 4, 6, 5};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    isTriplet(arr, arr_size)? cout << "Yes": cout << "No";
    return 0;
}

```

Java

```

// A Java program that returns true if there is a Pythagorean
// Triplet in a given array.
import java.io.*;
import java.util.*;

class PythagoreanTriplet
{
    // Returns true if there is a triplet with following property
    // A[i]*A[i] = A[j]*A[j] + A[k]*[k]
    // Note that this function modifies given array
    static boolean isTriplet(int arr[], int n)
    {
        // Square array elements
        for (int i=0; i<n; i++)
            arr[i] = arr[i]*arr[i];

        // Sort array elements
        Arrays.sort(arr);

        // Now fix one element one by one and find the other two
        // elements
        for (int i = n-1; i >= 2; i--)
        {
            // To find the other two elements, start two index
            // variables from two corners of the array and move
            // them toward each other
            int l = 0; // index of the first element in arr[0..i-1]
            int r = i-1; // index of the last element in arr[0..i-1]
            while (l < r)
            {
                // A triplet found
                if (arr[l] + arr[r] == arr[i])
                    return true;

                // Else either move 'l' or 'r'
                if (arr[l] + arr[r] < arr[i])
                    l++;
                else
                    r--;
            }
        }

        // If we reach here, then no triplet found
        return false;
    }

    // Driver program to test above function
}

```

```

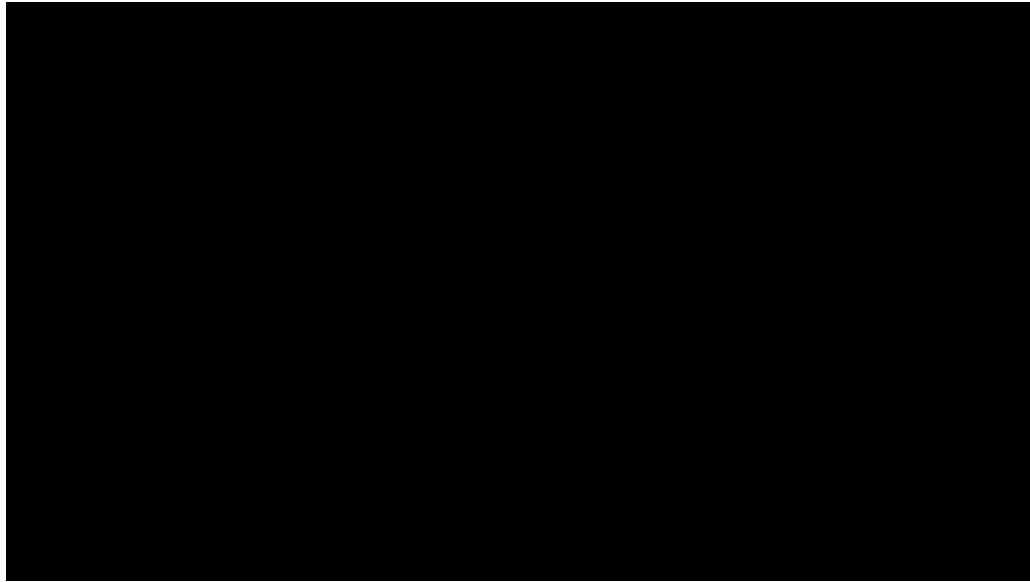
public static void main(String[] args)
{
    int arr[] = {3, 1, 4, 6, 5};
    int arr_size = arr.length;
    if (isTriplet(arr,arr_size)==true)
        System.out.println("Yes");
    else
        System.out.println("No");
}
/*This code is contributed by Devesh Agrawal*/

```

Output:

Yes

Time complexity of this method is $O(n^2)$.



This article is contributed by **Harshit Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Arrays

Length of the largest subarray with contiguous elements | Set 1

Given an array of distinct integers, find length of the longest subarray which contains numbers that can be arranged in a continuous sequence.

Examples:

Input: arr[] = {10, 12, 11};
Output: Length of the longest contiguous subarray is 3

Input: arr[] = {14, 12, 11, 20};
Output: Length of the longest contiguous subarray is 2

Input: arr[] = {1, 56, 58, 57, 90, 92, 94, 93, 91, 45};
Output: Length of the longest contiguous subarray is 5

We strongly recommend to minimize the browser and try this yourself first.

The important thing to note in question is, it is given that all elements are distinct. If all elements are distinct, then a subarray has contiguous elements if and only if the difference between maximum and minimum elements in subarray is equal to the difference between last and first indexes of subarray. So the idea is to keep track of minimum and maximum element in every subarray.

The following is the implementation of above idea.

C++

```

#include<iostream>
using namespace std;

// Utility functions to find minimum and maximum of
// two elements
int min(int x, int y) { return (x < y)? x : y; }
int max(int x, int y) { return (x > y)? x : y; }

// Returns length of the longest contiguous subarray
int findLength(int arr[], int n)
{
    int max_len = 1; // Initialize result
    for (int i=0; i<n-1; i++)
    {
        // Initialize min and max for all subarrays starting with i
        int mn = arr[i], mx = arr[i];

        // Consider all subarrays starting with i and ending with j
        for (int j=i+1; j<n; j++)
        {
            // Update min and max in this subarray if needed
            mn = min(mn, arr[j]);
            mx = max(mx, arr[j]);

            // If current subarray has all contiguous elements
            if ((mx - mn) == j-i)
                max_len = max(max_len, mx-mn+1);
        }
    }
    return max_len; // Return result
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 56, 58, 57, 90, 92, 94, 93, 91, 45};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Length of the longest contiguous subarray is "
         << findLength(arr, n);
    return 0;
}

```

Java

```

class LargestSubArray2
{
    // Utility functions to find minimum and maximum of
    // two elements

    int min(int x, int y)
    {
        return (x < y) ? x : y;
    }

    int max(int x, int y)
    {
        return (x > y) ? x : y;
    }

    // Returns length of the longest contiguous subarray
    int findLength(int arr[], int n)
    {
        int max_len = 1; // Initialize result
        for (int i = 0; i < n - 1; i++)
        {
            // Initialize min and max for all subarrays starting with i
            int mn = arr[i], mx = arr[i];

            // Consider all subarrays starting with i and ending with j
            for (int j = i + 1; j < n; j++)
            {
                // Update min and max in this subarray if needed
                mn = min(mn, arr[j]);
            }
        }
    }
}

```

```

mx = max(mx, arr[i]);

// If current subarray has all contiguous elements
if ((mx - mn) == j - i)
    max_len = max(max_len, mx - mn + 1);
}

return max_len; // Return result
}

public static void main(String[] args)
{
    LargestSubArray2 large = new LargestSubArray2();
    int arr[] = {1, 56, 58, 57, 90, 92, 94, 93, 91, 45};
    int n = arr.length;
    System.out.println("Length of the longest contiguous subarray is "
        + large.findLength(arr, n));
}
}

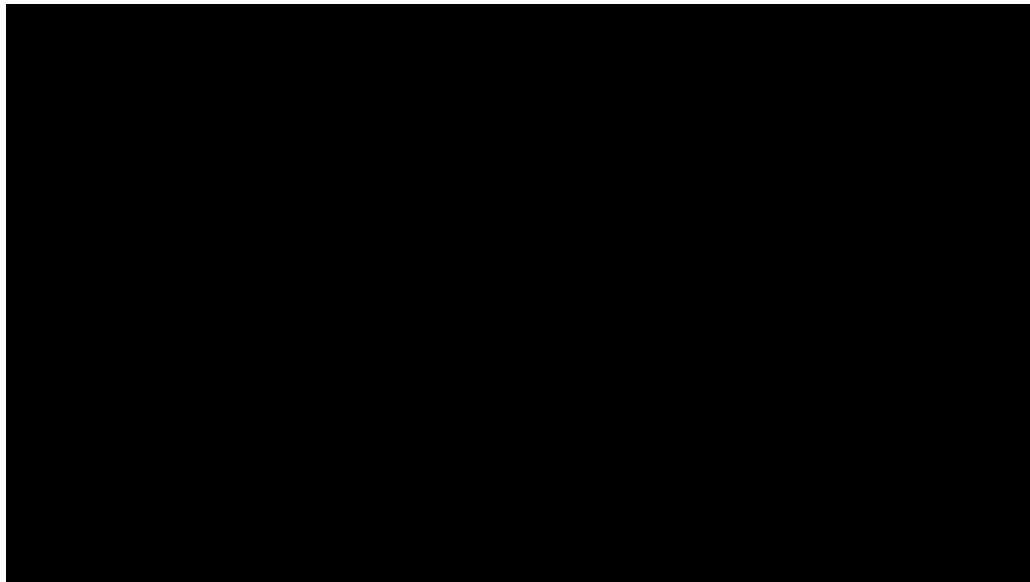
// This code has been contributed by Mayank Jaiswal

```

Output:

Length of the longest contiguous subarray is 5

Time Complexity of the above solution is $O(n^2)$.



We will soon be covering solution for the problem where duplicate elements are allowed in subarray.

[Length of the largest subarray with contiguous elements | Set 2](#)

This article is contributed by **Arjun**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Arrays

Find the smallest positive integer value that cannot be represented as sum of any subset of a given array

Given a sorted array (sorted in non-decreasing order) of positive numbers, find the smallest positive integer value that cannot be represented as sum of elements of any subset of given set.

Expected time complexity is $O(n)$.

Examples:

Input: arr[] = {1, 3, 6, 10, 11, 15};
Output: 2

```
Input: arr[] = {1, 1, 1, 1};  
Output: 5
```

```
Input: arr[] = {1, 1, 3, 4};  
Output: 10
```

```
Input: arr[] = {1, 2, 5, 10, 20, 40};  
Output: 4
```

```
Input: arr[] = {1, 2, 3, 4, 5, 6};  
Output: 22
```

We strongly recommend to minimize the browser and try this yourself first.

A **Simple Solution** is to start from value 1 and check all values one by one if they can sum to values in the given array. This solution is very inefficient as it reduces to [subset sum problem](#) which is a well known [NP Complete Problem](#).

We can solve this problem **in O(n) time** using a simple loop. Let the input array be arr[0..n-1]. We initialize the result as 1 (smallest possible outcome) and traverse the given array. Let the smallest element that cannot be represented by elements at indexes from 0 to (i-1) be 'res', there are following two possibilities when we consider element at index i:

1) We decide that 'res' is the final result: If arr[i] is greater than 'res', then we found the gap which is 'res' because the elements after arr[i] are also going to be greater than 'res'.

2) The value of 'res' is incremented after considering arr[i]: The value of 'res' is incremented by arr[i] (why? If elements from 0 to (i-1) can represent 1 to 'res-1', then elements from 0 to i can represent from 1 to 'res + arr[i] - 1' by adding 'arr[i]' to all subsets that represent 1 to 'res')

Following is the implementation of above idea.

C++

```
// C++ program to find the smallest positive value that cannot be  
// represented as sum of subsets of a given sorted array  
#include <iostream>  
using namespace std;  
  
// Returns the smallest number that cannot be represented as sum  
// of subset of elements from set represented by sorted array arr[0..n-1]  
int findSmallest(int arr[], int n)  
{  
    int res = 1; // Initialize result  
  
    // Traverse the array and increment 'res' if arr[i] is  
    // smaller than or equal to 'res'.  
    for (int i = 0; i < n && arr[i] <= res; i++)  
        res = res + arr[i];  
  
    return res;  
}  
  
// Driver program to test above function  
int main()  
{  
    int arr1[] = {1, 3, 4, 5};  
    int n1 = sizeof(arr1)/sizeof(arr1[0]);  
    cout << findSmallest(arr1, n1) << endl;  
  
    int arr2[] = {1, 2, 6, 10, 11, 15};  
    int n2 = sizeof(arr2)/sizeof(arr2[0]);  
    cout << findSmallest(arr2, n2) << endl;  
  
    int arr3[] = {1, 1, 1, 1};  
    int n3 = sizeof(arr3)/sizeof(arr3[0]);  
    cout << findSmallest(arr3, n3) << endl;  
  
    int arr4[] = {1, 1, 3, 4};  
    int n4 = sizeof(arr4)/sizeof(arr4[0]);  
    cout << findSmallest(arr4, n4) << endl;  
  
    return 0;  
}
```

Java

```
// Java program to find the smallest positive value that cannot be
// represented as sum of subsets of a given sorted array
class FindSmallestInteger
{
    // Returns the smallest number that cannot be represented as sum
    // of subset of elements from set represented by sorted array arr[0..n-1]
    int findSmallest(int arr[], int n)
    {
        int res = 1; // Initialize result

        // Traverse the array and increment 'res' if arr[i] is
        // smaller than or equal to 'res'.
        for (int i = 0; i < n && arr[i] <= res; i++)
            res = res + arr[i];

        return res;
    }

    // Driver program to test above functions
    public static void main(String[] args)
    {
        FindSmallestInteger small = new FindSmallestInteger();
        int arr1[] = {1, 3, 4, 5};
        int n1 = arr1.length;
        System.out.println(small.findSmallest(arr1, n1));

        int arr2[] = {1, 2, 6, 10, 11, 15};
        int n2 = arr2.length;
        System.out.println(small.findSmallest(arr2, n2));

        int arr3[] = {1, 1, 1, 1};
        int n3 = arr3.length;
        System.out.println(small.findSmallest(arr3, n3));

        int arr4[] = {1, 1, 3, 4};
        int n4 = arr4.length;
        System.out.println(small.findSmallest(arr4, n4));
    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

```
2
4
5
10
```

Time Complexity of above program is O(n).

This article is contributed by **Rahul Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Arrays

Smallest subarray with sum greater than a given value

Given an array of integers and a number x, find the smallest subarray with sum greater than the given value.

Examples:

```
arr[] = {1, 4, 45, 6, 0, 19}
x = 51
```

Output: 3

Minimum length subarray is {4, 45, 6}

```

arr[] = {1, 10, 5, 2, 7}
x = 9
Output: 1
Minimum length subarray is {10}

arr[] = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250}
x = 280
Output: 4
Minimum length subarray is {100, 1, 0, 200}

```

A **simple solution** is to use two nested loops. The outer loop picks a starting element, the inner loop considers all elements (on right side of current start) as ending element. Whenever sum of elements between current start and end becomes more than the given number, update the result if current length is smaller than the smallest length so far.

Following is the implementation of simple approach.

C

```

#include <iostream>
using namespace std;

// Returns length of smallest subarray with sum greater than x.
// If there is no subarray with given sum, then returns n+1
int smallestSubWithSum(int arr[], int n, int x)
{
    // Initialize length of smallest subarray as n+1
    int min_len = n + 1;

    // Pick every element as starting point
    for (int start=0; start<n; start++)
    {
        // Initialize sum starting with current start
        int curr_sum = arr[start];

        // If first element itself is greater
        if (curr_sum > x) return 1;

        // Try different ending points for current start
        for (int end=start+1; end<n; end++)
        {
            // add last element to current sum
            curr_sum += arr[end];

            // If sum becomes more than x and length of
            // this subarray is smaller than current smallest
            // length, update the smallest length (or result)
            if (curr_sum > x && (end - start + 1) < min_len)
                min_len = (end - start + 1);
        }
    }
    return min_len;
}

/* Driver program to test above function */
int main()
{
    int arr1[] = {1, 4, 45, 6, 10, 19};
    int x = 51;
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    cout << smallestSubWithSum(arr1, n1, x) << endl;

    int arr2[] = {1, 10, 5, 2, 7};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    x = 9;
    cout << smallestSubWithSum(arr2, n2, x) << endl;

    int arr3[] = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250};
    int n3 = sizeof(arr3)/sizeof(arr3[0]);
    x = 280;
    cout << smallestSubWithSum(arr3, n3, x) << endl;

    return 0;
}

```

Java

```
class SmallestSubArraySum
{
    // Returns length of smallest subarray with sum greater than x.
    // If there is no subarray with given sum, then returns n+1
    int smallestSubWithSum(int arr[], int n, int x)
    {
        // Initialize length of smallest subarray as n+1
        int min_len = n + 1;

        // Pick every element as starting point
        for (int start = 0; start < n; start++)
        {
            // Initialize sum starting with current start
            int curr_sum = arr[start];

            // If first element itself is greater
            if (curr_sum > x)
                return 1;

            // Try different ending points for current start
            for (int end = start + 1; end < n; end++)
            {
                // add last element to current sum
                curr_sum += arr[end];

                // If sum becomes more than x and length of
                // this subarray is smaller than current smallest
                // length, update the smallest length (or result)
                if (curr_sum > x && (end - start + 1) < min_len)
                    min_len = (end - start + 1);
            }
        }
        return min_len;
    }

    // Driver program to test above functions
    public static void main(String[] args)
    {
        SmallestSubArraySum array_sum = new SmallestSubArraySum();
        int arr1[] = {1, 4, 45, 6, 10, 19};
        int x = 51;
        int n1 = arr1.length;
        System.out.println(array_sum.smallestSubWithSum(arr1, n1, x));

        int arr2[] = {1, 10, 5, 2, 7};
        int n2 = arr2.length;
        x = 9;
        System.out.println(array_sum.smallestSubWithSum(arr2, n2, x));

        int arr3[] = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250};
        int n3 = arr3.length;
        x = 280;
        System.out.println(array_sum.smallestSubWithSum(arr3, n3, x));
    }
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
3
1
4
```

Time Complexity: Time complexity of the above approach is clearly $O(n^2)$.

Efficient Solution: This problem can be solved in **$O(n)$ time** using the idea used in [this post](#). Thanks to Ankit and Nitin for suggesting this optimized solution.

C++

```

// O(n) solution for finding smallest subarray with sum
// greater than x
#include <iostream>
using namespace std;

// Returns length of smallest subarray with sum greater than x.
// If there is no subarray with given sum, then returns n+1
int smallestSubWithSum(int arr[], int n, int x)
{
    // Initialize current sum and minimum length
    int curr_sum = 0, min_len = n+1;

    // Initialize starting and ending indexes
    int start = 0, end = 0;
    while (end < n)
    {
        // Keep adding array elements while current sum
        // is smaller than x
        while (curr_sum <= x && end < n)
            curr_sum += arr[end++];

        // If current sum becomes greater than x.
        while (curr_sum > x && start < n)
        {
            // Update minimum length if needed
            if (end - start < min_len)
                min_len = end - start;

            // remove starting elements
            curr_sum -= arr[start++];
        }
    }
    return min_len;
}

/* Driver program to test above function */
int main()
{
    int arr1[] = {1, 4, 45, 6, 10, 19};
    int x = 51;
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    cout << smallestSubWithSum(arr1, n1, x) << endl;

    int arr2[] = {1, 10, 5, 2, 7};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    x = 9;
    cout << smallestSubWithSum(arr2, n2, x) << endl;

    int arr3[] = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250};
    int n3 = sizeof(arr3)/sizeof(arr3[0]);
    x = 280;
    cout << smallestSubWithSum(arr3, n3, x);

    return 0;
}

```

Java

```

// O(n) solution for finding smallest subarray with sum
// greater than x

class SmallestSubArraySum
{
    // Returns length of smallest subarray with sum greater than x.
    // If there is no subarray with given sum, then returns n+1
    int smallestSubWithSum(int arr[], int n, int x)
    {
        // Initialize current sum and minimum length
        int curr_sum = 0, min_len = n + 1;

        // Initialize starting and ending indexes
        int start = 0, end = 0;
        while (end < n)

```

```

    {
        // Keep adding array elements while current sum
        // is smaller than x
        while (curr_sum <= x && end < n)
            curr_sum += arr[end++];

        // If current sum becomes greater than x.
        while (curr_sum > x && start < n)
        {
            // Update minimum length if needed
            if (end - start < min_len)
                min_len = end - start;

            // remove starting elements
            curr_sum -= arr[start++];
        }
    }

    return min_len;
}

// Driver program to test above functions
public static void main(String[] args) {
    SmallestSubArraySum array_sum = new SmallestSubArraySum();
    int arr1[] = {1, 4, 45, 6, 10, 19};
    int x = 51;
    int n1 = arr1.length;
    System.out.println(array_sum.smallestSubWithSum(arr1, n1, x));

    int arr2[] = {1, 10, 5, 2, 7};
    int n2 = arr2.length;
    x = 9;
    System.out.println(array_sum.smallestSubWithSum(arr2, n2, x));

    int arr3[] = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250};
    int n3 = arr3.length;
    x = 280;
    System.out.println(array_sum.smallestSubWithSum(arr3, n3, x));
}
}

// This code has been contributed by Mayank Jaiswal

```

Output:

```

3
1
4

```

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Arrays

Stock Buy Sell to Maximize Profit

The cost of a stock on each day is given in an array, find the max profit that you can make by buying and selling in those days. For example, if the given array is {100, 180, 260, 310, 40, 535, 695}, the maximum profit can be earned by buying on day 0, selling on day 3. Again buy on day 4 and sell on day 6. If the given array of prices is sorted in decreasing order, then profit cannot be earned at all.

If we are allowed to buy and sell only once, then we can use following algorithm. [Maximum difference between two elements](#). Here we are allowed to buy and sell multiple times. Following is algorithm for this problem.

1. Find the local minima and store it as starting index. If not exists, return.
2. Find the local maxima. and store it as ending index. If we reach the end, set the end as ending index.
3. Update the solution (Increment count of buy sell pairs)
4. Repeat the above steps if end is not reached.

```

// Program to find best buying and selling days
#include <stdio.h>

// solution structure
struct Interval
{
    int buy;
    int sell;
};

// This function finds the buy sell schedule for maximum profit
void stockBuySell(int price[], int n)
{
    // Prices must be given for at least two days
    if (n == 1)
        return;

    int count = 0; // count of solution pairs

    // solution vector
    Interval sol[n/2 + 1];

    // Traverse through given price array
    int i = 0;
    while (i < n-1)
    {
        // Find Local Minima. Note that the limit is (n-2) as we are
        // comparing present element to the next element.
        while ((i < n-1) && (price[i+1] <= price[i]))
            i++;

        // If we reached the end, break as no further solution possible
        if (i == n-1)
            break;

        // Store the index of minima
        sol[count].buy = i++;

        // Find Local Maxima. Note that the limit is (n-1) as we are
        // comparing to previous element
        while ((i < n) && (price[i] >= price[i-1]))
            i++;

        // Store the index of maxima
        sol[count].sell = i-1;

        // Increment count of buy/sell pairs
        count++;
    }

    // print solution
    if (count == 0)
        printf("There is no day when buying the stock will make profit\n");
    else
    {
        for (int i = 0; i < count; i++)
            printf("Buy on day: %d\n Sell on day: %d\n", sol[i].buy, sol[i].sell);
    }

    return;
}

// Driver program to test above functions
int main()
{
    // stock prices on consecutive days
    int price[] = {100, 180, 260, 310, 40, 535, 695};
    int n = sizeof(price)/sizeof(price[0]);

    // fucntion call
    stockBuySell(price, n);

    return 0;
}

```

```
}
```

Java

```
// Program to find best buying and selling days
import java.util.ArrayList;

// Solution structure
class Interval
{
    int buy, sell;
}

class StockBuySell
{
    // This function finds the buy sell schedule for maximum profit
    void stockBuySell(int price[], int n)
    {
        // Prices must be given for at least two days
        if (n == 1)
            return;

        int count = 0;

        // solution array
        ArrayList<Interval> sol = new ArrayList<Interval>();

        // Traverse through given price array
        int i = 0;
        while (i < n - 1)
        {
            // Find Local Minima. Note that the limit is (n-2) as we are
            // comparing present element to the next element.
            while ((i < n - 1) && (price[i + 1] <= price[i]))
                i++;

            // If we reached the end, break as no further solution possible
            if (i == n - 1)
                break;

            Interval e = new Interval();
            e.buy = i++;
            // Store the index of minima

            // Find Local Maxima. Note that the limit is (n-1) as we are
            // comparing to previous element
            while ((i < n) && (price[i] >= price[i - 1]))
                i++;

            // Store the index of maxima
            e.sell = i-1;
            sol.add(e);

            // Increment number of buy/sell
            count++;
        }

        // print solution
        if (count == 0)
            System.out.println("There is no day when buying the stock "
                + "will make profit");
        else
            for (int j = 0; j < count; j++)
                System.out.println("Buy on day: " + sol.get(j).buy
                    + "      " + "Sell on day : " + sol.get(j).sell);

        return;
    }

    public static void main(String args[])
    {
        StockBuySell stock = new StockBuySell();
```

```

// stock prices on consecutive days
int price[] = {100, 180, 260, 310, 40, 535, 695};

int n = price.length;

// function call
stock.stockBuySell(price, n);
}

// This code has been contributed by Mayank Jaiswal

```

Output:

```

Buy on day : 0 Sell on day: 3
Buy on day : 4 Sell on day: 6

```

Time Complexity: The outer loop runs till i becomes n-1. The inner two loops increment value of i in every iteration. So overall time complexity is O(n)

This article is compiled by [Ashish Anand](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

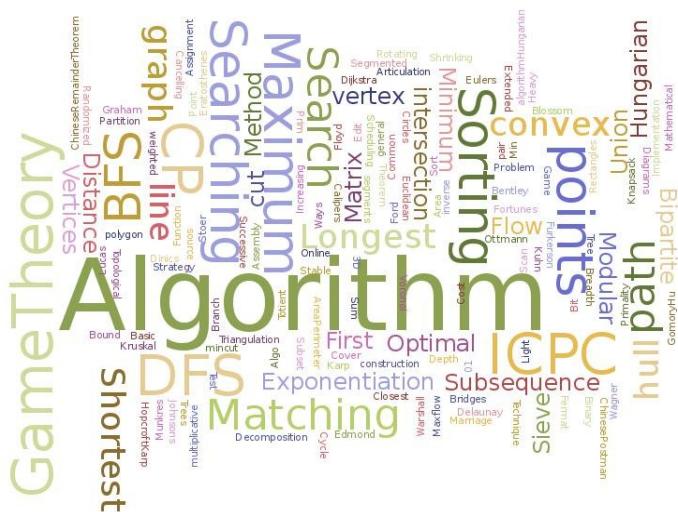
Arrays

Top 10 Algorithms and Data Structures for Competitive Programming

In this post “Important top 10 algorithms and data structures for competitive coding”.

Topics :

1. [Graph algorithms](#)
2. [Dynamic programming](#)
3. [Searching and Sorting:](#)
4. [Number theory and Other Mathematical](#)
5. [Geometrical and Network Flow Algorithms](#)
6. [Data Structures](#)



The below links cover all most important algorithms and data structure topics:

Graph Algorithms

1. Breadth First Search (BFS)
2. Depth First Search (DFS)
3. Shortest Path from source to all vertices **Dijkstra**

4. Shortest Path from every vertex to every other vertex **Floyd Warshall**
5. Minimum Spanning tree **Prim**
6. Minimum Spanning tree **Kruskal**
7. Topological Sort
8. Johnson's algorithm
9. Articulation Points (or Cut Vertices) in a Graph
10. Bridges in a graph

All Graph Algorithms

Dynamic Programming

1. Longest Common Subsequence
2. Longest Increasing Subsequence
3. Edit Distance
4. Minimum Partition
5. Ways to Cover a Distance
6. Longest Path In Matrix
7. Subset Sum Problem
8. Optimal Strategy for a Game
9. 0-1 Knapsack Problem
10. Assembly Line Scheduling

All DP Algorithms

Searching And Sorting

1. Binary Search
2. Quick Sort
3. Merge Sort
4. Order Statistics
5. KMP algorithm
6. Rabin karp
7. Z's algorithm
8. Aho Corasick String Matching
9. Counting Sort
10. Manacher's algorithm: [Part 1](#), [Part 2](#) and [Part 3](#)

All Articles on [Searching](#), [Sorting](#) and [Pattern Searching](#).

Number theory and Other Mathematical

Prime Numbers and Prime Factorization

1. Primality Test | Set 1 (Introduction and School Method)
2. Primality Test | Set 2 (Fermat Method)
3. Primality Test | Set 3 (Miller–Rabin)
4. Sieve of Eratosthenes
5. Segmented Sieve
6. Wilson's Theorem
7. Prime Factorisation
8. Pollard's rho algorithm

Modulo Arithmetic Algorithms

1. Basic and Extended Euclidean algorithms
2. Euler's Totient Function
3. Modular Exponentiation
4. Modular Multiplicative Inverse

5. Chinese remainder theorem Introduction
6. Chinese remainder theorem and Modulo Inverse Implementation
7. $nCr \% m$ and this.

Miscellaneous:

1. Counting Inversions
2. Counting Inversions using BIT
3. logarithmic exponentiation
4. Square root of an integer
5. Heavy light Decomposition , this and this
6. Matrix Rank
7. Gaussian Elimination to Solve Linear Equations
8. Hungarian algorithm
9. Link cut
10. Mo's algorithm and this
11. Factorial of a large number in C++
12. Factorial of a large number in Java+
13. Russian Peasant Multiplication
14. Catalan Number

[All Articles on Mathematical Algorithms](#)

Geometrical and Network Flow Algorithms

1. Convex Hull
2. Graham Scan
3. Line Intersection
4. Interval Tree
5. Matrix Exponentiation and this
6. Maxflow Ford Fulkerson Algo and Edmond Karp Implementation
7. Min cut
8. Stable Marriage Problem
9. Hopcroft–Karp Algorithm for Maximum Matching
10. Dinic's algo and e-maxx

[All Articles on Geometric Algorithms](#)

Data Structures

1. Binary Indexed Tree or Fenwick tree
2. Segment Tree (RMQ, Range Sum and Lazy Propagation)
3. K-D tree (See insert, minimum and delete)
4. Union Find Disjoint Set (Cycle Detection and By Rank and Path Compression)
5. Tries
6. Suffix array (this, this and this)
7. Sparse table
8. Suffix automata
9. Suffix automata II
10. LCA and RMQ

[All Articles on Advanced Data Structures.](#)

How to Begin?

Please see [How to begin with Competitive Programming?](#)

How to Practice?

Please see <http://practice.geeksforgeeks.org/>

What are top algorithms in Interview Questions?

[Top 10 algorithms in Interview Questions](#)

How to prepare for ACM – ICPC?

[How to prepare for ACM – ICPC?](#)

This is an initial draft. We will soon be adding more links and algorithms to this post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Competitive Programming
BFS
Binary-Indexed-Tree
BIT
DFS
modular-arithmetic
Segment-Tree
sieve
Topological Sorting

Interview Experiences

- Amazon
- Microsoft
- Google
- Yahoo
- Cisco
- Morgan Stanley
- Directi
- Works Applications Co., Ltd. Japan
- Adobe
- Facebook
- Flipkart
- Nvidia
- Qualcomm
- DELL
- Oracle
- Arista Network
- SAP Labs
- Goldman Sachs
- IBM IRL
- Yatra.com
- Drishti Soft Solutions
- Twitter
- Open Solutions
- Fiberlink (maas360)
- D E Shaw
- VMWare
- EA – A Gamer's Paradise
- Fab.com
- HCL
- GS-Labs
- Symantec
- Groupon
- Myntra
- Cadence India
- Bharti SoftBank (HIKE)
- Belzabar
- Expedia
- Headstrong
- SnapDeal
- Samsung
- One97 (Paytm)
- Paxcel Technologies
- Citrix
- Accolite
- Payu
- Aricent
- Intel
- Rockwell Collins
- Inmobi
- Sandisk Corporation
- MAQ Software
- Tower Research LLC
- Athena Health
- Amadeus
- Akamai
- U2opia Mobile
- CommVault Systems
- Linkedin
- Informatica
- Chalk Studio
- Service Now
- BrowserStack
- Nagarro
- MakeMyTrip
- Indus Valley Partners
- IBM-ISL
- Citicorp
- TCS
- ITC Infotech
- Citius Tech
- Wipro
- InfoEdge
- Sapient Sapient
- Quikr
- Visa
- Subex
- Practo
- Gramener
- Knowlarity
- Huawei
- Synopsys
- One Convergence Device
- Red Hat
- IIT Delhi MS(R)
- Polycom
- UHG(United Health Group)
- Brocade
- Housing.com
- Qubole
- Global Analytics
- Prop Tiger
- Target Corporation
- Walmart Lab
- CouponDunia

- Intuit
- Zoho
- Paypal
- ChargeBee
- WOW Labz
- codenation
- Xome
- Factset
- Oxigen Wallet
- Opera
- Teradata
- Infinera
- Salesforce
- Calsoft
- HP R&D
- Aspiring Minds
- Bidgely
- Mahindra Comviva
- National Instruments
- Zoomcar
- EXL Analytics
- ChargeBee
- Kronos
- Goibibo
- Carwale
- Bloomberg
- ScaleGrid
- Akosha
- Pubmatic
- Birst India
- Vizury Interactive Solutions
- AXIO-NET
- Monotype Solutions
- Komli Media
- Zycus
- PSTakeCare
- Aristocrat Gaming
- Barclays
- Tismo
- Pricewatercoopers(Pwc)
- Veritas
- Riverbed
- Wissen
- Thorogood
- Accenture
- Deloitte
- Shopclues
- BankBazaar
- Ola Cabs
- Grofers
- Endurance
- Epic Systems
- [24]7 Innovation Labs
- Axria
- Moonfrog Labs
- Swiggy
- Thoughtworks
- Wooquer
- Numerify
- Infosys
- Verifone
- enStage
- TinyOwl
- Baracuda Networks
- Ibibo
- WOW Labz
- CommonVault
- Alcatel Lucent
- Zopper.com
- Delhivery
- Palantir Technologies
- MetLife
- Taxi4Sure
- LensKart
- Commonfloor
- Compro Technologies
- Juniper Networks
- Persistent System
- Rockstand.in
- Philips
- NTT-DATA
- Times Internet
- Eze Software
- Societe General
- Amdocs
- HSBC
- Tolexo(IndiaMart company)
- Avaya
- Kuliza
- Financial Software Systems
- Cerner
- Uurmi
- Rivigo

Latest Interview Experiences

Placement Preparation Course

Share Your Questions/Experience

To share your interview experience, please mail your interview experience to contribute@geeksforgeeks.org. To share interview questions, please add questions at Geeksforgeeks Q&A.

Interview Questions

- Amazon
- Microsoft
- Google

Latest Interview Questions

Common Interview Questions

- Common Interview Puzzles
- Top 10 algorithms in Interview Questions
- Amazon's most asked interview questions
- Microsoft's most asked interview questions
- Accenture's most asked Interview Questions
- Commonly Asked OOP Interview Questions
- Commonly Asked C Programming Interview Questions | Set 1
- Commonly Asked C Programming Interview Questions | Set 2
- Commonly Asked C++ Interview Questions,
- Commonly Asked Java Programming Interview Questions | Set 1
- Commonly Asked Java Programming Interview Questions | Set 2
- Commonly asked DBMS interview questions | Set 1
- Commonly Asked Operating Systems Interview Questions | Set 1
- Commonly Asked Data Structure Interview Questions
- Commonly Asked Algorithm Interview Questions
- Commonly asked Computer Networks Interview Questions

See Books for recommended books on interview preparation.

Company Wise Coding Practice Topic Wise Coding Practice

All about GATE CS Preparation for 2017 aspirants. The page contains solutions of previous year GATE CS papers with explanations, topic wise Quizzes, notes/tutorials and important links for preparation.

GATE CS MOCK 2017

GATE CS Notes/Tutorials (According to Official GATE 2017 Syllabus)

Previous Years' questions/answers/explanation for GATE CS

- GATE-CS-2017 (Set 1)
- GATE-CS-2017 (Set 2)
- GATE-CS-2016 (Set 1)
- GATE-CS-2016 (Set 2)
- GATE-CS-2015 (Set 1)
- GATE-CS-2015 (Set 2)
- GATE-CS-2015 (Set 3)
- GATE-CS-2014-(Set-1)
- GATE-CS-2014-(Set-2)
- GATE-CS-2014-(Set-3)
- GATE CS 2013
- GATE CS 2012
- GATE CS 2011
- GATE CS 2010
- GATE-CS-2009
- GATE CS 2008
- GATE-CS-2007
- GATE-CS-2006
- GATE-CS-2005
- GATE-CS-2004
- GATE-CS-2003
- GATE-CS-2002
- GATE-CS-2001
- GATE-CS-2000

Previous Years' questions/answers/explanation for GATE IT

- GATE-IT-2004
- GATE-IT-2006
- GATE-IT-2008
- GATE-IT-2005
- GATE-IT-2007

Topic-wise Mock Quizzes for GATE CS

Data Structures and Algorithms

- Linked List
- Stack
- Queue
- Binary Trees
- Binary Search Trees

DBMS

- ER and Relational Models
- Database Design (Normal Forms)
- SQL
- Transactions and concurrency control

Balanced Binary Search Trees

Graph

Hash

Array

Misc

B and B+ Trees

Heap

Tree Traversals

Analysis of Algorithms

Sorting

Divide and Conquer

Greedy Algorithms

Dynamic Programming

Backtracking

Misc

NP Complete

Searching

Analysis of Algorithms (Recurrences)

Recursion

Bit Algorithms

Graph Traversals

Graph Shortest Paths

Graph Minimum Spanning Tree

Data Structures and Algorithm

C Language

Operating Systems

Operating System Notes

Process Management

CPU Scheduling

Memory Management

Input Output Systems

Operating Systems

Computer Organization and Architecture

Digital Logic & Number representation(28)

Computer Organization and Architecture(33)

Sample GATE Mock Test

Sequential files, indexing, B & B+ trees

Database Management Systems

Compiler Design

Lexical analysis

Parsing and Syntax directed translation

Code Generation and Optimization

Compiler Design

Computer Networks

Data Link Layer

Network Layer

Transport Layer

Misc Topics in Computer Networks

Application Layer

Network Security

Computer Networks

Theory of Computation

Regular languages and finite automata

Context free languages and Push-down automata

Recursively enumerable sets and Turing machines

Undecidability

Automata Theory

Aptitude

Probability

English

General Aptitude

Engineering Mathematics

Set Theory & Algebra

Linear Algebra

Numerical Methods and Calculus

Graph Theory

Combinatorics

Propositional and First Order Logic

Important Links:

[Solutions of GATE CS 2016 Mock Test](#)

[Top 5 Topics for for Section of GATE CS Syllabus](#)

[How to prepare in Last 10 days for GATE?](#)

[GATE CS Topic wise External Reference Links](#)

[Previous Year GATE Official Question Papers](#)

[Last Minute Notes](#)

[GATE CS 2016 Official Papers](#)

[GATE CS 2017 Dates](#)

[GATE CS 2017 Syllabus](#)

Please write comments if you find anything incorrect or wish to share more information for GATE CS preparation.

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

[Load Comments](#)

k largest(or smallest) elements in an array | added Min Heap method

Question: Write an efficient program for printing k largest elements in an array. Elements in array can be in any order.

For example, if given array is [1, 23, 12, 9, 30, 2, 50] and you are asked for the largest 3 elements i.e., k = 3 then your program should print 50, 30 and 23.

Method 1 (Use Bubble k times)

Thanks to Shailendra for suggesting this approach.

- 1) Modify [Bubble Sort](#) to run the outer loop at most k times.
- 2) Print the last k elements of the array obtained in step 1.

Time Complexity: $O(nk)$

Like Bubble sort, other sorting algorithms like [Selection Sort](#) can also be modified to get the k largest elements.

Method 2 (Use temporary array)

K largest elements from arr[0..n-1]

- 1) Store the first k elements in a temporary array temp[0..k-1].
- 2) Find the smallest element in temp[], let the smallest element be *min*.
- 3) For each element *x* in arr[k] to arr[n-1]
If *x* is greater than the min then remove *min* from temp[] and insert *x*.
- 4) Print final k elements of temp[]

Time Complexity: $O((n-k)*k)$. If we want the output sorted then $O((n-k)*k + k\log k)$

Thanks to nesamani1822 for suggesting this method.

Method 3(Use Sorting)

- 1) Sort the elements in descending order in $O(n\log n)$
- 2) Print the first k numbers of the sorted array $O(k)$.

Time complexity: $O(n\log n)$

Method 4 (Use Max Heap)

- 1) Build a Max Heap tree in $O(n)$
- 2) Use [Extract Max](#) k times to get k maximum elements from the Max Heap $O(k\log n)$

Time complexity: $O(n + k\log n)$

Method 5(Use Order Statistics)

- 1) Use order statistic algorithm to find the kth largest element. Please see the topic [selection in worst-case linear time](#) $O(n)$
- 2) Use [QuickSort](#) Partition algorithm to partition around the kth largest number $O(n)$.
- 3) Sort the k-1 elements (elements greater than the kth largest element) $O(k\log k)$. This step is needed only if sorted output is required.

Time complexity: $O(n)$ if we don't need the sorted output, otherwise $O(n+k\log k)$

Thanks to [Shilpi](#) for suggesting the first two approaches.

Method 6 (Use Min Heap)

This method is mainly an optimization of method 1. Instead of using temp[] array, use Min Heap.

Thanks to [geek4u](#) for suggesting this method.

- 1) Build a Min Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. $O(k)$
- 2) For each element, after the kth element (arr[k] to arr[n-1]), compare it with root of MH.
.....a) If the element is greater than the root then make it root and call [heapify](#) for MH
.....b) Else ignore it.
// The step 2 is $O((n-k)*\log k)$
- 3) Finally, MH has k largest elements and root of the MH is the kth largest element.

Time Complexity: $O(k + (n-k)\log k)$ without sorted output. If sorted output is needed then $O(k + (n-k)\log k + k\log k)$

All of the above methods can also be used to find the kth largest (or smallest) element.

Please write comments if you find any of the above explanations/algorithms incorrect, or find better ways to solve the same problem.

References:

http://en.wikipedia.org/wiki/Selection_algorithm

Asked by [geek4u](#)

Pythagorean Triplet in an array

Given an array of integers, write a function that returns true if there is a triplet (a, b, c) that satisfies $a^2 + b^2 = c^2$.

Example:

```
Input: arr[] = {3, 1, 4, 6, 5}
Output: True
There is a Pythagorean triplet (3, 4, 5).

Input: arr[] = {10, 4, 6, 12, 5}
Output: False
There is no Pythagorean triplet.
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Method 1 (Naive)

A simple solution is to run three loops, three loops pick three array elements and check if current three elements form a Pythagorean Triplet.

Below is C++ implementation of simple solution.

C++

```
// A C++ program that returns true if there is a Pythagorean
// Triplet in a given aray.
#include <iostream>
using namespace std;

// Returns true if there is Pythagorean triplet in ar[0..n-1]
bool isTriplet(int ar[], int n)
{
    for (int i=0; i<n; i++)
    {
        for (int j=i+1; j<n; j++)
        {
            for (int k=j+1; k<n; k++)
            {
                // Calculate square of array elements
                int x = ar[i]*ar[i], y = ar[j]*ar[j], z = ar[k]*ar[k];

                if (x == y + z || y == x + z || z == x + y)
                    return true;
            }
        }
    }

    // If we reach here, no triplet found
    return false;
}

/* Driver program to test above function */
int main()
{
    int ar[] = {3, 1, 4, 6, 5};
    int ar_size = sizeof(ar)/sizeof(ar[0]);
    isTriplet(ar, ar_size)? cout << "Yes": cout << "No";
    return 0;
}
```

Java

```
// A Java program that returns true if there is a Pythagorean
// Triplet in a given aray.
```

```

import java.io.*;

class PythagoreanTriplet {

    // Returns true if there is Pythagorean triplet in ar[0..n-1]
    static boolean isTriplet(int ar[], int n)
    {
        for (int i=0; i<n; i++)
        {
            for (int j=i+1; j<n; j++)
            {
                for (int k=j+1; k<n; k++)
                {
                    // Calculate square of array elements
                    int x = ar[i]*ar[i], y = ar[j]*ar[j], z = ar[k]*ar[k];

                    if (x == y + z || y == x + z || z == x + y)
                        return true;
                }
            }
        }

        // If we reach here, no triplet found
        return false;
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int ar[] = {3, 1, 4, 6, 5};
        int ar_size = ar.length;
        if(isTriplet(ar,ar_size)==true)
            System.out.println("Yes");
        else
            System.out.println("No");
    }
}

/* This code is contributed by Devesh Agrawal */

```

Output:

Yes

Time Complexity of the above solution is $O(n^3)$.

Method 2 (Use Sorting)

We can solve this in $O(n^2)$ time by sorting the array first.

- 1) Do square of every element in input array. This step takes $O(n)$ time.
- 2) Sort the squared array in increasing order. This step takes $O(n \log n)$ time.
- 3) To find a triplet (a, b, c) such that $a = b + c$, do following.

1. Fix ' a ' as last element of sorted array.
2. Now search for pair (b, c) in subarray between first element and ' a '. A pair (b, c) with given sum can be found in $O(n)$ time using meet in middle algorithm discussed in method 1 of [this](#) post.
3. If no pair found for current ' a ', then move ' a ' one position back and repeat step 3.2.

Below is C++ implementation of above algorithm.

C++

```

// A C++ program that returns true if there is a Pythagorean
// Triplet in a given array.
#include <iostream>
#include <algorithm>
using namespace std;

// Returns true if there is a triplet with following property

```

```

// A[i]*A[i] = A[j]*A[j] + A[k]*[k]
// Note that this function modifies given array
bool isTriplet(int arr[], int n)
{
    // Square array elements
    for (int i=0; i<n; i++)
        arr[i] = arr[i]*arr[i];

    // Sort array elements
    sort(arr, arr + n);

    // Now fix one element one by one and find the other two
    // elements
    for (int i = n-1; i >= 2; i--)
    {
        // To find the other two elements, start two index
        // variables from two corners of the array and move
        // them toward each other
        int l = 0; // index of the first element in arr[0..i-1]
        int r = i-1; // index of the last element in arr[0..i-1]
        while (l < r)
        {
            // A triplet found
            if (arr[l] + arr[r] == arr[i])
                return true;

            // Else either move 'l' or 'r'
            (arr[l] + arr[r] < arr[i])? l++: r--;
        }
    }

    // If we reach here, then no triplet found
    return false;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {3, 1, 4, 6, 5};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    isTriplet(arr, arr_size)? cout << "Yes": cout << "No";
    return 0;
}

```

Java

```

// A Java program that returns true if there is a Pythagorean
// Triplet in a given array.
import java.io.*;
import java.util.*;

class PythagoreanTriplet
{
    // Returns true if there is a triplet with following property
    // A[i]*A[i] = A[j]*A[j] + A[k]*[k]
    // Note that this function modifies given array
    static boolean isTriplet(int arr[], int n)
    {
        // Square array elements
        for (int i=0; i<n; i++)
            arr[i] = arr[i]*arr[i];

        // Sort array elements
        Arrays.sort(arr);

        // Now fix one element one by one and find the other two
        // elements
        for (int i = n-1; i >= 2; i--)
        {
            // To find the other two elements, start two index
            // variables from two corners of the array and move
            // them toward each other
            int l = 0; // index of the first element in arr[0..i-1]

```

```

int r = i-1; // index of the last element in arr[0..i-1]
while (l < r)
{
    // A triplet found
    if (arr[l] + arr[r] == arr[i])
        return true;

    // Else either move 'l' or 'r'
    if (arr[l] + arr[r] < arr[i])
        l++;
    else
        r--;
}
}

// If we reach here, then no triplet found
return false;
}

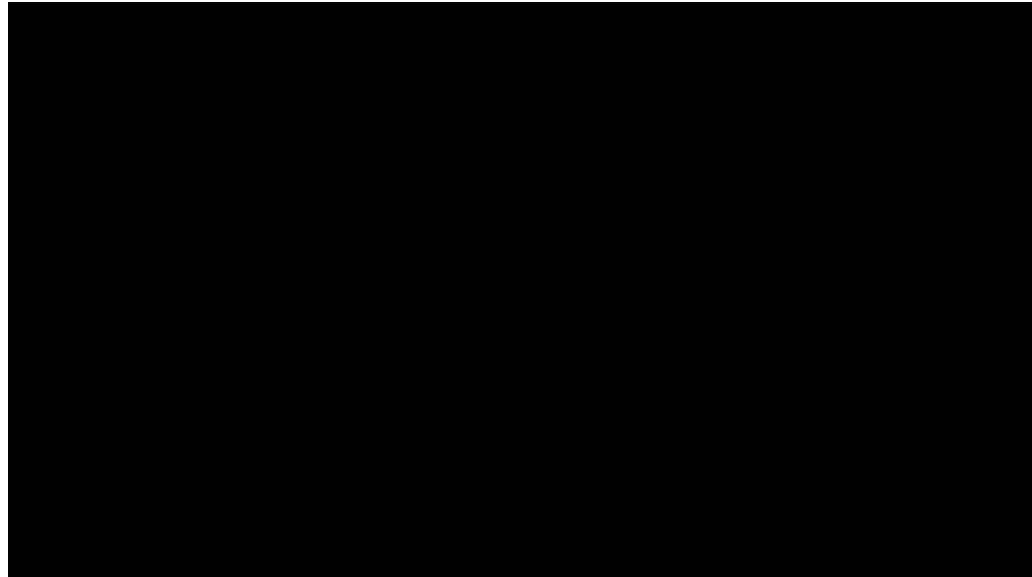
// Driver program to test above function
public static void main(String[] args)
{
    int arr[] = {3, 1, 4, 6, 5};
    int arr_size = arr.length;
    if (isTriplet(arr,arr_size)==true)
        System.out.println("Yes");
    else
        System.out.println("No");
}
}
/*This code is contributed by Devesh Agrawal*/

```

Output:

Yes

Time complexity of this method is $O(n^2)$.



This article is contributed by **Harshit Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Arrays

Question 1

Following function is supposed to calculate the maximum depth or height of a Binary tree -- the number of nodes along the longest path from the root node down to the farthest leaf node.

```

int maxDepth(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node->left);
        int rDepth = maxDepth(node->right);

        /* use the larger one */
        if (lDepth > rDepth)
            return X;
        else return Y;
    }
}

```

What should be the values of X and Y so that the function works correctly?

- | |
|----------------------------------|
| A X = lDepth, Y = rDepth |
| B X = lDepth + 1, Y = rDepth + 1 |
| C X = lDepth - 1, Y = rDepth - 1 |
| D None of the above |

Tree Traversals

Discuss it

Question 1 Explanation:

If a tree is not empty, height of tree is MAX(Height of Left Subtree, Height of Right Subtree) + 1 See [program to Find the Maximum Depth or Height of a Tree](#) for more details.

Question 2

What is common in three different types of traversals (Inorder, Preorder and Postorder)?

- | |
|---|
| A Root is visited before right subtree |
| B Left subtree is always visited before right subtree |
| C Root is visited after left subtree |
| D All of the above |
| E None of the above |

Tree Traversals

Discuss it

Question 2 Explanation:

The order of inorder traversal is LEFT ROOT RIGHT The order of preorder traversal is ROOT LEFT RIGHT The order of postorder traversal is LEFT RIGHT ROOT In all three traversals, LEFT is traversed before RIGHT

Question 3

The inorder and preorder traversal of a binary tree are d b e a f c g and a b d e c f g, respectively. The postorder traversal of the binary tree is:

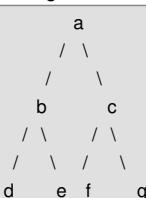
- | |
|-----------------|
| A d e b f g c a |
| B e d b g f c a |
| C e d b f g c a |
| D d e f g b c a |

Tree Traversals

Discuss it

Question 3 Explanation:

Below is the given tree.



Question 4

What does the following function do for a given binary tree?

```

int fun(struct node *root)
{
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;
    return 1 + fun(root->left) + fun(root->right);
}

```

- | |
|---|
| A Counts leaf nodes |
| B Counts internal nodes |
| C Returns height where height is defined as number of edges on the path from root to deepest node |

D Return diameter where diameter is number of edges on the longest path between any two nodes.

Tree Traversals

Discuss it

Question 4 Explanation:

The function counts internal nodes. 1) If root is NULL or a leaf node, it returns 0. 2) Otherwise returns, 1 plus count of internal nodes in left subtree, plus count of internal nodes in right subtree. See the following complete program.

Question 5

Which of the following pairs of traversals is not sufficient to build a binary tree from the given traversals?

- | |
|--------------------------|
| A Preorder and Inorder |
| B Preorder and Postorder |
| C Inorder and Postorder |
| D None of the Above |

Tree Traversals

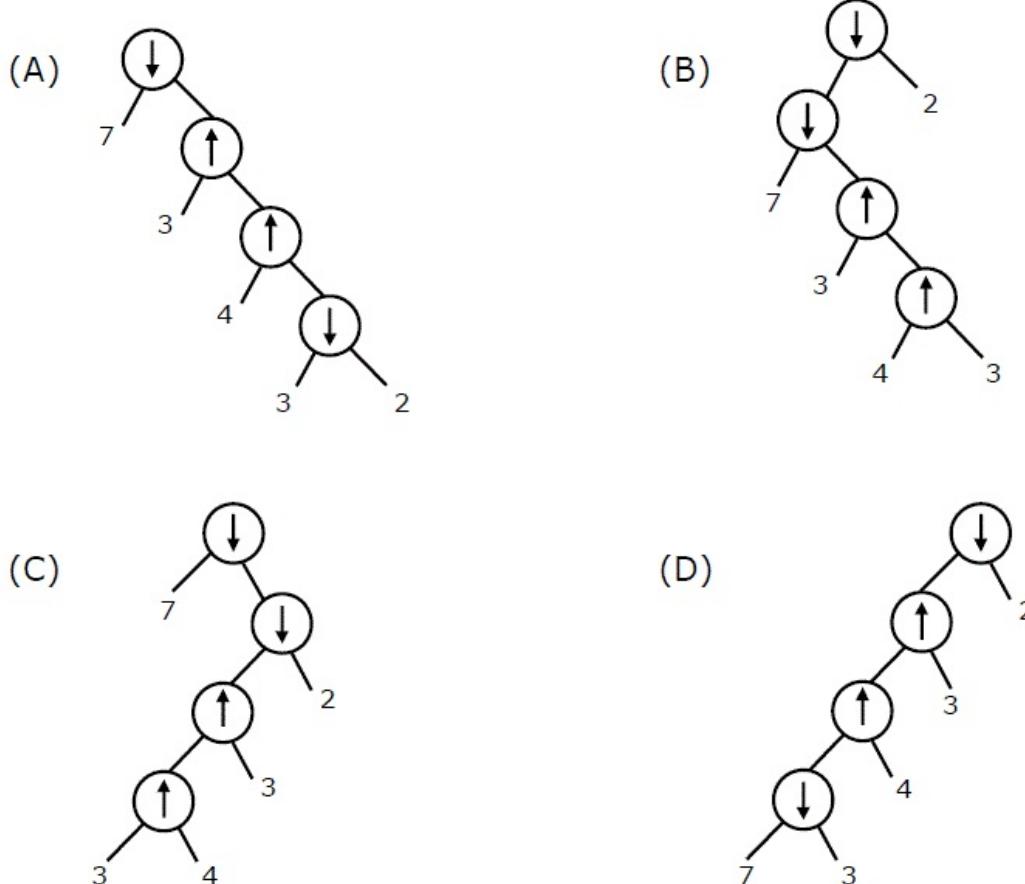
Discuss it

Question 5 Explanation:

See <http://www.geeksforgeeks.org/if-you-are-given-two-traversal-sequences-can-you-construct-the-binary-tree/> for details.

Question 6

Consider two binary operators ' \uparrow ' and ' \downarrow ' with the precedence of operator \downarrow being lower than that of the \uparrow operator. Operator \uparrow is right associative while operator \downarrow is left associative. Which one of the following represents the parse tree for expression $(7 \downarrow 3 \uparrow 4 \uparrow 3 \downarrow 2)$? CS
(GATE 2011)



- | |
|-----|
| A A |
| B B |
| C C |
| D D |

Tree Traversals

Discuss it

Question 6 Explanation:

Let us consider the given expression $(7 \downarrow 3 \uparrow 4 \uparrow 3 \downarrow 2)$. Since the precedence of \uparrow is higher, the sub-expression $(3 \uparrow 4 \uparrow 3)$ will be evaluated first. In this sub-expression, $3 \uparrow 4$ would be evaluated first because \uparrow is right to left associative. So the expression is evaluated as $((7 \downarrow (3 \uparrow (4 \uparrow 3))) \downarrow 2)$. Also, note that among the two \downarrow operators, first one is evaluated before the second one because the associativity of \downarrow is left to right.

Question 7

Which traversal of tree resembles the breadth first search of the graph?

- | |
|---------------|
| A Preorder |
| B Inorder |
| C Postorder |
| D Level order |

Tree Traversals

Discuss it

Question 7 Explanation:

Breadth first search visits all the neighbors first and then deepens into each neighbor one by one. The level order traversal of the tree also visits nodes on the current level and then goes to the next level.

Question 8

Which of the following tree traversal uses a queue data structure?

- A Preorder
- B Inorder
- C Postorder
- D Level order

Tree Traversals

Discuss it

Question 8 Explanation:

Level order traversal uses a queue data structure to visit the nodes level by level.

Question 9

Which of the following cannot generate the full binary tree?

- A Inorder and Preorder
- B Inorder and Postorder
- C Preorder and Postorder
- D None of the above

Tree Traversals

Discuss it

Question 9 Explanation:

To generate a binary tree, two traversals are necessary and one of them must be inorder. But, a full binary tree can be generated from preorder and postorder traversals. Read the algorithm [here](#). Read [Can tree be constructed from given traversals?](#)

Question 10

Consider the following C program segment

```
struct CellNode
{
    struct CellNode *leftchild;
    int element;
    struct CellNode *rightChild;
}

int Dosomething(struct CellNode *ptr)
{
    int value = 0;
    if (ptr != NULL)
    {
        if (ptr->leftChild != NULL)
            value = 1 + DoSomething(ptr->leftChild);
        if (ptr->rightChild != NULL)
            value = max(value, 1 + DoSomething(ptr->rightChild));
    }
    return (value);
}
```

The value returned by the function DoSomething when a pointer to the root of a non-empty tree is passed as argument is (GATE CS 2004)

- A The number of leaf nodes in the tree
- B The number of nodes in the tree
- C The number of internal nodes in the tree
- D The height of the tree

Tree Traversals

Discuss it

Question 10 Explanation:

Explanation: DoSomething() returns $\max(\text{height of left child} + 1, \text{height of right child} + 1)$. So given that pointer to root of tree is passed to DoSomething(), it will return height of the tree. Note that this implementation follows the convention where height of a single node is 0.

Question 11

Let LASTPOST, LASTIN and LASTPRE denote the last vertex visited in a postorder, inorder and preorder traversal. Respectively, of a complete binary tree. Which of the following is always true? (GATE CS 2000)

- A LASTIN = LASTPOST
- B LASTIN = LASTPRE
- C LASTPRE = LASTPOST
- D None of the above

Tree Traversals

Discuss it

Question 11 Explanation:

It is given that the given tree is [complete binary tree](#). For a complete binary tree, the last visited node will always be same for inorder and preorder traversal. None of the above is true even for a complete binary tree. The option (a) is incorrect because the last node visited in Inorder

traversal is right child and last node visited in Postorder traversal is root. The option (c) is incorrect because the last node visited in Preorder traversal is right child and last node visited in Postorder traversal is root. For option (b), see the following counter example. Thanks to [Hunain Muhammed](#) for providing the correct explanation.

```

1
/ \
2   3
/\  /
4 5 6

```

Inorder traversal is 4 2 5 1 6 3
 Preorder traversal is 1 2 4 5 3 6

Question 12

The array representation of a complete binary tree contains the data in sorted order. Which traversal of the tree will produce the data in sorted form?

- | | |
|---|-------------|
| A | Preorder |
| B | Inorder |
| C | Postorder |
| D | Level order |

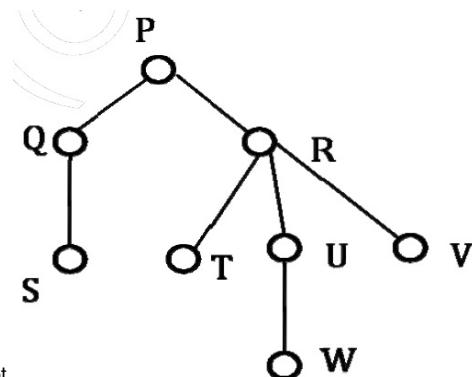
Tree Traversals

Discuss it

Question 12 Explanation:

The [level order traversal](#) of a binary tree prints the data in the same order as it is stored in the array representation of a complete binary tree.

Question 13



The order in which the

Consider the following rooted tree with the vertex P labeled as root nodes are visited during in-order traversal is

- | | |
|---|----------|
| A | SQPTRWUV |
| B | SQPTURWV |
| C | SQPTWUVR |
| D | SQPTRUWV |

Tree Traversals GATE-CS-2014-(Set-3)

Discuss it

Question 13 Explanation:

Algorithm Inorder(tree) - Use of Recursion

Steps:

1. Traverse the left subtree,
i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree,
i.e., call Inorder(right-subtree)

Understanding this algorithm requires the basic understanding of Recursion

Therefore, We begin in the above tree with root as the starting point, which is P.

Step 1(for node P) :

Traverse the left subtree of node or root P.

So we have node Q on left of P.

-> Step 1(for node Q)

Traverse the left subtree of node Q.

So we have node S on left of Q.

* Step 1 (for node S)

Now again traverse the left subtree of node S which is NULL here.

* Step 2(for node S)

Visit the node S, i.e print node S as the 1st element of inorder traversal.

* Step 3(for node S)

Traverse the right subtree of node S.

Which is NULL here.

Now move up in the tree to Q which is parent of S.(Recursion, function of Q called for function of S). Hence we go back to Q.

-> Step 2(for node Q):

Visit the node Q, i.e print node Q as the 2nd element of inorder traversal.

-> Step 3 (for node Q)

Traverse the right subtree of node Q.

Which is NULL here.

Now move up in the tree to P which is parent of Q.(Recursion, function of P called for function of Q). Hence we go back to P.

Step 2(for node P)

Visit the node P, i.e print node S as the 3rd element of inorder traversal.

Step 3 (for node P)

Traverse the right subtree of node P.

Node R is at the right of P.

Till now we have printed SQP as the inorder of the tree. Similarly other elements can be obtained by traversing the right subtree of P.

The final correct order of Inorder traversal would be SQPTRWUV.

Question 14

Consider the pseudocode given below. The function DoSomething() takes as argument a pointer to the root of an arbitrary tree represented by the leftMostChild-rightSibling representation. Each node of the tree is of type treeNode.

```
typedef struct treeNode* treeptr;
struct treeNode
{
    treeptr leftMostChild, rightSibling;
};
int DoSomething (treeptr tree)
{
    int value=0;
    if (tree != NULL)
    {
        if (tree->leftMostChild == NULL)
            value = 1;
        else
            value = DoSomething(tree->leftMostChild);
        value = value + DoSomething(tree->rightSibling);
    }
    return(value);
}
```

When the pointer to the root of a tree is passed as the argument to DoSomething, the value returned by the function corresponds to the

A	number of internal nodes in the tree.
B	height of the tree.
C	number of nodes without a right sibling in the tree.
D	number of leaf nodes in the tree.

Tree Traversals GATE-CS-2014-(Set-3)

Discuss it

Question 14 Explanation:

The function counts leaf nodes for a tree represented using leftMostChild-rightSibling representation. Below is function with comments added to demonstrate how function works. 1

Question 15

Level order traversal of a rooted tree can be done by starting from the root and performing

A	preorder traversal
B	inorder traversal
C	depth first search
D	breadth first search

Tree Traversals GATE-CS-2004

Discuss it

Question 15 Explanation:

See [this post](#) for details

Question 16

Consider the label sequences obtained by the following pairs of traversals on a labeled binary tree. Which of these pairs identify a tree uniquely?

- (i) preorder and postorder
- (ii) inorder and postorder
- (iii) preorder and inorder
- (iv) level order and postorder

A	(i) only
B	(ii), (iii)
C	(iii) only
D	(iv) only

Tree Traversals GATE-CS-2004

Discuss it

Question 16 Explanation:

Here, we consider each and every option to check whether it is true or false.

1) Preorder and postorder



For the above trees, Preorder is AB Postorder is BA It shows

that preorder and postorder can't identify a tree uniquely.

2) Inorder and postorder define a tree uniquely

3) Preorder and Inorder also define a tree uniquely

4) Levelorder and postorder can't define a tree uniquely.

For the above example, Level order is AB Postorder is BA See <http://www.geeksforgeeks.org/if-you-are-given-two-traversal-sequences-can-you-construct-the-binary-tree/> for details

This solution is contributed by **Anil Saikrishna Devarasetty**

Question 17

Let LASTPOST, LASTIN and LASTPRE denote the last vertex visited in a postorder, inorder and preorder traversal, respectively, of a complete binary tree. Which of the following is always true?

A	LASTIN = LASTPOST
B	LASTIN = LASTPRE
C	LASTPRE = LASTPOST
D	None of the above

Tree Traversals GATE-CS-2000

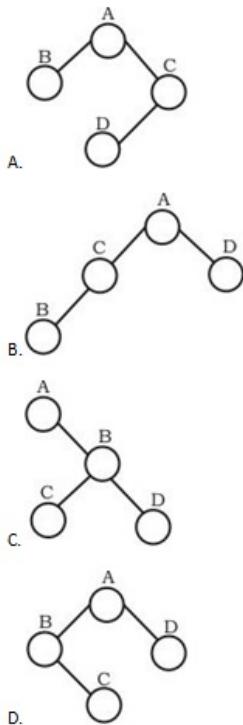
Discuss it

Question 17 Explanation:

See question 1 of <http://www.geeksforgeeks.org/data-structures-and-algorithms-set-1/>

Question 18

Which one of the following binary trees has its inorder and preorder traversals as BCAD and ABCD, respectively?



AA
BB
CC
DD

Tree Traversals GATE-IT-2004

Discuss it

Question 18 Explanation:

Inorder Traversal: **Left -Root -Right** PreOrder Traversal: **Root-Left-Right**

InOrder PreOrder

- A. BADC ABCD
- B. BCAD ACBD
- C. ACBD ABCD

D. BCAD ABCD Therefore, D is Correct

Question 19

The numbers 1, 2, ..., n are inserted in a binary search tree in some order. In the resulting tree, the right subtree of the root contains p nodes. The first number to be inserted in the tree must be

A p
B p + 1
C n - p
D n - p + 1

Tree Traversals Gate IT 2005

Discuss it

Question 19 Explanation:

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree. There must be no duplicate nodes.

So let us say n=10, p=4. According to BST property the root must be 10-4=6 (considering all unique elements in BST)
And according to **BST insertion**, root is the first element to be inserted in a BST.

Therefore, the answer is (n-p).

Question 20

A binary search tree contains the numbers 1, 2, 3, 4, 5, 6, 7, 8. When the tree is traversed in pre-order and the values in each node printed out, the sequence of values obtained is 5, 3, 1, 2, 4, 6, 8, 7. If the tree is traversed in post-order, the sequence obtained would be

A 8, 7, 6, 5, 4, 3, 2, 1
B 1, 2, 3, 4, 8, 7, 6, 5

C	2, 1, 4, 3, 6, 7, 8, 5
D	2, 1, 4, 3, 7, 8, 6, 5

Tree Traversals Gate IT 2005

Discuss it

Question 20 Explanation:

Please see this link for more details <http://www.geeksforgeeks.org/construct-tree-from-given-inorder-and-preorder-traversal/>

Question 21

If all the edge weights of an undirected graph are positive, then any subset of edges that connects all the vertices and has minimum total weight is a

A	Hamiltonian cycle
B	grid
C	hypercube
D	tree

Tree Traversals Graph Theory GATE IT 2006

Discuss it

Question 21 Explanation:

As here we want subset of edges that connects all the vertices and has minimum total weight i.e. Minimum Spanning Tree Option A - includes cycle, so may or may not connect all edges. Option B - has no relevance to this question. Option C - includes cycle, so may or may not connect all edges. Related : <http://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/> <http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/> This solution is contributed by **Mohit Gupta**.

Question 22

When searching for the key value 60 in a binary search tree, nodes containing the key values 10, 20, 40, 50, 70 80, 90 are traversed, not necessarily in the order given. How many different orders are possible in which these key values can occur on the search path from the root to the node containing the value 60?

A	35
B	64
C	128
D	5040

Binary Search Trees Tree Traversals Gate IT 2007

Discuss it

Question 22 Explanation:

There are two set of values, smaller than 60 and greater than 60. Smaller values 10, 20, 40 and 50 are visited, means they are visited in order. Similarly, 90, 80 and 70 are visited in order.

$$= 7!/(4!3!)$$

$$= 35$$

Question 23

The following three are known to be the preorder, inorder and postorder sequences of a binary tree. But it is not known which is which.

MBCAFHPYK

KAMCBYPFH

MABCKYFPH

Pick the true statement from the following.

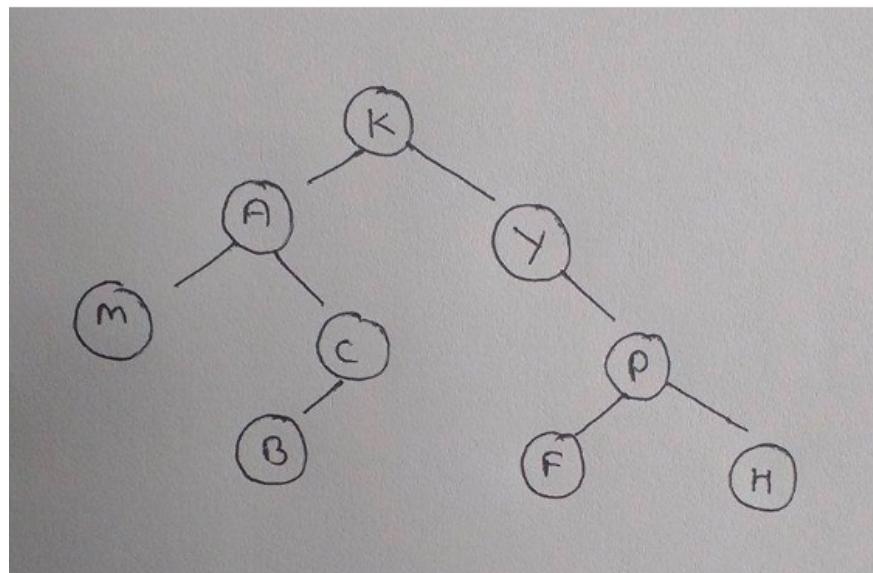
A	I and II are preorder and inorder sequences, respectively
B	I and III are preorder and postorder sequences, respectively
C	II is the inorder sequence, but nothing more can be said about the other two sequences
D	II and III are the preorder and inorder sequences, respectively

Binary Trees Tree Traversals Gate IT 2008

Discuss it

Question 23 Explanation:

The approach to solve this question is to first find 2 sequences whose first and last element is same. The reason being first element in the Pre-order of any binary tree is the root and last element in the Post-order of any binary tree is the root. Looking at the sequences given, Pre-order = KAMCBYPFH Post-order = MBCAFHPYK Left-over sequence MABCKYFPH will be in order. Since we have all the traversals identified,



I.

let's try to draw the binary tree if possible.

Post order

II. Pre order

III. Inorder This solution is contributed by **Pranjal Ahuja**.

Question 24

Consider the following sequence of nodes for the undirected graph given below.

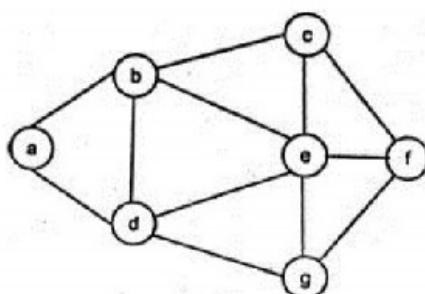
a b e f d g c

a b e f c g d

a d g e b c f

a d b c g e f

A Depth First Search (DFS) is started at node a. The nodes are listed in the order they are first visited. Which all of the above is (are) possible output(s)?



- | |
|------------------|
| A1 and 3 only |
| B2 and 3 only |
| C2, 3 and 4 only |
| D1, 2, and 3 |

Tree Traversals Gate IT 2008

Discuss it

Question 24 Explanation:

1: abef->c or g should be covered

4: adbc->e or f should be covered

2: abefcgd correct

3: adgebcf correct

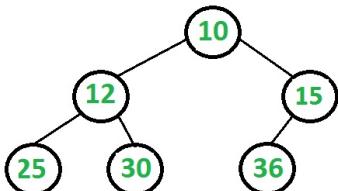
There are 24 questions to complete.

GATE CS Corner

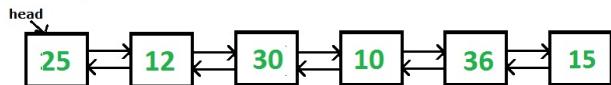
See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.

Convert a given Binary Tree to Doubly Linked List | Set 4

Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL) In-Place. The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.



The above tree should be in-place converted to following Doubly Linked List(DLL).



Below three different solutions have been discussed for this problem.

[Convert a given Binary Tree to Doubly Linked List | Set 1](#)

[Convert a given Binary Tree to Doubly Linked List | Set 2](#)

[Convert a given Binary Tree to Doubly Linked List | Set 3](#)

In the following implementation, we traverse the tree in inorder fashion. We add nodes at the beginning of current linked list and update head of the list using pointer to head pointer. Since we insert at the beginning, we need to process leaves in reverse order. For reverse order, we first traverse the right subtree before the left subtree. i.e. do a reverse inorder traversal.

C++

```

// C++ program to convert a given Binary
// Tree to Doubly Linked List
#include <stdio.h>
#include <stdlib.h>

// Structure for tree and linked list
struct Node
{
    int data;
    Node *left, *right;
};

// A simple recursive function to convert a given
// Binary tree to Doubly Linked List
// root --> Root of Binary Tree
// head_ref --> Pointer to head node of created
//          doubly linked list
void BToDLL(Node* root, Node** head_ref)
{
    // Base cases
    if (root == NULL)
        return;

    // Recursively convert right subtree
    BToDLL(root->right, head_ref);

    // Insert root into DLL
    root->right = *head_ref;

    // Change left pointer of previous head
    if (*head_ref != NULL)
        (*head_ref)->left = root;

    // Change head of Doubly linked list
    *head_ref = root;

    // Recursively convert left subtree
    BToDLL(root->left, head_ref);
}

// Utility function for allocating node for Binary
// Tree.
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
}

```

```

    return node;
}

// Utility function for printing double linked list.
void printList(Node* head)
{
    printf("Extracted Double Linked list is:\n");
    while (head)
    {
        printf("%d ", head->data);
        head = head->right;
    }
}

// Driver program to test above function
int main()
{
    /* Constructing below tree
       5
      / \
     3   6
    / \   \
   1   4   8
  / \   / \
 0   2   7   9 */
    Node* root = newNode(5);
    root->left = newNode(3);
    root->right = newNode(6);
    root->left->left = newNode(1);
    root->left->right = newNode(4);
    root->right->right = newNode(8);
    root->left->left->left = newNode(0);
    root->left->left->right = newNode(2);
    root->right->right->left = newNode(7);
    root->right->right->right = newNode(9);

    Node* head = NULL;
    BToDLL(root, &head);

    printList(head);

    return 0;
}

```

Java

```

// Java program to convert a given Binary Tree to
// Doubly Linked List

/* Structure for tree and Linked List */
class Node
{
    int data;
    Node left, right;

    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree
{
    // 'root' - root of binary tree
    Node root;

    // 'head' - reference to head node of created
    //double linked list
    Node head;

    // A simple recursive function to convert a given

```

```

// Binary tree to Doubly Linked List
void BToDLL(Node root)
{
    // Base cases
    if (root == null)
        return;

    // Recursively convert right subtree
    BToDLL(root.right);

    // insert root into DLL
    root.right = head;

    // Change left pointer of previous head
    if (head != null)
        (head).left = root;

    // Change head of Doubly linked list
    head = root;

    // Recursively convert left subtree
    BToDLL(root.left);
}

// Utility function for printing double linked list.
void printList(Node head)
{
    System.out.println("Extracted Double Linked List is : ");
    while (head != null)
    {
        System.out.print(head.data + " ");
        head = head.right;
    }
}

// Driver program to test the above functions
public static void main(String[] args)
{
    /* Constructing below tree
       5
      / \
     3   6
    / \   \
   1   4   8
  / \   / \
 0   2   7   9 */

    BinaryTree tree = new BinaryTree();
    tree.root = new Node(5);
    tree.root.left = new Node(3);
    tree.root.right = new Node(6);
    tree.root.left.right = new Node(4);
    tree.root.left.left = new Node(1);
    tree.root.right.right = new Node(8);
    tree.root.left.left.right = new Node(2);
    tree.root.left.left.left = new Node(0);
    tree.root.right.right.left = new Node(7);
    tree.root.right.right.right = new Node(9);

    tree.BToDLL(tree.root);
    tree.printList(tree.head);
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)

```

Output :

Extracted Double Linked list is:
0 1 2 3 4 5 6 7 8 9

Time Complexity: O(n), as the solution does a single traversal of given Binary Tree.

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

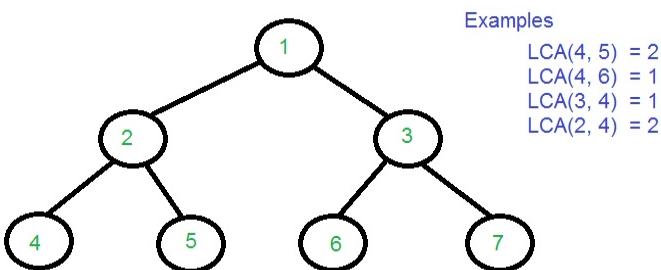
Lowest Common Ancestor in a Binary Tree | Set 1

Given a binary tree (not a binary search tree) and two values say n1 and n2, write a program to find the least common ancestor.

Following is definition of LCA from Wikipedia:

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source [Wiki](#))



We have discussed an efficient solution to find [LCA in Binary Search Tree](#). In Binary Search Tree, using BST properties, we can find LCA in $O(h)$ time where h is height of tree. Such an implementation is not possible in Binary Tree as keys Binary Tree nodes don't follow any order. Following are different approaches to find LCA in Binary Tree.

Method 1 (By Storing root to n1 and root to n2 paths):

Following is simple $O(n)$ algorithm to find LCA of n1 and n2.

- 1) Find path from root to n1 and store it in a vector or array.
- 2) Find path from root to n2 and store it in another vector or array.
- 3) Traverse both paths till the values in arrays are same. Return the common element just before the mismatch.

Following is C++ implementation of above algorithm.

C++

```
// A O(n) solution to find LCA of two given values n1 and n2
#include <iostream>
#include <vector>
using namespace std;

// A Binary Tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// Utility function creates a new binary tree node with given key
Node * newNode(int k)
{
    Node *temp = new Node;
    temp->key = k;
    temp->left = temp->right = NULL;
    return temp;
}

// Finds the path from root node to given root of the tree, Stores the
// path in a vector path[], returns true if path exists otherwise false
bool findPath(Node *root, vector<int> &path, int k)
{
    // base case
    if (root == NULL) return false;
    path.push_back(root->key);
    if (root->key == k)
        return true;
    if (findPath(root->left, path, k))
        return true;
    if (findPath(root->right, path, k))
        return true;
    path.pop_back();
    return false;
}

// This function mainly finds LCA of n1 and n2, stores the path
// from root to n1 and n2 in path[], and returns LCA
Node* findLCA(Node *root, int n1, int n2)
{
    vector<int> path1, path2;
    if (!findPath(root, path1, n1) || !findPath(root, path2, n2))
        return NULL;
    int i = 0;
    while (i < path1.size() && i < path2.size() && path1[i] == path2[i])
        i++;
    return (Node *)(&path1[i - 1]);
}
```

```

// Store this node in path vector. The node will be removed if
// not in path from root to k
path.push_back(root->key);

// See if the k is same as root's key
if (root->key == k)
    return true;

// Check if k is found in left or right sub-tree
if ( (root->left && findPath(root->left, path, k)) ||
     (root->right && findPath(root->right, path, k)) )
    return true;

// If not present in subtree rooted with root, remove root from
// path[] and return false
path.pop_back();
return false;
}

// Returns LCA if node n1, n2 are present in the given binary tree,
// otherwise return -1
int findLCA(Node *root, int n1, int n2)
{
    // to store paths to n1 and n2 from the root
    vector<int> path1, path2;

    // Find paths from root to n1 and root to n1. If either n1 or n2
    // is not present, return -1
    if ( !findPath(root, path1, n1) || !findPath(root, path2, n2) )
        return -1;

    /* Compare the paths to get the first different value */
    int i;
    for (i = 0; i < path1.size() && i < path2.size(); i++)
        if (path1[i] != path2[i])
            break;
    return path1[i];
}

// Driver program to test above functions
int main()
{
    // Let us create the Binary Tree shown in above diagram.
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5);
    cout << "\nLCA(4, 6) = " << findLCA(root, 4, 6);
    cout << "\nLCA(3, 4) = " << findLCA(root, 3, 4);
    cout << "\nLCA(2, 4) = " << findLCA(root, 2, 4);
    return 0;
}

```

Python

```

# O(n) solution to find LCS of two given values n1 and n2

# A binary tree node
class Node:
    # Constructor to create a new binary node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    # Finds the path from root node to given root of the tree.
    # Stores the path in a list path[], returns true if path
    # exists otherwise false

```

```

def findPath( root, path, k):

    # Base Case
    if root is None:
        return False

    # Store this node in path vector. The node will be
    # removed if not in path from root to k
    path.append(root.key)

    # See if the k is same as root's key
    if root.key == k :
        return True

    # Check if k is found in left or right sub-tree
    if ((root.left != None and findPath(root.left, path, k)) or
        (root.right!= None and findPath(root.right, path, k))):
        return True

    # If not present in subtree rooted with root, remove
    # root from path and return False

    path.pop()
    return False

# Returns LCA if node n1 , n2 are present in the given
# binary tree otherwise return -1
def findLCA(root, n1, n2):

    # To store paths to n1 and n2 from the root
    path1 = []
    path2 = []

    # Find paths from root to n1 and root to n2.
    # If either n1 or n2 is not present , return -1
    if (not findPath(root, path1, n1) or not findPath(root, path2, n2)):
        return -1

    # Compare the paths to get the first different value
    i = 0
    while(i < len(path1) and i < len(path2)):
        if path1[i] != path2[i]:
            break
        i += 1
    return path1[i-1]

# Driver program to test above function
# Let's create the Binary Tree shown in above diagram
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

print "LCA(4, 5) = %d" %(findLCA(root, 4, 5))
print "LCA(4, 6) = %d" %(findLCA(root, 4, 6))
print "LCA(3, 4) = %d" %(findLCA(root, 3, 4))
print "LCA(2, 4) = %d" %(findLCA(root, 2, 4))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

LCA(4, 5) = 2
LCA(4, 6) = 1
LCA(3, 4) = 1
LCA(2, 4) = 2

```

Time Complexity: Time complexity of the above solution is O(n). The tree is traversed twice, and then path arrays are compared. Thanks to *Ravi Chandra Enaganti* for suggesting the initial solution based on this method.

Method 2 (Using Single Traversal)

The method 1 finds LCA in O(n) time, but requires three tree traversals plus extra spaces for path arrays. If we assume that the keys n1 and n2 are present in Binary Tree, we can find LCA using single traversal of Binary Tree and without extra storage for path arrays.

The idea is to traverse the tree starting from root. If any of the given keys (n1 and n2) matches with root, then root is LCA (assuming that both keys are present). If root doesn't match with any of the keys, we recur for left and right subtree. The node which has one key present in its left subtree and the other key present in right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA also, otherwise LCA lies in right subtree.

C

```
/* Program to find LCA of n1 and n2 using one traversal of Binary Tree */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};

// Utility function to create a new tree Node
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// This function returns pointer to LCA of two given values n1 and n2.
// This function assumes that n1 and n2 are present in Binary Tree
struct Node *findLCA(struct Node* root, int n1, int n2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
    // ancestor of other, then the ancestor key becomes LCA
    if (root->key == n1 || root->key == n2)
        return root;

    // Look for keys in left and right subtrees
    Node *left_lca = findLCA(root->left, n1, n2);
    Node *right_lca = findLCA(root->right, n1, n2);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca && right_lca) return root;

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != NULL)? left_lca: right_lca;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5)->key;
    cout << "\nLCA(4, 6) = " << findLCA(root, 4, 6)->key;
    cout << "\nLCA(3, 4) = " << findLCA(root, 3, 4)->key;
    cout << "\nLCA(2, 4) = " << findLCA(root, 2, 4)->key;
    return 0;
}
```

1

Java

```

//Java implementation to find lowest common ancestor of
// n1 and n2 using one traversal of binary tree

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

public class BinaryTree
{
    //Root of the Binary Tree
    Node root;

    Node findLCA(int n1, int n2)
    {
        return findLCA(root, n1, n2);
    }

    // This function returns pointer to LCA of two given
    // values n1 and n2. This function assumes that n1 and
    // n2 are present in Binary Tree
    Node findLCA(Node node, int n1, int n2)
    {
        // Base case
        if (node == null)
            return null;

        // If either n1 or n2 matches with root's key, report
        // the presence by returning root (Note that if a key is
        // ancestor of other, then the ancestor key becomes LCA
        if (node.data == n1 || node.data == n2)
            return node;

        // Look for keys in left and right subtrees
        Node left_lca = findLCA(node.left, n1, n2);
        Node right_lca = findLCA(node.right, n1, n2);

        // If both of the above calls return Non-NULL, then one key
        // is present in once subtree and other is present in other,
        // So this node is the LCA
        if (left_lca!=null && right_lca!=null)
            return node;

        // Otherwise check if left subtree or right subtree is LCA
        return (left_lca != null) ? left_lca : right_lca;
    }
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
    tree.root.right.right = new Node(7);
    System.out.println("LCA(4, 5) = " +
        tree.findLCA(4, 5).data);
}

```

```

        System.out.println("LCA(4, 6) = " +
            tree.findLCA(4, 6).data);
        System.out.println("LCA(3, 4) = " +
            tree.findLCA(3, 4).data);
        System.out.println("LCA(2, 4) = " +
            tree.findLCA(2, 4).data);
    }
}

```

Python

```

# Python program to find LCA of n1 and n2 using one
# traversal of Binary tree

# A binary tree node
class Node:

    # Constructor to create a new tree node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    # This function returns pointer to LCA of two given
    # values n1 and n2
    # This function assumes that n1 and n2 are present in
    # Binary Tree
    def findLCA(root, n1, n2):

        # Base Case
        if root is None:
            return None

        # If either n1 or n2 matches with root's key, report
        # the presence by returning root (Note that if a key is
        # ancestor of other, then the ancestor key becomes LCA
        if root.key == n1 or root.key == n2:
            return root

        # Look for keys in left and right subtrees
        left_lca = findLCA(root.left, n1, n2)
        right_lca = findLCA(root.right, n1, n2)

        # If both of the above calls return Non-NULL, then one key
        # is present in once subtree and other is present in other,
        # So this node is the LCA
        if left_lca and right_lca:
            return root

        # Otherwise check if left subtree or right subtree is LCA
        return left_lca if left_lca is not None else right_lca

# Driver program to test above function

# Let us create a binary tree given in the above example
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
print "LCA(4,5) = ", findLCA(root, 4, 5).key
print "LCA(4,6) = ", findLCA(root, 4, 6).key
print "LCA(3,4) = ", findLCA(root, 3, 4).key
print "LCA(2,4) = ", findLCA(root, 2, 4).key

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```
LCA(4, 5) = 2
LCA(4, 6) = 1
LCA(3, 4) = 1
LCA(2, 4) = 2
```

Thanks to *Atul Singh* for suggesting this solution.

Time Complexity: Time complexity of the above solution is O(n) as the method does a simple tree traversal in bottom up fashion.

Note that the above method assumes that keys are present in Binary Tree. If one key is present and other is absent, then it returns the present key as LCA (Ideally should have returned NULL).

We can extend this method to handle all cases by passing two boolean variables v1 and v2. v1 is set as true when n1 is present in tree and v2 is set as true if n2 is present in tree.

C

```
/* Program to find LCA of n1 and n2 using one traversal of Binary Tree.
   It handles all cases even when n1 or n2 is not there in Binary Tree */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};

// Utility function to create a new tree Node
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// This function returns pointer to LCA of two given values n1 and n2.
// v1 is set as true by this function if n1 is found
// v2 is set as true by this function if n2 is found
struct Node *findLCAUtil(struct Node* root, int n1, int n2, bool &v1, bool &v2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report the presence
    // by setting v1 or v2 as true and return root (Note that if a key
    // is ancestor of other, then the ancestor key becomes LCA)
    if (root->key == n1)
    {
        v1 = true;
        return root;
    }
    if (root->key == n2)
    {
        v2 = true;
        return root;
    }

    // Look for keys in left and right subtrees
    Node *left_lca = findLCAUtil(root->left, n1, n2, v1, v2);
    Node *right_lca = findLCAUtil(root->right, n1, n2, v1, v2);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca && right_lca) return root;

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != NULL)? left_lca: right_lca;
}
```

```

// Returns true if key k is present in tree rooted with root
bool find(Node *root, int k)
{
    // Base Case
    if (root == NULL)
        return false;

    // If key is present at root, or in left subtree or right subtree,
    // return true;
    if (root->key == k || find(root->left, k) || find(root->right, k))
        return true;

    // Else return false
    return false;
}

// This function returns LCA of n1 and n2 only if both n1 and n2 are present
// in tree, otherwise returns NULL;
Node *findLCA(Node *root, int n1, int n2)
{
    // Initialize n1 and n2 as not visited
    bool v1 = false, v2 = false;

    // Find lca of n1 and n2 using the technique discussed above
    Node *lca = findLCAUtil(root, n1, n2, v1, v2);

    // Return LCA only if both n1 and n2 are present in tree
    if (v1 && v2 || v1 && find(lca, n2) || v2 && find(lca, n1))
        return lca;

    // Else return NULL
    return NULL;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    Node *lca = findLCA(root, 4, 5);
    if (lca != NULL)
        cout << "LCA(4, 5) = " << lca->key;
    else
        cout << "Keys are not present ";

    lca = findLCA(root, 4, 10);
    if (lca != NULL)
        cout << "\nLCA(4, 10) = " << lca->key;
    else
        cout << "\nKeys are not present ";

    return 0;
}

```

Java

```

// Java implementation to find lowest common ancestor of
// n1 and n2 using one traversal of binary tree
// It also handles cases even when n1 and n2 are not there in Tree

/* Class containing left and right child of current node and key */
class Node
{
    int data;
    Node left, right;

    public Node(int item)

```

```

    {
        data = item;
        left = right = null;
    }
}

public class BinaryTree
{
    // Root of the Binary Tree
    Node root;
    static boolean v1 = false, v2 = false;

    // This function returns pointer to LCA of two given
    // values n1 and n2.
    // v1 is set as true by this function if n1 is found
    // v2 is set as true by this function if n2 is found
    Node findLCAUtil(Node node, int n1, int n2)
    {
        // Base case
        if (node == null)
            return null;

        // If either n1 or n2 matches with root's key, report the presence
        // by setting v1 or v2 as true and return root (Note that if a key
        // is ancestor of other, then the ancestor key becomes LCA)
        if (node.data == n1)
        {
            v1 = true;
            return node;
        }
        if (node.data == n2)
        {
            v2 = true;
            return node;
        }

        // Look for keys in left and right subtrees
        Node left_lca = findLCAUtil(node.left, n1, n2);
        Node right_lca = findLCAUtil(node.right, n1, n2);

        // If both of the above calls return Non-NULL, then one key
        // is present in once subtree and other is present in other,
        // So this node is the LCA
        if (left_lca != null && right_lca != null)
            return node;

        // Otherwise check if left subtree or right subtree is LCA
        return (left_lca != null) ? left_lca : right_lca;
    }

    // Finds lca of n1 and n2 under the subtree rooted with 'node'
    Node findLCA(int n1, int n2)
    {
        // Initialize n1 and n2 as not visited
        v1 = false;
        v2 = false;

        // Find lca of n1 and n2 using the technique discussed above
        Node lca = findLCAUtil(root, n1, n2);

        // Return LCA only if both n1 and n2 are present in tree
        if (v1 && v2)
            return lca;

        // Else return NULL
        return null;
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
    }
}

```

```

tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);
tree.root.right.left = new Node(6);
tree.root.right.right = new Node(7);

Node lca = tree.findLCA(4, 5);
if (lca != null)
    System.out.println("LCA(4, 5) = " + lca.data);
else
    System.out.println("Keys are not present");

lca = tree.findLCA(4, 10);
if (lca != null)
    System.out.println("LCA(4, 10) = " + lca.data);
else
    System.out.println("Keys are not present");
}
}

```

Python

```

""" Program to find LCA of n1 and n2 using one traversal of
Binary tree
It handles all cases even when n1 or n2 is not there in tree
"""

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    # This function return pointer to LCA of two given values
    # n1 and n2
    # v1 is set as true by this function if n1 is found
    # v2 is set as true by this function if n2 is found
    def findLCAUtil(self, n1, n2, v):

        # Base Case
        if self is None:
            return None

        # If either n1 or n2 matches ith root's key, report
        # the presence by setting v1 or v2 as true and return
        # root (Note that if a key is ancestor of other, then
        # the ancestor key becomes LCA)
        if self.key == n1 :
            v[0] = True
            return self

        if self.key == n2:
            v[1] = True
            return self

        # Look for keys in left and right subtree
        left_lca = self.findLCAUtil(self.left, n1, n2, v)
        right_lca = self.findLCAUtil(self.right, n1, n2, v)

        # If both of the above calls return Non-NULL, then one key
        # is present in once subtree and other is present in other,
        # So this node is the LCA
        if left_lca and right_lca:
            return self

        # Otherwise check if left subtree or right subtree is LCA
        return left_lca if left_lca is not None else right_lca

def find(self, k):

```

```

# Base Case
if root is None:
    return False

# If key is present at root, or if left subtree or right
# subtree , return true
if (root.key == k or find(root.left, k) or
    find(root.right, k)):
    return True

# Else return false
return False

# This function returns LCA of n1 and n2 onlue if both
# n1 and n2 are present in tree, otherwise returns None
def findLCA(root, n1, n2):

    # Initialize n1 and n2 as not visited
    v = [False, False]

    # Find lac of n1 and n2 using the technique discussed above
    lca = findLCAUtil(root, n1, n2, v)

    # Returns LCA only if both n1 and n2 are present in tree
    if (v[0] and v[1] or v[0] and find(lca, n2) or v[1] and
        find(lca, n1)):
        return lca

    # Else return None
    return None

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

lca = findLCA(root, 4, 5)

if lca is not None:
    print "LCA(4, 5) = ", lca.key
else :
    print "Keys are not present"

lca = findLCA(root, 4, 10)
if lca is not None:
    print "LCA(4,10) = ", lca.key
else:
    print "Keys are not present"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

LCA(4, 5) = 2
Keys are not present

```

Thanks to Dhruv for suggesting this extended solution.

You may like to see below articles as well :

[LCA using Parent Pointer](#)

[Lowest Common Ancestor in a Binary Search Tree.](#)

[Find LCA in Binary Tree using RMQ](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Design and Implement Special Stack Data Structure | Added Space Optimized Version

Question: Design a Data Structure SpecialStack that supports all the stack operations like push(), pop(), isEmpty(), isFull() and an additional operation getMin() which should return minimum element from the SpecialStack. All these operations of SpecialStack must be O(1). To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, list, ... etc.

Example:

Consider the following SpecialStack

16 --> TOP

15

29

19

18

When getMin() is called it should return 15, which is the minimum element in the current stack.

If we do pop two times on stack, the stack becomes

29 --> TOP

19

18

When getMin() is called, it should return 18 which is the minimum in the current stack.

Solution: Use two stacks: one to store actual stack elements and other as an auxiliary stack to store minimum values. The idea is to do push() and pop() operations in such a way that the top of auxiliary stack is always the minimum. Let us see how push() and pop() operations work.

Push(int x) // inserts an element x to Special Stack

- 1) push x to the first stack (the stack with actual elements)
- 2) compare x with the top element of the second stack (the auxiliary stack). Let the top element be y.
 -a) If x is smaller than y then push x to the auxiliary stack.
 -b) If x is greater than y then push y to the auxiliary stack.

int Pop() // removes an element from Special Stack and return the removed element

- 1) pop the top element from the auxiliary stack.
- 2) pop the top element from the actual stack and return it.

The step 1 is necessary to make sure that the auxiliary stack is also updated for future operations.

int getMin() // returns the minimum element from Special Stack

- 1) Return the top element of auxiliary stack.

We can see that **all above operations are O(1)**.

Let us see an example. Let us assume that both stacks are initially empty and 18, 19, 29, 15 and 16 are inserted to the SpecialStack.

When we insert 18, both stacks change to following.

Actual Stack

18

Following is C++ implementation for SpecialStack class. In the below implementation, SpecialStack inherits from Stack and has one Stack object min which stores the minimum value of the stack.

```
#include<iostream>
#include<stdlib.h>

using namespace std;

/* A simple stack class with basic stack functionalities */
class Stack
{
private:
    static const int max = 100;
    int arr[max];
    int top;
public:
    Stack() { top = -1; }
```

```

bool isEmpty();
bool isFull();
int pop();
void push(int x);
};

/* Stack's member method to check if the stack is empty */
bool Stack::isEmpty()
{
    if(top == -1)
        return true;
    return false;
}

/* Stack's member method to check if the stack is full */
bool Stack::isFull()
{
    if(top == max - 1)
        return true;
    return false;
}

/* Stack's member method to remove an element from it */
int Stack::pop()
{
    if(isEmpty())
    {
        cout<<"Stack Underflow";
        abort();
    }
    int x = arr[top];
    top--;
    return x;
}

/* Stack's member method to insert an element to it */
void Stack::push(int x)
{
    if(isFull())
    {
        cout<<"Stack Overflow";
        abort();
    }
    top++;
    arr[top] = x;
}

/* A class that supports all the stack operations and one additional
operation getMin() that returns the minimum element from stack at
any time. This class inherits from the stack class and uses an
auxiliarry stack that holds minimum elements */
class SpecialStack: public Stack
{
    Stack min;
public:
    int pop();
    void push(int x);
    int getMin();
};

/* SpecialStack's member method to insert an element to it. This method
makes sure that the min stack is also updated with appropriate minimum
values */
void SpecialStack::push(int x)
{
    if(isEmpty()==true)
    {
        Stack::push(x);
        min.push(x);
    }
    else
    {
        Stack::push(x);
        int y = min.pop();
        min.push(y);
    }
}

```

```

        if( x < y )
            min.push(x);
        else
            min.push(y);
    }
}

/* SpecialStack's member method to remove an element from it. This method
   removes top element from min stack also. */
int SpecialStack::pop()
{
    int x = Stack::pop();
    min.pop();
    return x;
}

/* SpecialStack's member method to get minimum element from it. */
int SpecialStack::getMin()
{
    int x = min.pop();
    min.push(x);
    return x;
}

/* Driver program to test SpecialStack methods */
int main()
{
    SpecialStack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout<<s.getMin()<<endl;
    s.push(5);
    cout<<s.getMin();
    return 0;
}

```

Output:

10

5

Space Optimized Version

The above approach can be optimized. We can limit the number of elements in auxiliary stack. We can push only when the incoming element of main stack is smaller than or equal to top of auxiliary stack.

```

/* SpecialStack's member method to insert an element to it. This method
   makes sure that the min stack is also updated with appropriate minimum
   values */
void SpecialStack::push(int x)
{
    if(isEmpty()==true)
    {
        Stack::push(x);
        min.push(x);
    }
    else
    {
        Stack::push(x);
        int y = min.pop();
        min.push(y);

        /* push only when the incoming element of main stack is smaller
           than or equal to top of auxiliary stack */
        if( x <= y )
            min.push(x);
    }
}

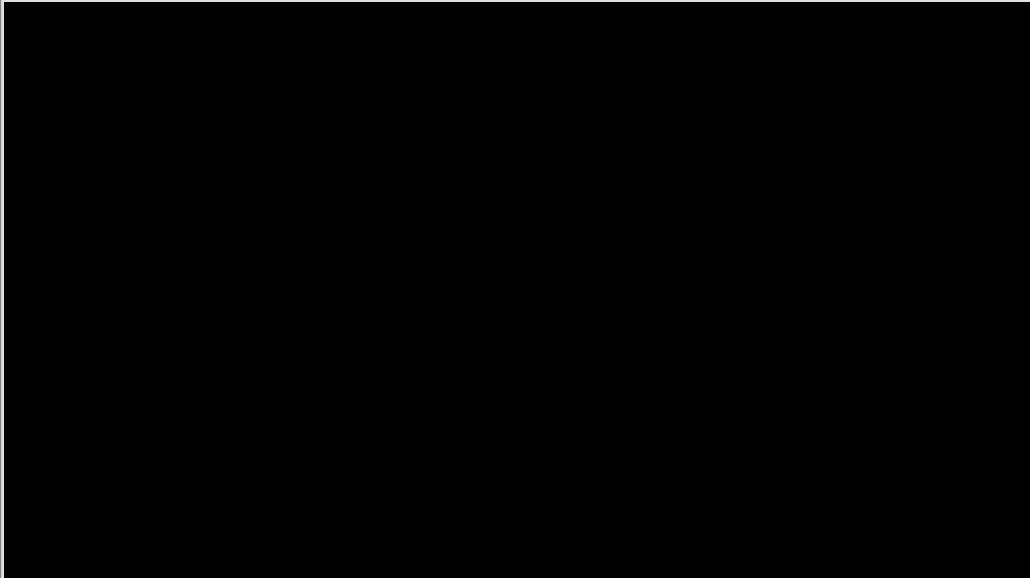
/* SpecialStack's member method to remove an element from it. This method
   removes top element from min stack also. */

```

```
int SpecialStack::pop()
{
    int x = Stack::pop();
    int y = min.pop();

    /* Push the popped element y back only if it is not equal to x */
    if (y != x)
        min.push(y);

    return x;
}
```

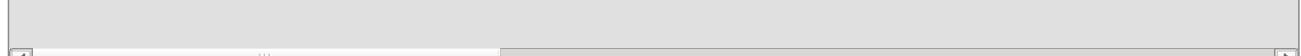


Thanks to @Venki, @swarup and @Jing Huang for their inputs.

Please write comments if you find the above code incorrect, or find other ways to solve the same problem.

GATE CS Corner

Company Wise Coding Practice



Stack
stack
Stack-Queue
StackAndQueue

Reverse a Linked List in groups of given size

Given a linked list, write a function to reverse every k nodes (where k is an input to the function).

Example:

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 3

Output: 3->2->1->6->5->4->8->7->NULL.

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 5

Output: 5->4->3->2->1->8->7->6->NULL.

Algorithm: *reverse(head, k)*

- 1) Reverse the first sub-list of size k. While reversing keep track of the next node and previous node. Let the pointer to the next node be *next* and pointer to the previous node be *prev*. See [this post](#) for reversing a linked list.
- 2) *head->next = reverse(next, k)* /* Recursively call for rest of the list and link the two sub-lists */
- 3) return *prev* /* *prev* becomes the new head of the list (see the diagrams of iterative method of [this post](#)) */

C/C++

```
// C program to reverse a linked list in groups of given size
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Reverses the linked list in groups of size k and returns the
   pointer to the new head node. */
struct node *reverse (struct node *head, int k)
{
    struct node* current = head;
    struct node* next = NULL;
    struct node* prev = NULL;
    int count = 0;

    /*reverse first k nodes of the linked list*/
    while (current != NULL && count < k)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
        count++;
    }

    /* next is now a pointer to (k+1)th node
       Recursively call for the list starting from current.
       And make rest of the list as next of first node */
    if (next != NULL)
        head->next = reverse(next, k);

    /* prev is new head of the input list*/
    return prev;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}
```

```

}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Created Linked list is 1->2->3->4->5->6->7->8->9 */
    push(&head, 9);
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\nGiven linked list \n");
    printList(head);
    head = reverse(head, 3);

    printf("\nReversed Linked list \n");
    printList(head);

    return(0);
}

```

Java

```

// Java program to reverse a linked list in groups of
// given size
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    Node reverse(Node head, int k)
    {
        Node current = head;
        Node next = null;
        Node prev = null;

        int count = 0;

        /* Reverse first k nodes of linked list */
        while (count < k && current != null)
        {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
            count++;
        }
    }
}

```

```

/* next is now a pointer to (k+1)th node
   Recursively call for the list starting from current.
   And make rest of the list as next of first node */
if (next != null)
    head.next = reverse(next, k);

// prev is now head of input list
return prev;
}

/* Utility functions */

/* Inserts a new Node at front of the list.*/
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* Function to print linked list */
void printList()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Driver program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    /* Constructed Linked List is 1->2->3->4->5->6->
       7->8->8->9->null */
    llist.push(9);
    llist.push(8);
    llist.push(7);
    llist.push(6);
    llist.push(5);
    llist.push(4);
    llist.push(3);
    llist.push(2);
    llist.push(1);

    System.out.println("Given Linked List");
    llist.printList();

    llist.head = llist.reverse(llist.head, 3);

    System.out.println("Reversed list");
    llist.printList();
}

/* This code is contributed by Rajat Mishra */

```

Python

```

# Python program to reverse a linked list in group of given size

# Node class

```

```

class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def reverse(self, head, k):
        current = head
        next = None
        prev = None
        count = 0

        # Reverse first k nodes of the linked list
        while(current is not None and count < k):
            next = current.next
            current.next = prev
            prev = current
            current = next
            count += 1

        # next is now a pointer to (k+1)th node
        # recursively call for the list starting
        # from current . And make rest of the list as
        # next of first node
        if next is not None:
            head.next = self.reverse(next, k)

        # prev is new head of the input list
        return prev

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

# Driver program
llist = LinkedList()
llist.push(9)
llist.push(8)
llist.push(7)
llist.push(6)
llist.push(5)
llist.push(4)
llist.push(3)
llist.push(2)
llist.push(1)

print "Given linked list"
llist.printList()
llist.head = llist.reverse(llist.head, 3)

print "\nReversed Linked list"
llist.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```
Given Linked List  
1 2 3 4 5 6 7 8 9  
Reversed list  
3 2 1 6 5 4 9 8 7
```

Time Complexity: O(n) where n is the number of nodes in the given list.

Please write comments if you find the above code/algorithm incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Linked Lists
Adobe-Question
Amazon-Question
Reverse
Snapdeal-Question
Yatra.com-Question

Add two numbers represented by linked lists | Set 1

Given two numbers represented by two lists, write a function that returns sum list. The sum list is list representation of addition of two input numbers.

Example 1

```
Input:  
First List: 5->6->3 // represents number 365  
Second List: 8->4->2 // represents number 248  
Output  
Resultant list: 3->1->6 // represents number 613
```

Example 2

```
Input:  
First List: 7->5->9->4->6 // represents number 64957  
Second List: 8->4 // represents number 48  
Output  
Resultant list: 5->0->0->5->6 // represents number 65005
```

Solution

Traverse both lists. One by one pick nodes of both lists and add the values. If sum is more than 10 then make carry as 1 and reduce sum. If one list has more elements than the other then consider remaining values of this list as 0. Following is the implementation of this approach.

C

```
#include<stdio.h>  
#include<stdlib.h>  
  
/* Linked list node */  
struct node  
{  
    int data;  
    struct node* next;  
};  
  
/* Function to create a new node with given data */  
struct node *newNode(int data)  
{  
    struct node *new_node = (struct node *) malloc(sizeof(struct node));  
    new_node->data = data;  
    new_node->next = NULL;  
    return new_node;  
}  
  
/* Function to insert a node at the beginning of the Doubly Linked List */  
void push(struct node** head_ref, int new_data)  
{  
    /* allocate node */  
    struct node* new_node = newNode(new_data);  
  
    /* link the old list off the new node */  
    new_node->next = (*head_ref);
```

```

/* move the head to point to the new node */
(*head_ref) = new_node;
}

/* Adds contents of two linked lists and return the head node of resultant list */
struct node* addTwoLists (struct node* first, struct node* second)
{
    struct node* res = NULL; // res is head node of the resultant list
    struct node *temp, *prev = NULL;
    int carry = 0, sum;

    while (first != NULL || second != NULL) //while both lists exist
    {
        // Calculate value of next digit in resultant list.
        // The next digit is sum of following things
        // (i) Carry
        // (ii) Next digit of first list (if there is a next digit)
        // (iii) Next digit of second list (if there is a next digit)
        sum = carry + (first? first->data: 0) + (second? second->data: 0);

        // update carry for next calculation
        carry = (sum >= 10)? 1 : 0;

        // update sum if it is greater than 10
        sum = sum % 10;

        // Create a new node with sum as data
        temp = newNode(sum);

        // if this is the first node then set it as head of the resultant list
        if(res == NULL)
            res = temp;
        else // If this is not the first node then connect it to the rest.
            prev->next = temp;

        // Set prev for next insertion
        prev = temp;

        // Move first and second pointers to next nodes
        if (first) first = first->next;
        if (second) second = second->next;
    }

    if (carry > 0)
        temp->next = newNode(carry);

    // return head of the resultant list
    return res;
}

// A utility function to print a linked list
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Driver program to test above function */
int main(void)
{
    struct node* res = NULL;
    struct node* first = NULL;
    struct node* second = NULL;

    // create first list 7->5->9->4->6
    push(&first, 6);
    push(&first, 4);
    push(&first, 9);
    push(&first, 5);
    push(&first, 7);
    printf("First List is ");
}

```

```

printList(first);

// create second list 8->4
push(&second, 4);
push(&second, 8);
printf("Second List is ");
printList(second);

// Add the two lists and see result
res = addTwoLists(first, second);
printf("Resultant list is ");
printList(res);

return 0;
}

```

Java

```

// Java program to delete a given node in linked list under given constraints

class LinkedList {

    static Node head1, head2;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    /* Adds contents of two linked lists and return the head node of resultant list */
    Node addTwoLists(Node first, Node second) {
        Node res = null; // res is head node of the resultant list
        Node prev = null;
        Node temp = null;
        int carry = 0, sum;

        while (first != null || second != null) //while both lists exist
        {
            // Calculate value of next digit in resultant list.
            // The next digit is sum of following things
            // (i) Carry
            // (ii) Next digit of first list (if there is a next digit)
            // (iii) Next digit of second list (if there is a next digit)
            sum = carry + (first != null ? first.data : 0)
                + (second != null ? second.data : 0);

            // update carry for next calculation
            carry = (sum >= 10) ? 1 : 0;

            // update sum if it is greater than 10
            sum = sum % 10;

            // Create a new node with sum as data
            temp = new Node(sum);

            // if this is the first node then set it as head of
            // the resultant list
            if (res == null) {
                res = temp;
            } else // If this is not the first node then connect it to the rest.
            {
                prev.next = temp;
            }

            // Set prev for next insertion
            prev = temp;
        }
    }
}

```

```

// Move first and second pointers to next nodes
if (first != null) {
    first = first.next;
}
if (second != null) {
    second = second.next;
}
}

if (carry > 0) {
    temp.next = new Node(carry);
}

// return head of the resultant list
return res;
}
/* Utility function to print a linked list */

void printList(Node head) {
    while (head != null) {
        System.out.print(head.data + " ");
        head = head.next;
    }
    System.out.println("");
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    // creating first list
    list.head1 = new Node(7);
    list.head1.next = new Node(5);
    list.head1.next.next = new Node(9);
    list.head1.next.next.next = new Node(4);
    list.head1.next.next.next.next = new Node(6);
    System.out.print("First List is ");
    list.printList(head1);

    // creating second list
    list.head2 = new Node(8);
    list.head2.next = new Node(4);
    System.out.print("Second List is ");
    list.printList(head2);

    // add the two lists and see the result
    Node rs = list.addTwoLists(head1, head2);
    System.out.print("Resultant List is ");
    list.printList(rs);
}
}

// this code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to add two numbers represented by linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

```

```

# Function to insert a new node at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Add contents of two linked lists and return the head
# node of resultant list
def addTwoLists(self, first, second):
    prev = None
    temp = None
    carry = 0

    # While both list exists
    while(first is not None or second is not None):

        # Calculate the value of next digit in
        # resultant list
        # The next digit is sum of following things
        # (i) Carry
        # (ii) Next digit of first list (if ther is a
        # next digit)
        # (iii) Next digit of second list ( if there
        # is a next digit)
        fdata = 0 if first is None else first.data
        sdata = 0 if second is None else second.data
        Sum = carry + fdata + sdata

        # update carry for next calculation
        carry = 1 if Sum >= 10 else 0

        # update sum if it is greater than 10
        Sum = Sum if Sum < 10 else Sum % 10

        # Create a new node with sum as data
        temp = Node(Sum)

        # if this is the first node then set it as head
        # of resultant list
        if self.head is None:
            self.head = temp
        else :
            prev.next = temp

        # Set prev for next insertion
        prev = temp

        # Move first and second pointers to next nodes
        if first is not None:
            first = first.next
        if second is not None:
            second = second.next

        if carry > 0:
            temp.next = Node(carry)

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program to test above function
first = LinkedList()
second = LinkedList()

# Create first list
first.push(6)
first.push(4)
first.push(9)
first.push(5)
first.push(7)
print "First List is ",
first.printList()

```

```

# Create second list
second.push(4)
second.push(8)
print "\nSecond List is ",
second.printList()

# Add the two lists and see result
res = LinkedList()
res.addTwoLists(first.head, second.head)
print "\nResultant list is ",
res.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

First List is 7 5 9 4 6
Second List is 8 4
Resultant list is 5 0 0 5 6

```

Time Complexity: O(m + n) where m and n are number of nodes in first and second lists respectively.

Related Article : [Add two numbers represented by linked lists | Set 2](#)

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Inplace rotate square matrix by 90 degrees

Given an square matrix, turn it by 90 degrees in anti-clockwise direction without using any extra space.

Examples:

```

Input
1 2 3
4 5 6
7 8 9

```

```

Output:
3 6 9
2 5 8
1 4 7

```

```

Input:
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

```

```

Output:
4 8 12 16
3 7 11 15
2 6 10 14
1 5 9 13

```

We strongly recommend you to minimize your browser and try this yourself first.

An approach that requires extra space is already discussed [here](#).

How to do without extra space?

Below are some important observations.

First row of source → First column of destination, elements filled in opposite order

Second row of source → Second column of destination, elements filled in opposite order

so ... on

Last row of source → Last column of destination, elements filled in opposite order.

An $N \times N$ matrix will have $\text{floor}(N/2)$ square cycles. For example, a 4×4 matrix will have 2 cycles. The first cycle is formed by its 1st row, last column, last row and 1st column. The second cycle is formed by 2nd row, second-last column, second-last row and 2nd column.

The idea is for each square cycle, we swap the elements involved with the corresponding cell in the matrix in anti-clockwise direction i.e. from top to left, left to bottom, bottom to right and from right to top one at a time. We use nothing but a temporary variable to achieve this.

Below steps demonstrate the idea

First Cycle (Involves Red Elements)

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

Moving first group of four elements (First elements of 1st row, last row, 1st column and last column) of first cycle in counter clockwise.

4 2 3 16
5 6 7 8
9 10 11 12
1 14 15 13

Moving next group of four elements of first cycle in counter clockwise

4 8 3 16
5 6 7 15
2 10 11 12
1 14 9 13

Moving final group of four elements of first cycle in counter clockwise

4 8 12 16
3 6 7 15
2 10 11 14
1 5 9 13

Second Cycle (Involves Blue Elements)

4 8 12 16
3 6 7 15
2 10 11 14
1 5 9 13

Fixing second cycle

4 8 12 16
3 7 11 15
2 6 10 14
1 5 9 13

Below is C++ implementation of above idea.

```
// C++ program to rotate a matrix by 90 degrees
#include <bits/stdc++.h>
#define N 4
using namespace std;

void displayMatrix(int mat[N][N]);

// An Inplace function to rotate a N x N matrix
// by 90 degrees in anti-clockwise direction
void rotateMatrix(int mat[N][N])
{
    // Consider all squares one by one
    for (int x = 0; x < N / 2; x++)
    {
        // Consider elements in group of 4 in
        // current square
        for (int y = x; y < N - x - 1; y++)
        {
            // store current cell in temp variable
            int temp = mat[x][y];

            // move values from right to top
            for (int i = x + 1; i <= N - y - 1; i++)
                mat[i][y] = mat[i + 1][y];
```

```

mat[x][y] = mat[y][N-1-x];

// move values from bottom to right
mat[y][N-1-x] = mat[N-1-x][N-1-y];

// move values from left to bottom
mat[N-1-x][N-1-y] = mat[N-1-y][x];

// assign temp to left
mat[N-1-y][x] = temp;
}

}

}

// Function to print the matrix
void displayMatrix(int mat[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf("%2d ", mat[i][j]);

        printf("\n");
    }
    printf("\n");
}

/* Driver program to test above functions */
int main()
{
    // Test Case 1
    int mat[N][N] =
    {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12},
        {13, 14, 15, 16}
    };

    // Test Case 2
    /* int mat[N][N] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    }; */

    // Test Case 3
    /*int mat[N][N] = {
        {1, 2},
        {4, 5}
    };*/

    //displayMatrix(mat);

    rotateMatrix(mat);

    // Print rotated matrix
    displayMatrix(mat);

    return 0;
}

```

Output:

Rotated Matrix
4 8 12 16
3 7 11 15
2 6 10 14
1 5 9 13

Exercise: Turn 2D matrix by 90 degrees in clockwise direction without using extra space.

This article is contributed by **Aditya Goel**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

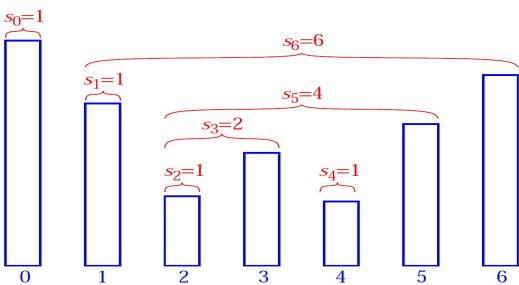
Matrix
rotation

The Stock Span Problem

The stock span problem is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock's price for all n days.

The span S_i of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.

For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}



We strongly recommend that you click here and practice it, before moving on to the solution.

A Simple but inefficient method

Traverse the input price array. For every element being visited, traverse elements on left of it and increment the span value of it while elements on the left side are smaller.

Following is implementation of this method.

C

```
// C program for brute force method to calculate stock span values
#include <stdio.h>

// Fills array S[] with span values
void calculateSpan(int price[], int n, int S[])
{
    // Span value of first day is always 1
    S[0] = 1;

    // Calculate span value of remaining days by linearly checking
    // previous days
    for (int i = 1; i < n; i++)
    {
        S[i] = 1; // Initialize span value

        // Traverse left while the next element on left is smaller
        // than price[i]
        for (int j = i-1; (j>=0)&&(price[i]>=price[j]); j--)
            S[i]++;
    }
}

// A utility function to print elements of array
void printArray(int arr[], int n)
```

```

{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
}

// Driver program to test above function
int main()
{
    int price[] = {10, 4, 5, 90, 120, 80};
    int n = sizeof(price)/sizeof(price[0]);
    int S[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S, n);

    return 0;
}

```

Python

```

# Python program for brute force method to calculate stock span values

# Fills list S[] with span values
def calculateSpan(price, n, S):

    # Span value of first day is always 1
    S[0] = 1

    # Calculate span value of remaining days by linearly
    # checking previous days
    for i in range(1, n, 1):
        S[i] = 1 # Initialize span value

        # Traverse left while the next element on left is
        # smaller than price[i]
        j = i - 1
        while (j >= 0) and (price[i] >= price[j]) :
            S[i] += 1
            j -= 1

# A utility function to print elements of array
def printArray(arr, n):

    for i in range(n):
        print(arr[i], end = " ")

# Driver program to test above function
price = [10, 4, 5, 90, 120, 80]
n = len(price)
S = [None] * n

# Fill the span values in list S[]
calculateSpan(price, n, S)

# print the calculated span values
printArray(S, n)

# This code is contributed by Sunny Karira

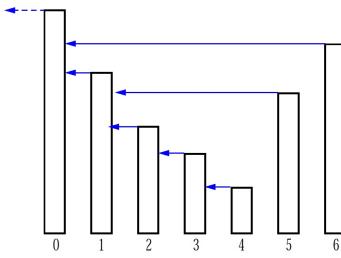
```

Time Complexity of the above method is $O(n^2)$. We can calculate stock span values in $O(n)$ time.

A Linear Time Complexity Method

We see that $S[i]$ on day i can be easily computed if we know the closest day preceding i , such that the price is greater than on that day than the price on day i . If such a day exists, let's call it $h(i)$, otherwise, we define $h(i) = -1$.

The span is now computed as $S[i] = i - h(i)$. See the following diagram.



To implement this logic, we use a stack as an abstract data type to store the days i, h(i), h(h(i)) and so on. When we go from day i-1 to i, we pop the days when the price of the stock was less than or equal to price[i] and then push the value of day i back into the stack.

Following is C++ implementation of this method.

C++

```
// a linear time solution for stock span problem
#include <iostream>
#include <stack>
using namespace std;

// A stack based efficient method to calculate stock span values
void calculateSpan(int price[], int n, int S[])
{
    // Create a stack and push index of first element to it
    stack<int> st;
    st.push(0);

    // Span value of first element is always 1
    S[0] = 1;

    // Calculate span values for rest of the elements
    for (int i = 1; i < n; i++)
    {
        // Pop elements from stack while stack is not empty and top of
        // stack is smaller than price[i]
        while (!st.empty() && price[st.top()] <= price[i])
            st.pop();

        // If stack becomes empty, then price[i] is greater than all elements
        // on left of it, i.e., price[0], price[1]..price[i-1]. Else price[i]
        // is greater than elements after top of stack
        S[i] = (st.empty()) ? (i + 1) : (i - st.top());

        // Push this element to stack
        st.push(i);
    }

    // A utility function to print elements of array
    void printArray(int arr[], int n)
    {
        for (int i = 0; i < n; i++)
            cout << arr[i] << " ";
    }

    // Driver program to test above function
    int main()
    {
        int price[] = {10, 4, 5, 90, 120, 80};
        int n = sizeof(price)/sizeof(price[0]);
        int S[n];

        // Fill the span values in array S[]
        calculateSpan(price, n, S);

        // print the calculated span values
        printArray(S, n);

        return 0;
    }
}
```

Python

```
# A linear time solution for stack stock problem

# A stack based efficient method to calculate s
def calculateSpan(price, S):

    n = len(price)
    # Create a stack and push index of fist element to it
    st = []
    st.append(0)

    # Span value of first element is always 1
    S[0] = 1

    # Calculate span values for rest of the elements
    for i in range(1, n):

        # Pop elements from stack whlie stack is not
        # empty and top of stack is smaller than price[i]
        while( len(st) > 0 and price[st[0]] <= price[i]):
            st.pop()

        # If stack becomes empty, then price[i] is greater
        # than all elements on left of it, i.e. price[0],
        # price[1], ..price[i-1]. Else the price[i] is
        # greater than elements after top of stack
        # S[i] = i+1 if len(st) <= 0 else (i - st[0])

        # Push this element to stack
        st.append(i)

# A utility function to print elements of array
def printArray(arr, n):
    for i in range(0,n):
        print arr[i],


# Driver program to test above function
price = [10, 4, 5, 90, 120, 80]
S = [0 for i in range(len(price)+1)]

# Fill the span values in array S[]
calculateSpan(price, S)

# Print the calculated span values
printArray(S, len(price))

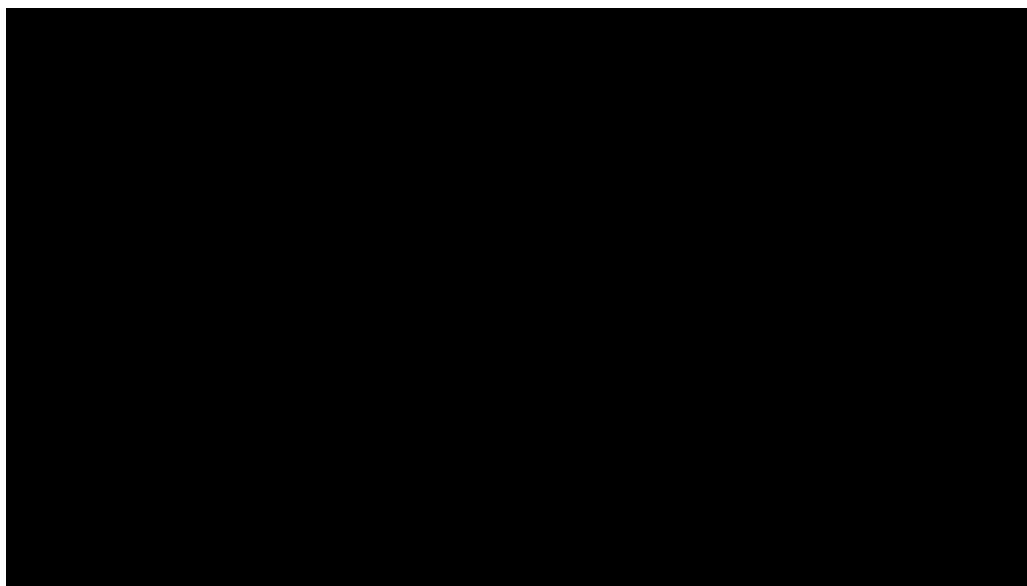
# This code is contributed by Nikhil Kumar Singh (nickzuck_007)
```

Output:

```
1 1 2 4 5 1
```

Time Complexity: O(n). It seems more than O(n) at first look. If we take a closer look, we can observe that every element of array is added and removed from stack at most once. So there are total 2n operations at most. Assuming that a stack operation takes O(1) time, we can say that the time complexity is O(n).

Auxiliary Space: O(n) in worst case when all elements are sorted in decreasing order.



References:

[http://en.wikipedia.org/wiki/Stack_\(abstract_data_type\)#The_Stock_Span_Problem](http://en.wikipedia.org/wiki/Stack_(abstract_data_type)#The_Stock_Span_Problem)
<http://crypto.cs.mcgill.ca/~crepeau/CS250/2004/Stack-I.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Stack
stack

Next Greater Element

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

Examples:

- a) For any array, rightmost element always has next greater element as -1.
- b) For an array which is sorted in decreasing order, all elements have next greater element as -1.
- c) For the input array [4, 5, 2, 25], the next greater elements for each element are as follows.

Element	NGE
4	-> 5
5	-> 25
2	-> 25
25	-> -1

- d) For the input array [13, 7, 6, 12], the next greater elements for each element are as follows.

Element	NGE
13	-> -1
7	-> 12
6	-> 12
12	-> -1

Method 1 (Simple)

Use two loops: The outer loop picks all the elements one by one. The inner loop looks for the first greater element for the element picked by outer loop. If a greater element is found then that element is printed as next, otherwise -1 is printed.

Thanks to [Sachin](#) for providing following code.

C/C++

```
// Simple C program to print next greater elements
// in a given array
#include<stdio.h>

/* prints element and NGE pair for all elements of
```

```

arr[] of size n */
void printNGE(int arr[], int n)
{
    int next, i, j;
    for (i=0; i<n; i++)
    {
        next = -1;
        for (j = i+1; j<n; j++)
        {
            if (arr[i] < arr[j])
            {
                next = arr[j];
                break;
            }
        }
        printf("%d -- %d\n", arr[i], next);
    }
}

int main()
{
    int arr[]={11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}

```

Python

```

# Function to print element and NGE pair for all elements of list
def printNGE(arr):

    for i in range(0, len(arr), 1):

        next = -1
        for j in range(i+1, len(arr), 1):
            if arr[i] < arr[j]:
                next = arr[j]
                break

        print(str(arr[i]) + " -- " + str(next))

# Driver program to test above function
arr = [11,13,21,3]
printNGE(arr)

# This code is contributed by Sunny Karira

```

Output:

```

11 -- 13
13 -- 21
21 -- -1
3 -- -1

```

Time Complexity: O(n^2). The worst case occurs when all elements are sorted in decreasing order.

Method 2 (Using Stack)

Thanks to [pchild](#) for suggesting following approach.

- 1) Push the first element to stack.
- 2) Pick rest of the elements one by one and follow following steps in loop.
 -a) Mark the current element as *next*.
 -b) If stack is not empty, then pop an element from stack and compare it with *next*.
 -c) If *next* is greater than the popped element, then *next* is the next greater element for the popped element.
 -d) Keep popping from the stack while the popped element is smaller than *next*. *next* becomes the next greater element for all such popped elements
 -g) If *next* is smaller than the popped element, then push the popped element back.
- 3) After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

C

```
// A Stack based C program to find next greater element
// for all array elements.
#include<stdio.h>
#include<stdlib.h>
#define STACKSIZE 100

// stack structure
struct stack
{
    int top;
    int items[STACKSIZE];
};

// Stack Functions to be used by printNGE()
void push(struct stack *ps, int x)
{
    if (ps->top == STACKSIZE-1)
    {
        printf("Error: stack overflow\n");
        getchar();
        exit(0);
    }
    else
    {
        ps->top += 1;
        int top = ps->top;
        ps->items [top] = x;
    }
}

bool isEmpty(struct stack *ps)
{
    return (ps->top == -1)? true : false;
}

int pop(struct stack *ps)
{
    int temp;
    if (ps->top == -1)
    {
        printf("Error: stack underflow \n");
        getchar();
        exit(0);
    }
    else
    {
        int top = ps->top;
        temp = ps->items [top];
        ps->top -= 1;
        return temp;
    }
}

/* prints element and NGE pair for all elements of
arr[] of size n */
void printNGE(int arr[], int n)
{
    int i = 0;
    struct stack s;
    s.top = -1;
    int element, next;

    /* push the first element to stack */
    push(&s, arr[0]);

    // iterate for rest of the elements
    for (i=1; i<n; i++)
    {
        next = arr[i];

        if (isEmpty(&s) == false)
        {
```

```

// if stack is not empty, then pop an element from stack
element = pop(&s);

/* If the popped element is smaller than next, then
   a) print the pair
   b) keep popping while elements are smaller and
      stack is not empty */
while (element < next)
{
    printf("\n %d --> %d", element, next);
    if(isEmpty(&s) == true)
        break;
    element = pop(&s);
}

/* If element is greater than next, then push
   the element back */
if (element > next)
    push(&s, element);
}

/* push next to stack so that we can find
   next greater for it */
push(&s, next);

/* After iterating over the loop, the remaining
   elements in stack do not have the next greater
   element, so print -1 for them */
while (isEmpty(&s) == false)
{
    element = pop(&s);
    next = -1;
    printf("\n %d -- %d", element, next);
}
}

/* Driver program to test above functions */
int main()
{
    int arr[]={11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}

```

Python

```

# Python program to print next greater element using stack

# Stack Functions to be used by printNGE()
def createStack():
    stack = []
    return stack

def isEmpty(stack):
    return len(stack) == 0

def push(stack, x):
    stack.append(x)

def pop(stack):
    if isEmpty(stack):
        print("Error : stack underflow")
    else:
        return stack.pop()

"""prints element and NGE pair for all elements of
arr[] """
def printNGE(arr):
    s = createStack()
    element = 0

```

```

next = 0

# push the first element to stack
push(s, arr[0])

# iterate for rest of the elements
for i in range(1, len(arr), 1):
    next = arr[i]

    if isEmpty(s) == False:

        # if stack is not empty, then pop an element from stack
        element = pop(s)

        """If the popped element is smaller than next, then
        a) print the pair
        b) keep popping while elements are smaller and
           stack is not empty """
        while element < next :
            print(str(element)+ " -- " + str(next))
            if isEmpty(s) == True :
                break
            element = pop(s)

        """If element is greater than next, then push
           the element back """
        if element > next:
            push(s, element)

    """push next to stack so that we can find
       next greater for it """
    push(s, next)

"""After iterating over the loop, the remaining
   elements in stack do not have the next greater
   element, so print -1 for them """

while isEmpty(s) == False:
    element = pop(s)
    next = -1
    print(str(element) + " -- " + str(next))

# Driver program to test above functions
arr = [11, 13, 21, 3]
printNGE(arr)

# This code is contributed by Sunny Karira

```

Output:

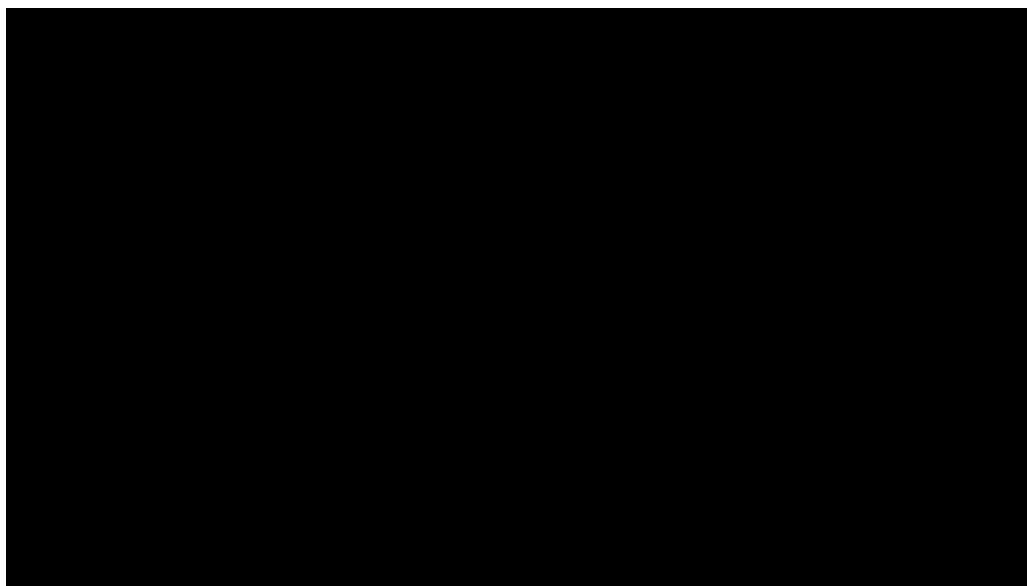
```

11 -- 13
13 -- 21
3 -- -1
21 -- -1

```

Time Complexity: O(n). The worst case occurs when all elements are sorted in decreasing order. If elements are sorted in decreasing order, then every element is processed at most 4 times.

- a) Initially pushed to the stack.
- b) Popped from the stack when next element is being processed.
- c) Pushed back to the stack because next element is smaller.
- d) Popped from the stack in step 3 of algo.



Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Stack

Maximum sum such that no two elements are adjacent

Given an array of positive numbers, find the maximum sum of a subsequence with the constraint that no 2 numbers in the sequence should be adjacent in the array. So 3 2 7 10 should return 13 (sum of 3 and 10) or 3 2 5 10 7 should return 15 (sum of 3, 5 and 7). Answer the question in most efficient way.

Examples :

```
Input : arr[] = {5, 5, 10, 100, 10, 5}
Output : 110
```

```
Input : arr[] = {1, 2, 3}
Output : 4
```

```
Input : arr[] = {1, 20, 3}
Output : 20
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Algorithm:

Loop for all elements in arr[] and maintain two sums incl and excl where incl = Max sum including the previous element and excl = Max sum excluding the previous element.

Max sum excluding the current element will be max(incl, excl) and max sum including the current element will be excl + current element (Note that only excl is considered because elements cannot be adjacent).

At the end of the loop return max of incl and excl.

Example:

```
arr[] = {5, 5, 10, 40, 50, 35}
```

```
inc = 5
exc = 0
```

```
For i = 1 (current element is 5)
incl = (excl + arr[i]) = 5
excl = max(5, 0) = 5
```

```
For i = 2 (current element is 10)
```

```

incl = (excl + arr[i]) = 15
excl = max(5, 5) = 5

For i = 3 (current element is 40)
incl = (excl + arr[i]) = 45
excl = max(5, 15) = 15

For i = 4 (current element is 50)
incl = (excl + arr[i]) = 65
excl = max(45, 15) = 45

For i = 5 (current element is 35)
incl = (excl + arr[i]) = 80
excl = max(5, 15) = 65

And 35 is the last element. So, answer is max(incl, excl) = 80

```

Thanks to [Debanjan](#) for providing code.

Implementation:

C/C++

```

#include<stdio.h>

/*Function to return max sum such that no two elements
are adjacent*/
int FindMaxSum(int arr[], int n)
{
    int incl = arr[0];
    int excl = 0;
    int excl_new;
    int i;

    for (i = 1; i < n; i++)
    {
        /* current max excluding i */
        excl_new = (incl > excl)? incl : excl;

        /* current max including i */
        incl = excl + arr[i];
        excl = excl_new;
    }

    /* return max of incl and excl */
    return ((incl > excl)? incl : excl);
}

/* Driver program to test above function */
int main()
{
    int arr[] = {5, 5, 10, 100, 10, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("%d \n", FindMaxSum(arr, 6));
    return 0;
}

```

Java

```

class MaximumSum
{
    /*Function to return max sum such that no two elements
    are adjacent */
    int FindMaxSum(int arr[], int n)
    {
        int incl = arr[0];
        int excl = 0;
        int excl_new;
        int i;

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

```

```

/* current max excluding i */
excl_new = (incl > excl) ? incl : excl;

/* current max including i */
incl = excl + arr[i];
excl = excl_new;
}

/* return max of incl and excl */
return ((incl > excl) ? incl : excl);
}

// Driver program to test above functions
public static void main(String[] args)
{
    MaximumSum sum = new MaximumSum();
    int arr[] = new int[]{5, 5, 10, 100, 10, 5};
    System.out.println(sum.FindMaxSum(arr, arr.length));
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Function to return max sum such that
# no two elements are adjacent
def find_max_sum(arr):
    incl = 0
    excl = 0

    for i in arr:

        # Current max excluding i (No ternary in
        # Python)
        new_excl = excl if excl>incl else incl

        # Current max including i
        incl = excl + i
        excl = new_excl

    # return max of incl and excl
    return (excl if excl>incl else incl)

# Driver program to test above function
arr = [5, 5, 10, 100, 10, 5]
print find_max_sum(arr)

# This code is contributed by Kalai Selvan

```

Output:

110

Time Complexity: O(n)

Now try the same problem for array with negative numbers also.

Please write comments if you find any bug in the above program/algorithm or other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Arrays
array

Dynamic Programming | Set 5 (Edit Distance)

Given two strings str1 and str2 and below operations that can be performed on str1. Find minimum number of edits (operations) required to convert 'str1' into 'str2'.

- a. Insert

- b. Remove
- c. Replace

All of the above operations are of equal cost.

Examples:

```
Input: str1 = "geek", str2 = "gesek"
Output: 1
We can convert str1 into str2 by inserting a 's'.
```



```
Input: str1 = "cat", str2 = "cut"
Output: 1
We can convert str1 into str2 by replacing 'a' with 'u'.
```



```
Input: str1 = "sunday", str2 = "saturday"
Output: 3
Last three and first characters are same. We basically
need to convert "un" to "atur". This can be done using
below three operations.
Replace 'n' with 't', insert t, insert a
```

What are the subproblems in this case?

The idea is process all characters one by one staring from either from left or right sides of both strings.

Let we traverse from right corner, there are two possibilities for every pair of character being traversed.

m: Length of str1 (first string)
n: Length of str2 (second string)

1. If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
2. Else (If last characters are not same), we consider all operations on 'str1', consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.
 - a. Insert: Recur for m and n-1
 - b. Remove: Recur for m-1 and n
 - c. Replace: Recur for m-1 and n-1

Below is C++ implementation of above Naive recursive solution.

C++

```
// A Naive recursive C++ program to find minimum number
// operations to convert str1 to str2
#include<bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers
int min(int x, int y, int z)
{
    return min(min(x, y), z);
}

int editDist(string str1 ,string str2 ,int m ,int n)
{
    // If first string is empty, the only option is to
    // insert all characters of second string into first
    if (m == 0) return n;

    // If second string is empty, the only option is to
    // remove all characters of first string
    if (n == 0) return m;

    // If last characters of two strings are same, nothing
    // much to do. Ignore last characters and get count for
    // remaining strings.
    if (str1[m-1] == str2[n-1])
        return editDist(str1, str2, m-1, n-1);

    // If last characters are not same, consider all three
    // operations on last character of first string, recursively
```

```

// compute minimum cost for all three operations and take
// minimum of three values.
return 1 + min ( editDist(str1, str2, m, n-1), // Insert
                 editDist(str1, str2, m-1, n), // Remove
                 editDist(str1, str2, m-1, n-1) // Replace
               );
}

// Driver program
int main()
{
    // your code goes here
    string str1 = "sunday";
    string str2 = "saturday";

    cout << editDist( str1 , str2 , str1.length(), str2.length());

    return 0;
}

```

Java

```

// A Naive recursive Java program to find minimum number
// operations to convert str1 to str2
class EDIST
{
    static int min(int x,int y,int z)
    {
        if (x<y && x<z) return x;
        if (y<x && y<z) return y;
        else return z;
    }

    static int editDist(String str1 , String str2 , int m ,int n)
    {
        // If first string is empty, the only option is to
        // insert all characters of second string into first
        if (m == 0) return n;

        // If second string is empty, the only option is to
        // remove all characters of first string
        if (n == 0) return m;

        // If last characters of two strings are same, nothing
        // much to do. Ignore last characters and get count for
        // remaining strings.
        if (str1.charAt(m-1) == str2.charAt(n-1))
            return editDist(str1, str2, m-1, n-1);

        // If last characters are not same, consider all three
        // operations on last character of first string, recursively
        // compute minimum cost for all three operations and take
        // minimum of three values.

        return 1 + min ( editDist(str1, str2, m, n-1), // Insert
                         editDist(str1, str2, m-1, n), // Remove
                         editDist(str1, str2, m-1, n-1) // Replace
                       );
    }

    public static void main(String args[])
    {
        String str1 = "sunday";
        String str2 = "saturday";

        System.out.println( editDist( str1 , str2 , str1.length(), str2.length()) );
    }
}
/*This code is contributed by Rajat Mishra*/

```

Python

```
# A Naive recursive Python program to fin minimum number
```

```

# operations to convert str1 to str2
def editDistance(str1, str2, m ,n):

    # If first string is empty, the only option is to
    # insert all characters of second string into first
    if m==0:
        return n

    # If second string is empty, the only option is to
    # remove all characters of first string
    if n==0:
        return m

    # If last characters of two strings are same, nothing
    # much to do. Ignore last characters and get count for
    # remaining strings.
    if str1[m-1]==str2[n-1]:
        return editDistance(str1,str2,m-1,n-1)

    # If last characters are not same, consider all three
    # operations on last character of first string, recursively
    # compute minimum cost for all three operations and take
    # minimum of three values.
    return 1 + min(editDistance(str1, str2, m, n-1), # Insert
                   editDistance(str1, str2, m-1, n), # Remove
                   editDistance(str1, str2, m-1, n-1) # Replace
                  )

# Driver program to test the above function
str1 = "sunday"
str2 = "saturday"
print editDistance(str1, str2, len(str1), len(str2))

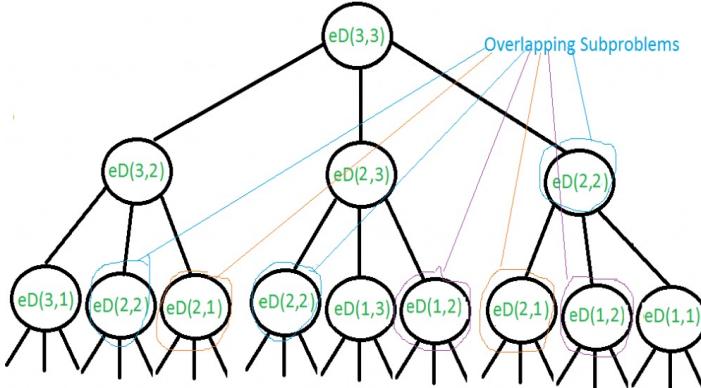
# This code is contributed by Bhavya Jain

```

Output:

3

The time complexity of above solution is exponential. In worst case, we may end up doing $O(3^m)$ operations. The worst case happens when none of characters of two strings match. Below is a recursive call diagram for worst case.



Worst case recursion tree when $m = 3, n = 3$.
Worst case example str1="abc" str2="xyz"

We can see that many subproblems are solved again and again, for example $eD(2,2)$ is called three times. Since same subproblems are called again, this problem has Overlapping Subproblems property. So Edit Distance problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array that stores results of subproblems.

C++

```

// A Dynamic Programming based C++ program to find minimum
// number operations to convert str1 to str2
#include<bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers

```

```

int min(int x, int y, int z)
{
    return min(min(x, y), z);
}

int editDistDP(string str1, string str2, int m, int n)
{
    // Create a table to store results of subproblems
    int dp[m+1][n+1];

    // Fill d[][]
    for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            // If first string is empty, only option is to
            // insert all characters of second string
            if (i==0)
                dp[i][j] = j; // Min. operations = j

            // If second string is empty, only option is to
            // remove all characters of second string
            else if (j==0)
                dp[i][j] = i; // Min. operations = i

            // If last characters are same, ignore last char
            // and recur for remaining string
            else if (str1[i-1] == str2[j-1])
                dp[i][j] = dp[i-1][j-1];

            // If last character are different, consider all
            // possibilities and find minimum
            else
                dp[i][j] = 1 + min(dp[i][j-1], // Insert
                                   dp[i-1][j], // Remove
                                   dp[i-1][j-1]); // Replace
        }
    }

    return dp[m][n];
}

// Driver program
int main()
{
    // your code goes here
    string str1 = "sunday";
    string str2 = "saturday";

    cout << editDistDP(str1, str2, str1.length(), str2.length());

    return 0;
}

```

Java

```

// A Dynamic Programming based Java program to find minimum
// number operations to convert str1 to str2
class EDIST
{
    static int min(int x,int y,int z)
    {
        if (x < y && x < z) return x;
        if (y < x && y < z) return y;
        else return z;
    }

    static int editDistDP(String str1, String str2, int m, int n)
    {
        // Create a table to store results of subproblems
        int dp[][] = new int[m+1][n+1];

        // Fill d[][]
        for (int i=0; i<=m; i++)
        {
            for (int j=0; j<=n; j++)
            {
                // If first string is empty, only option is to
                // insert all characters of second string
                if (i==0)
                    dp[i][j] = j; // Min. operations = j

                // If second string is empty, only option is to
                // remove all characters of second string
                else if (j==0)
                    dp[i][j] = i; // Min. operations = i

                // If last characters are same, ignore last char
                // and recur for remaining string
                else if (str1.charAt(i-1) == str2.charAt(j-1))
                    dp[i][j] = dp[i-1][j-1];

                // If last character are different, consider all
                // possibilities and find minimum
                else
                    dp[i][j] = 1 + min(dp[i][j-1], // Insert
                                       dp[i-1][j], // Remove
                                       dp[i-1][j-1]); // Replace
            }
        }

        return dp[m][n];
    }
}

```

```

for (int i=0; i<=m; i++)
{
    for (int j=0; j<=n; j++)
    {
        // If first string is empty, only option is to
        // insert all characters of second string
        if (i==0)
            dp[i][j] = j; // Min. operations = j

        // If second string is empty, only option is to
        // remove all characters of second string
        else if (j==0)
            dp[i][j] = i; // Min. operations = i

        // If last characters are same, ignore last char
        // and recur for remaining string
        else if (str1.charAt(i-1) == str2.charAt(j-1))
            dp[i][j] = dp[i-1][j-1];

        // If last character are different, consider all
        // possibilities and find minimum
        else
            dp[i][j] = 1 + min(dp[i][j-1], // Insert
                               dp[i-1][j], // Remove
                               dp[i-1][j-1]); // Replace
    }
}

return dp[m][n];
}

public static void main(String args[])
{
    String str1 = "sunday";
    String str2 = "saturday";
    System.out.println( editDistDP( str1 , str2 , str1.length() , str2.length() ) );
}
/*This code is contributed by Rajat Mishra*/

```

Python

```

dp[i-1][j-1]) # Replace

return dp[m][n]

# Driver program
str1 = "sunday"
str2 = "saturday"

print(editDistDP(str1, str2, len(str1), len(str2)))
# This code is contributed by Bhavya Jain

```

Output:

3

Time Complexity: $O(m \times n)$

Auxiliary Space: $O(m \times n)$

Applications: There are many practical applications of edit distance algorithm, refer [Lucene API](#) for sample. Another example, display all the words in a dictionary that are near proximity to a given word/incorrectly spelled word.

Thanks to Vivek Kumar for suggesting above updates.

Thanks to [Venki](#) for providing initial post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

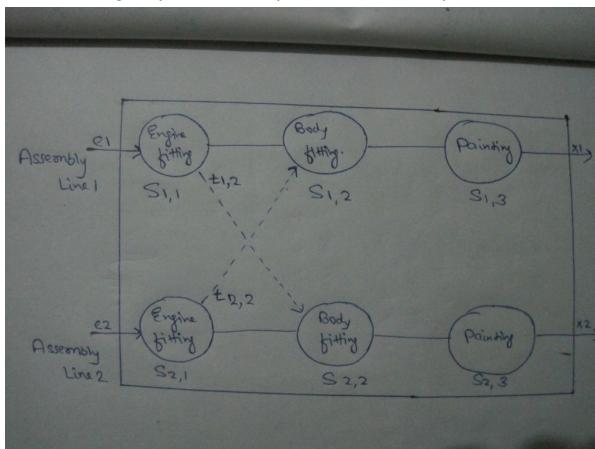
GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming
[About Venki](#)
Software Engineer
[View all posts by Venki →](#)

Dynamic Programming | Set 34 (Assembly Line Scheduling)

A car factory has two assembly lines, each with n stations. A station is denoted by $S_{i,j}$ where i is either 1 or 2 and indicates the assembly line the station is on, and j indicates the number of the station. The time taken per station is denoted by $a_{i,j}$. Each station is dedicated to some sort of work like engine fitting, body fitting, painting and so on. So, a car chassis must pass through each of the n stations in order before exiting the factory. The parallel stations of the two assembly lines perform the same task. After it passes through station $S_{i,j}$, it will continue to station $S_{i,j+1}$ unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line i at station $j - 1$ to station j on the other line takes time $t_{i,j}$. Each assembly line takes an entry time e_i and exit time x_i which may be different for the two lines. Give an algorithm for computing the minimum time it will take to build a car chassis.

The below figure presents the problem in a clear picture:



The following information can be extracted from the problem statement to make it simpler:

- Two assembly lines, 1 and 2, each with stations from 1 to n .
- A car chassis must pass through all stations from 1 to n in order (in any of the two assembly lines). i.e. it cannot jump from station i to station j if they are not at one move distance.
- The car chassis can move one station forward in the same line, or one station diagonally in the other line. It incurs an extra cost $t_{i,j}$ to move to station j from line i . No cost is incurred for movement in same line.
- The time taken in station j on line i is $a_{i,j}$.
- $S_{i,j}$ represents a station j on line i .

Breaking the problem into smaller sub-problems:

We can easily find the i th factorial if $(i-1)$ th factorial is known. Can we apply the similar funda here?

If the minimum time taken by the chassis to leave station $S_{i,j-1}$ is known, the minimum time taken to leave station $S_{i,j}$ can be calculated quickly by combining $a_{i,j}$ and $t_{i,j}$.

T1(j) indicates the minimum time taken by the car chassis to leave station j on assembly line 1.

T2(j) indicates the minimum time taken by the car chassis to leave station j on assembly line 2.

Base cases:

The entry time e_i comes into picture only when the car chassis enters the car factory.

Time taken to leave first station in line 1 is given by:

$$T1(1) = \text{Entry time in Line 1} + \text{Time spent in station } S_{1,1}$$

$$T1(1) = e_1 + a_{1,1}$$

Similarly, time taken to leave first station in line 2 is given by:

$$T2(1) = e_2 + a_{2,1}$$

Recursive Relations:

If we look at the problem statement, it quickly boils down to the below observations:

The car chassis at station $S_{1,j}$ can come either from station $S_{1,j-1}$ or station $S_{2,j-1}$.

Case #1: Its previous station is $S_{1,j-1}$

The minimum time to leave station $S_{1,j}$ is given by:

$$T1(j) = \text{Minimum time taken to leave station } S_{1,j-1} + \text{Time spent in station } S_{1,j}$$

$$T1(j) = T1(j-1) + a_{1,j}$$

Case #2: Its previous station is $S_{2,j-1}$

The minimum time to leave station $S_{1,j}$ is given by:

$$T1(j) = \text{Minimum time taken to leave station } S_{2,j-1} + \text{Extra cost incurred to change the assembly line} + \text{Time spent in station } S_{1,j}$$

$$T1(j) = T2(j-1) + t_{2,j} + a_{1,j}$$

The minimum time $T1(j)$ is given by the minimum of the two obtained in cases #1 and #2.

$$T1(j) = \min((T1(j-1) + a_{1,j}), (T2(j-1) + t_{2,j} + a_{1,j}))$$

Similarly the minimum time to reach station $S_{2,j}$ is given by:

$$T2(j) = \min((T2(j-1) + a_{2,j}), (T1(j-1) + t_{1,j} + a_{2,j}))$$

The total minimum time taken by the car chassis to come out of the factory is given by:

$$T_{\min} = \min(\text{Time taken to leave station } S_{i,n} + \text{Time taken to exit the car factory})$$

$$T_{\min} = \min(T1(n) + x_1, T2(n) + x_2)$$

Why dynamic programming?

The above recursion exhibits overlapping sub-problems. There are two ways to reach station $S_{1,j}$:

1. From station $S_{1,j-1}$
2. From station $S_{2,j-1}$

So, to find the minimum time to leave station $S_{1,j}$ the minimum time to leave the previous two stations must be calculated(as explained in above recursion).

Similarly, there are two ways to reach station $S_{2,j}$:

1. From station $S_{2,j-1}$
2. From station $S_{1,j-1}$

Please note that the minimum times to leave stations $S_{1,j-1}$ and $S_{2,j-1}$ have already been calculated.

So, we need two tables to store the partial results calculated for each station in an assembly line. The table will be filled in bottom-up fashion.

Note:

In this post, the word "leave" has been used in place of "reach" to avoid the confusion. Since the car chassis must spend a fixed time in each station, the word leave suits better.

Implementation:

```
// A C program to find minimum possible time by the car chassis to complete
#include <stdio.h>
#define NUM_LINE 2
#define NUM_STATION 4

// Utility function to find minimum of two numbers
int min(int a, int b) { return a < b ? a : b; }
```

```

int carAssembly(int a[][NUM_STATION], int t[][NUM_STATION], int *e, int *x)
{
    int T1[NUM_STATION], T2[NUM_STATION], i;

    T1[0] = e[0] + a[0][0]; // time taken to leave first station in line 1
    T2[0] = e[1] + a[1][0]; // time taken to leave first station in line 2

    // Fill tables T1[] and T2[] using the above given recursive relations
    for (i = 1; i < NUM_STATION; ++i)
    {
        T1[i] = min(T1[i-1] + a[0][i], T2[i-1] + t[1][i] + a[0][i]);
        T2[i] = min(T2[i-1] + a[1][i], T1[i-1] + t[0][i] + a[1][i]);
    }

    // Consider exit times and return minimum
    return min(T1[NUM_STATION-1] + x[0], T2[NUM_STATION-1] + x[1]);
}

int main()
{
    int a[][NUM_STATION] = {{4, 5, 3, 2},
                           {2, 10, 1, 4}};
    int t[][NUM_STATION] = {{0, 7, 4, 5},
                           {0, 9, 2, 8}};
    int e[] = {10, 12}, x[] = {18, 7};

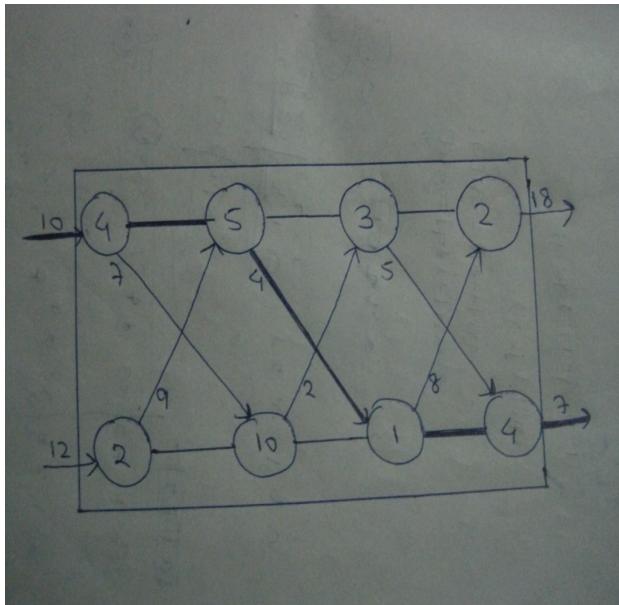
    printf("%d", carAssembly(a, t, e, x));

    return 0;
}

```

Output:

35



The bold line shows the path covered by the car chassis for given input values.

Exercise:

Extend the above algorithm to print the path covered by the car chassis in the factory.

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

This article is compiled by **Aashish Barnwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

March 9, 2017

Started with online test at Hackerearth. It had 2 questions. 1. Find subarray with max sum. Array have both -ve and +ve integers. 2. Consider a matrix with rows and columns, where each cell contains either a 0 or a 1 and any cell containing a 1 is called a filled cell. Two cells are... [Read More »](#)

Interview Experiences
Amazon

Amazon's most frequently asked interview questions | Set 2

[Amazon's Most Frequently Asked Questions | Set 1](#)

Level – Easy

1. Get minimum element from stack – Practice [here](#)
2. Serialize and deserialize a binary tree – Practice [here](#)
3. Print a binary tree in a vertical order – Practice [here](#)
4. Celebrity problem – Practice [here](#)
5. Level order traversal
6. Swap the kth element from starting and from the end position – Practice [here](#)
7. Binary tree to bst – Practice [here](#)
8. Max sum in the configuration – Practice [here](#)
9. Find the nth element of spiral matrix – Practice [here](#)
10. Count the number of occurrences in a sorted array
11. Find the smallest window in a string containing all characters of another string
12. Find the maximum of all subarrays of size k
13. Multiply two numbers represented as a linked list.
14. Find the kth smallest element in row wise and column wise sorted matrix
15. Minimum swaps required to arrange pairs
16. There is an array of N numbers ranging from 1 to N. Only 1 number is missing, return the index of that number
17. Find the second largest and second smallest in a given array in single traversal.
18. Find power(x,y) without using pow function.(divide and conquer approach required)
19. Count possible decoding sequence

Level – Medium

1. Given two string print them inter leaving strings characters
2. Minimum cost required to travel from top left to the bottom right in a matrix
3. Maximum difference between node and its ancestors – Practice [here](#)
4. Min distance between two given nodes of a binary tree – Practice [here](#)
5. Find the number of island – Practice [here](#)
6. Topological Sort – Practice [here](#)
7. Detect cycle in a directed graph – Practice [here](#)
8. Flattening a link list – Practice [here](#)
9. Detect a loop in a linked list – Practice [here](#)
10. Check if a binary tree is BST or not
11. Min Cost path
12. Count ways to reach nth stair
13. Maximum Subarray Problem
14. Palindrome Partitioning
15. Given a binary tree find the minimum root to leaf height.
16. Implement LRU cache

Level – Hard

1. Boolean parenthesis – Practice [here](#)
2. Maximum Index – Practice [here](#)
3. Largest Number formed in the array – Practice [here](#)
4. Find the length of maximum numbers of consecutive numbers jumped up in an array
5. Delete the elements in a linklist whose sum is equal to zero
6. Given a list of numbers of odd length design an algorithm to remove a number and divide the rest numbers equally so as it makes their sum same

7. Find diameter of a binary tree

Also see

- [Amazon Interview Experiences](#)
- [Amazon Practice Questions](#)
- [Top topics for Interview Preparation](#)

If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Articles Tags: [Amazon](#), [Interview Tips](#), [Placement preparation](#)

Last Minute Notes – GATE 2017

Top 5 Topics for Each Section of GATE CS Syllabus

GATE CS MOCK 2017

Previous year papers GATE CS, solutions and explanations year-wise and topic-wise.

This page contains GATE CS Preparation Notes / Tutorials on Mathematics, Digital Logic, Computer Organization and Architecture, Programming and Data Structures, Algorithms, Theory of Computation, Compiler Design, Operating Systems, DBMS (Database Management Systems), and Computer Networks listed according to the GATE CS 2017 official Syllabus.

GATE 2017 SYLLABUS	LEARN	PRACTICE	EXTERNAL RESOURCES
Section1: Engineering Mathematics <u>Discrete Mathematics:</u> Propositional and first order logic. Sets, relations, functions, partial orders and lattices. Groups. <u>Graphs:</u> connectivity, matching, coloring. <u>Combinatorics:</u> counting, recurrence relations, generating functions. <u>Linear Algebra:</u> Matrices, determinants, system of linear equations, eigenvalues and eigenvectors, LU decomposition. <u>Calculus:</u> Limits, continuity and differentiability. Maxima and minima. Mean value theorem. Integration. <u>Probability:</u> Random variables. Uniform, normal, exponential, poisson and binomial distributions. Mean, median, mode and standard deviation. Conditional probability and Bayes	Discrete Mathematics: Proposition Logic Introduction Propositional Equivalence Predicates and Quantifiers Set Theory Introduction Set Theory Set Operations Relations and their types Relations and their representations Groups (New) Combinatorics: Recurrence relations Pigeonhole Principle Linear Algebra: Matrix Introduction Eigen Values and Eigen Vectors L U Decomposition Calculus: Lagrange's Mean Value Theorem Mean Value Theorem Rolle's Theorem Probability: Random Variables Mean Variance And Standard Deviation Conditional Probability Bayes's formula for Conditional	Set Theory & Algebra Linear Algebra Numerical Methods and Calculus Graph Theory Combinatorics Propositional and First Order Logic	Set theory,Algebra,Mathematical Logic Discrete Mathematics-NPTEL Video Lectures Book PDF-Schaum's Graph Theory Video Lectures-IISC Bangalore Lecture Notes-MIT

theorem.	Probability		
Section 2: Digital Logic Boolean algebra. Combinational and sequential circuits. Minimization. Number representations and computer arithmetic (fixed and floating point).	Flip-flop types and their Conversion Half Adder Half Subtractor K-Map Counters Synchronous Sequential Circuits (New) Number System and base conversions (New) Floating Point Representation (New) Last Minute Notes	Digital Logic & Number representation(28)	Video Lectures-NPTEL Notes-Number System-Swarthmore
Section 3: Computer Organization and Architecture Machine instructions and addressing modes. ALU, data-path and control unit. Instruction pipelining. <u>Memory hierarchy</u> : cache, main memory and secondary storage; I/O interface (interrupt and DMA mode).	Machine Instructions Addressing Modes Computer Arithmetic Set – 1 Computer Arithmetic Set – 2 Pipelining Set 1 (Execution, Stages and Throughput) Pipelining Set 2 (Dependencies and Data Hazard) Pipelining Set 3 (Types and Stalling) Memory hierarchy: Cache Memory Cache Organization Introduction I/O Interface (Interrupt and DMA Mode)	Computer Organization and Architecture(33)	Book PDF- Carl Hamacher Book PDF-Morris Mano
Section 4: Programming and Data Structures Programming in C. Recursion. Arrays, stacks, queues, linked lists, trees, binary search trees, binary heaps, graphs.	C Programming Recursion Recursive functions Tail Recursion Arrays Stack Queue Binary Trees Tree Traversals Binary Search Trees Balanced Binary Search Trees Array Heap Graph Graph Traversals B and B+ Trees Misc Data Structures and Algorithm	C Language Recursion Linked List Stack Queue Binary Trees Tree Traversals Binary Search Trees Balanced Binary Search Trees Array Heap Graph Graph Traversals B and B+ Trees Misc Data Structures and Algorithm	Video lectures-IITD Book- Introduction to Algorithms by Cormen, Thomas H.
Section 5: Algorithms Searching, sorting, hashing. Asymptotic worst case time and space complexity. <u>Algorithm design techniques</u> : greedy, dynamic programming and divide-and-conquer. Graph search, minimum spanning trees, shortest paths.	Searching and Sorting Hashing Analysis of Algorithms Greedy Algorithms Dynamic Programming Divide and Conquer Graph Algorithms (Search Algorithms) Minimum Spanning Tree: Prims Minimum Spanning tree Prims Minimum Spanning Tree for adjacency list representation Kruskals Minimum Spanning tree Shortest Paths: Dijkstras Shortest Path Algorithm Dijkstra's Algorithm for Adjacency List Representation Bellman–Ford Algorithm Floyd Warshall Algorithm Shortest Path in Directed Acyclic Graph Shortest path with exactly k edges in a directed and weighted graph	Searching Sorting Hash Analysis of Algorithms Analysis of Algorithms (Recurrences) Divide and Conquer Greedy Algorithms Dynamic Programming Backtracking Graph Shortest Paths Graph Minimum Spanning Tree NP Complete Misc Data Structures and Algorithm	Video lectures-IITD Book- Introduction to Algorithms by Cormen, Thomas H.
Section 6: Theory of Computation Regular expressions and finite automata.	Finite Automata Introduction Chomsky Hierarchy Pumping Lemma	Regular languages and finite automata Context free languages	Web Resource-ArsDigita University Sample Problems and Solutions-Loyola Univ

Context-free grammars and push-down automata. Regular and context-free languages, pumping lemma. Turing machines and undecidability.	Designing Finite Automata from Regular Expression Regular Expressions, Regular Grammar and Regular Languages Pushdown Automata Closure Properties of Context Free Languages Conversion from NFA to DFA Minimization of DFA Mealy and Moore Machines Decidability Turing Machine Ambiguity in CFG and CFL Simplifying Context Free Grammars Recursive and Recursive Enumerable Languages Undecidability and Reducibility Last Minute Notes	and Push-down automata Recursively enumerable sets and Turing machines Undecidability Automata Theory	
Section 7: Compiler Design Lexical analysis, parsing, syntax-directed translation. Runtime environments. Intermediate code generation.	Lexical Analysis Introduction to Syntax Analysis Parsing Set 1 Parsing Set 2 Parsing Set 3 Syntax Directed Translation Intermediate Code Generation Runtime Environments Classification of Context Free Grammars Ambiguous Grammar Why FIRST and FOLLOW? FIRST Set in Syntax Analysis FOLLOW Set in Syntax Analysis	Lexical analysis Parsing and Syntax directed translation Code Generation and Optimization Compiler Design	Lecture Slides-Stanford Book-Aho and Ullman Dragon Book Lecture Notes Video Lectures-Stanford
Section 8: Operating System Processes, threads, inter-process communication, concurrency and synchronization. Deadlock. CPU scheduling. Memory management and virtual memory. File systems.	Process Management Introduction Process Scheduling Process scheduler Disk Scheduling Process Synchronization Introduction Process Synchronization Monitors Deadlock Introduction Deadlock Prevention And Avoidance Deadlock Detection And Recovery Memory Management Partition Allocation Method Page Replacement Algorithm User Thread VS Kernel Thread Multi threading Model Inter-Process Communication Fork System Call Paging Segmentation Banker's Algorithm Readers-Writers Problem Set 1 (Introduction and Readers Preference Solution) Difference between Priority Inversion and Priority Inheritance File Systems Virtual Memory Operating System Notes Last Minute Notes Commonly Asked Interview Question	Process Management CPU Scheduling Memory Management Input Output Systems Operating Systems	Web resource- VirginiaTech Univ. Lecture Slides- Silberschatz, Galvin, Gagne Video Lectures-IIT KGP Practice Problems and Solutions- William Stallings
Section 9: Databases ER-model. <u>Relational model:</u> relational algebra,	Need for DBMS Relational Model and Algebra : Relational Model Relational Model Introduction and Codd Rules	ER and Relational Models Database Design (Normal Forms) SQL	Lecture Slides-Silberschatz, Korth and Sudarshan Lecture Slides-Raghu Ramakrishnan and Johannes

<p>tuple calculus, SQL.</p> <p>Integrity constraints, normal forms.</p> <p>File organization, indexing (e.g., B and B+ trees).</p> <p>Transactions and concurrency control.</p>	<p>Keys in Relational Model (Candidate, Super, Primary, Alternate and Foreign) Relational Algebra-Overview Relational Algebra-Basic Operators Relational Algebra-Extended Operators Normalization: Lossless Decomposition Dependency Preserving Decomposition Attribute Closure/Candidate Key- Functional Dependencies Database Normalization Introduction Database Normal Form Equivalence of Functional Dependencies Find the highest normal form of a relation ER Model : ER Model Minimization of ER Diagram Mapping from ER Model to Relational Model SQL: SQL Inner VS Outer Join Having Vs Where Clause Nested Queries in SQL Concurrency: ACID Properties Concurrency Control View equal schedule Conflict Serializability Recoverability of Schedules Misc: Indexing in Databases Set 1 How to solve Relational Algebra problems for GATE</p>	<p>Transactions and concurrency control Sequential files, indexing, B & B+ trees Database Management Systems</p>	<p>Gehrke Lecture Slides-Stanford DBMS course Video Lectures-IIT KGP</p>
<p>Section 10: Computer Networks</p> <p>Concept of layering.</p> <p>LAN technologies (Ethernet).</p> <p>Flow and error control techniques, switching.</p> <p>IPv4/IPv6, routers and routing algorithms (distance vector, link state).</p> <p>TCP/UDP and sockets, congestion control.</p> <p>Application layer protocols (DNS, SMTP, POP, FTP, HTTP).</p> <p>Basics of Wi-Fi.</p> <p><u>Network security</u>: authentication, basics of public key and private key cryptography, digital signatures and certificates, firewalls.</p>	<p>Network Topologies (New) LAN Technologies Network Devices IP Addressing Introduction and Classful Addressing IP Addressing Classless Addressing Network Layer Introduction Network Layer IPv4 Datagram Fragmentation and Delay Longest Prefix Matching in Routers Why DNS uses UDP not TCP Error Detection Congestion Control Leaky Bucket Algorithm (New) Stop and Wait ARQ Sliding Window Protocol Set 1 (Sender Side) Sliding Window Protocol Set 2 (Receiver Side) Difference between http and https DNS SMTP ICMP (New) Circuit Switching VS Packet Switching Basics of Wi-Fi Digital Signatures and Certificates Commonly asked Computer Networks Interview Questions Set 1</p>	<p>Data Link Layer Network Layer Transport Layer Misc Topics in Computer Networks Application Layer Network Security Computer Networks</p>	<p>Video Lectures by University of Washington. Lecture Notes-Prof. Dheeraj Sanghi, IIT Kanpur Web Resources on Computer Networks by Andrew S. Tanenbaum.</p>

Previous year papers GATE CS, solutions and explanations year-wise and topic-wise.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above!

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.

All about GATE CS Preparation for 2017 aspirants. The page contains solutions of previous year GATE CS papers with explanations, topic wise Quizzes, notes/tutorials and important links for preparation.

GATE CS MOCK 2017

GATE CS Notes/Tutorials (According to Official GATE 2017 Syllabus)

Previous Years' questions/answers/explanation for GATE CS

- [GATE-CS-2017 \(Set 1\)](#)
- [GATE-CS-2017 \(Set 2\)](#)
- [GATE-CS-2016 \(Set 1\)](#)
- [GATE-CS-2016 \(Set 2\)](#)
- [GATE-CS-2015 \(Set 1\)](#)
- [GATE-CS-2015 \(Set 2\)](#)
- [GATE-CS-2015 \(Set 3\)](#)
- [GATE-CS-2014-\(Set-1\)](#)
- [GATE-CS-2014-\(Set-2\)](#)
- [GATE-CS-2014-\(Set-3\)](#)
- [GATE CS 2013](#)
- [GATE CS 2012](#)
- [GATE CS 2011](#)
- [GATE CS 2010](#)
- [GATE-CS-2009](#)
- [GATE CS 2008](#)
- [GATE-CS-2007](#)
- [GATE-CS-2006](#)
- [GATE-CS-2005](#)
- [GATE-CS-2004](#)
- [GATE-CS-2003](#)
- [GATE-CS-2002](#)
- [GATE-CS-2001](#)
- [GATE-CS-2000](#)

Previous Years' questions/answers/explanation for GATE IT

- [GATE-IT-2004](#)
- [GATE-IT-2006](#)
- [GATE-IT-2008](#)
- [GATE-IT-2005](#)
- [GATE-IT-2007](#)

Topic-wise Mock Quizzes for GATE CS

Data Structures and Algorithms

- [Linked List](#)
- [Stack](#)
- [Queue](#)
- [Binary Trees](#)
- [Binary Search Trees](#)
- [Balanced Binary Search Trees](#)
- [Graph](#)
- [Hash](#)
- [Array](#)
- [Misc](#)
- [B and B+ Trees](#)
- [Heap](#)
- [Tree Traversals](#)
- [Analysis of Algorithms](#)
- [Sorting](#)
- [Divide and Conquer](#)
- [Greedy Algorithms](#)
- [Dynamic Programming](#)

DBMS

- [ER and Relational Models](#)
- [Database Design \(Normal Forms\)](#)
- [SQL](#)
- [Transactions and concurrency control](#)
- [Sequential files, indexing, B & B+ trees](#)
- [Database Management Systems](#)

Compiler Design

- [Lexical analysis](#)
- [Parsing and Syntax directed translation](#)
- [Code Generation and Optimization](#)
- [Compiler Design](#)

Computer Networks

- [Data Link Layer](#)
- [Network Layer](#)

Backtracking	Transport Layer
Misc	Misc Topics in Computer Networks
NP Complete	Application Layer
Searching	Network Security
Analysis of Algorithms (Recurrences)	Computer Networks
Recursion	
Bit Algorithms	Theory of Computation
Graph Traversals	Regular languages and finite automata
Graph Shortest Paths	Context free languages and Push-down automata
Graph Minimum Spanning Tree	Recursively enumerable sets and Turing machines
Data Structures and Algorithm	Undecidability
C Language	Automata Theory
Operating Systems	
Operating System Notes	Aptitude
Process Management	Probability
CPU Scheduling	English
Memory Management	General Aptitude
Input Output Systems	
Operating Systems	Engineering Mathematics
Computer Organization and Architecture	Set Theory & Algebra
Digital Logic & Number representation(28)	Linear Algebra
Computer Organization and Architecture(33)	Numerical Methods and Calculus
Sample GATE Mock Test	Graph Theory
	Combinatorics
	Propositional and First Order Logic

Important Links:

Solutions of GATE CS 2016 Mock Test	Last Minute Notes
Top 5 Topics for for Section of GATE CS Syllabus	GATE CS 2016 Official Papers
How to prepare in Last 10 days for GATE?	GATE CS 2017 Dates
GATE CS Topic wise External Reference Links	GATE CS 2017 Syllabus
Previous Year GATE Official Question Papers	

Please write comments if you find anything incorrect or wish to share more information for GATE CS preparation.

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

[Load Comments](#)

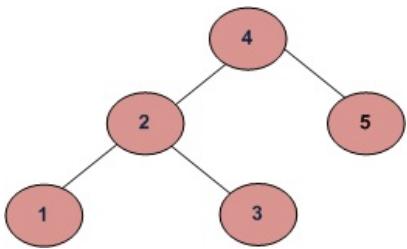
A program to check if a binary tree is BST or not

A binary search tree (BST) is a node based binary tree data structure which has the following properties.

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

From the above properties it naturally follows that:

- Each node (item in the tree) has a distinct key.



METHOD 1 (Simple but Wrong)

Following is a simple program. For each node, check if left node of it is smaller than the node and right node of it is greater than the node.

```

int isBST(struct node* node)
{
    if (node == NULL)
        return 1;

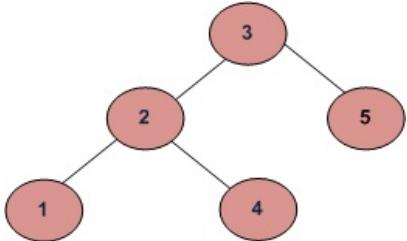
    /* false if left is > than node */
    if (node->left != NULL && node->left->data > node->data)
        return 0;

    /* false if right is < than node */
    if (node->right != NULL && node->right->data < node->data)
        return 0;

    /* false if, recursively, the left or right is not a BST */
    if (!isBST(node->left) || !isBST(node->right))
        return 0;

    /* passing all that, it's a BST */
    return 1;
}
  
```

This approach is wrong as this will return true for below binary tree (and below tree is not a BST because 4 is in left subtree of 3)



METHOD 2 (Correct but not efficient)

For each node, check if max value in left subtree is smaller than the node and min value in right subtree greater than the node.

```

/* Returns true if a binary tree is a binary search tree */
int isBST(struct node* node)
{
    if (node == NULL)
        return(true);

    /* false if the max of the left is > than us */
    if (node->left!=NULL && maxValue(node->left) > node->data)
        return(false);

    /* false if the min of the right is <= than us */
    if (node->right!=NULL && minValue(node->right) < node->data)
        return(false);

    /* false if, recursively, the left or right is not a BST */
    if (!isBST(node->left) || !isBST(node->right))
        return(false);

    /* passing all that, it's a BST */
    return(true);
}
  
```

It is assumed that you have helper functions `minValue()` and `maxValue()` that return the min or max int value from a non-empty tree

METHOD 3 (Correct and Efficient)

Method 2 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function `isBSTUtil(struct node* node, int min, int max)` that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be `INT_MIN` and `INT_MAX` — they narrow from there.

```
/* Returns true if the given tree is a binary search tree
 * (efficient version). */
int isBST(struct node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
 * values are >= min and
 * Implementation:
```

C

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

int isBSTUtil(struct node* node, int min, int max);

/* Returns true if the given tree is a binary search tree
 * (efficient version). */
int isBST(struct node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
 * values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)
{
    /* an empty tree is BST */
    if (node==NULL)
        return 1;

    /* false if this node violates the min/max constraint */
    if (node->data < min || node->data > max)
        return 0;

    /* otherwise check the subtrees recursively,
       tightening the min or max constraint */
    return
        isBSTUtil(node->left, min, node->data-1) && // Allow only distinct values
        isBSTUtil(node->right, node->data+1, max); // Allow only distinct values
}

/* Helper function that allocates a new node with the
 * given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
```

```

        malloc(sizeof(struct node));
node->data = data;
node->left = NULL;
node->right = NULL;

return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(4);
    root->left     = newNode(2);
    root->right    = newNode(5);
    root->left->left = newNode(1);
    root->left->right = newNode(3);

    if(isBST(root))
        printf("Is BST");
    else
        printf("Not a BST");

    getchar();
    return 0;
}

```

Java

```

//Java implementation to check if given Binary tree
//is a BST or not

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

public class BinaryTree
{
    //Root of the Binary Tree
    Node root;

    /* can give min and max value according to your code or
    can write a function to find min and max value of tree. */

    /* returns true if given search tree is binary
    search tree (efficient version) */
    boolean isBST() {
        return isBSTUtil(root, Integer.MIN_VALUE,
                        Integer.MAX_VALUE);
    }

    /* Returns true if the given tree is a BST and its
    values are >= min and <= max. */
    boolean isBSTUtil(Node node, int min, int max)
    {
        /* an empty tree is BST */
        if (node == null)
            return true;

```

```

/* false if this node violates the min/max constraints */
if (node.data < min || node.data > max)
    return false;

/* otherwise check the subtrees recursively
tightening the min/max constraints */
// Allow only distinct values
return (isBSTUtil(node.left, min, node.data-1) &&
        isBSTUtil(node.right, node.data+1, max));
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(4);
    tree.root.left = new Node(2);
    tree.root.right = new Node(5);
    tree.root.left.left = new Node(1);
    tree.root.left.right = new Node(3);

    if (tree.isBST())
        System.out.println("IS BST");
    else
        System.out.println("Not a BST");
}
}

```

Python

```

# Python program to check if a binary tree is bst or not

INT_MAX = 4294967296
INT_MIN = -4294967296

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


    # Returns true if the given tree is a binary search tree
    # (efficient version)
    def isBST(node):
        return (isBSTUtil(node, INT_MIN, INT_MAX))

    # Returns true if the given tree is a BST and its values
    # >= min and <= max
    def isBSTUtil(node, mini, maxi):

        # An empty tree is BST
        if node is None:
            return True

        # False if this node violates min/max constraint
        if node.data < mini or node.data > maxi:
            return False

        # Otherwise check the subtrees recursively
        # tightening the min or max constraint
        return (isBSTUtil(node.left, mini, node.data -1) and
                isBSTUtil(node.right, node.data+1, maxi))

```

```

# Driver program to test above function
root = Node(4)
root.left = Node(2)
root.right = Node(5)
root.left.left = Node(1)
root.left.right = Node(3)

if (isBST(root)):
    print "Is BST"
else:
    print "Not a BST"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Time Complexity: O(n)

Auxiliary Space : O(1) if Function Call Stack size is not considered, otherwise O(n)

METHOD 4(Using In-Order Traversal)

Thanks to [LJW489](#) for suggesting this method.

- 1) Do In-Order Traversal of the given tree and store the result in a temp array.
- 3) Check if the temp array is sorted in ascending order, if it is, then the tree is BST.

Time Complexity: O(n)

We can avoid the use of Auxiliary Array. While doing In-Order traversal, we can keep track of previously visited node. If the value of the currently visited node is less than or equal to the previous node, then the tree is not BST.

C

```

bool isBST(struct node* root)
{
    static struct node *prev = NULL;

    // traverse the tree in inorder fashion and keep track of prev node
    if (root)
    {
        if (!isBST(root->left))
            return false;

        // Allows only distinct valued nodes
        if (prev != NULL && root->data <= prev->data)
            return false;

        prev = root;

        return isBST(root->right);
    }

    return true;
}

```

Java

```

// Java implementation to check if given Binary tree
// is a BST or not

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

public class BinaryTree
{
    // Root of the Binary Tree
    Node root;

    // To keep tract of previous node in Inorder Traversal
    Node prev;

    boolean isBST() {
        prev = null;
        return isBST(root);
    }

    /* Returns true if given search tree is binary
       search tree (efficient version) */
    boolean isBST(Node node)
    {
        // traverse the tree in inorder fashion and
        // keep a track of previous node
        if (node != null)
        {
            if (!isBST(node.left))
                return false;

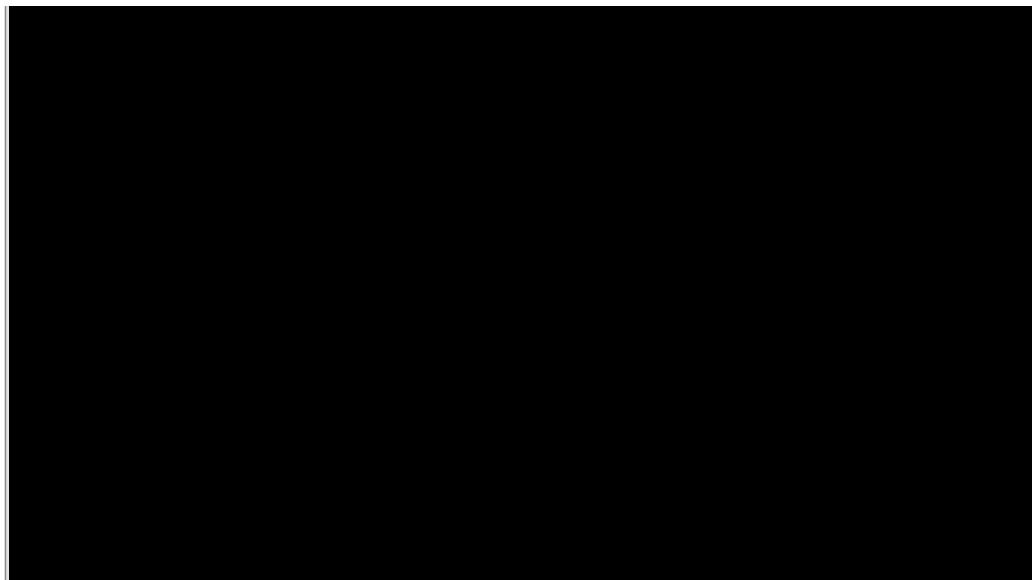
            // allows only distinct values node
            if (prev != null && node.data <= prev.data )
                return false;
            prev = node;
            return isBST(node.right);
        }
        return true;
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(4);
        tree.root.left = new Node(2);
        tree.root.right = new Node(5);
        tree.root.left.left = new Node(1);
        tree.root.left.right = new Node(3);

        if (tree.isBST())
            System.out.println("IS BST");
        else
            System.out.println("Not a BST");
    }
}

```

The use of static variable can also be avoided by using reference to prev node as a parameter (Similar to [this post](#)).



Sources:

http://en.wikipedia.org/wiki/Binary_search_tree

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

Please write comments if you find any bug in the above programs/algorithms or other ways to solve the same problem.

GATE CS Corner
Company Wise Coding Practice

Binary Search Tree

Remove all duplicates from a given string

Below are the different methods to remove duplicates in a string.

METHOD 1 (Use Sorting)

Algorithm:

- 1) Sort the elements.
- 2) Now in a loop, remove duplicates by comparing the current character with previous character.
- 3) Remove extra characters at the end of the resultant string.

Example:

```
Input string: geeksforgeeks
1) Sort the characters
   eeeefggkkorss
2) Remove duplicates
   efgkosgkkorss
3) Remove extra characters
   efgkos
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Note that, this method doesn't keep the original order of the input string. For example, if we are to remove duplicates for geeksforgeeks and keep the order of characters same, then output should be geksfor, but above function returns efgkos. We can modify this method by storing the original order. METHOD 2 keeps the order same.

Implementation:**C++**

```
// C++ program to remove duplicates, the order of
// characters is not maintained in this program
#include<bits/stdc++.h>
using namespace std;

/* Function to remove duplicates in a sorted array */
char *removeDupsSorted(char *str)
{
    int res_ind = 1, ip_ind = 1;

    /* In place removal of duplicate characters*/
    while (*(str + ip_ind))
    {
        if (*(str + ip_ind) != *(str + ip_ind - 1))
        {
            *(str + res_ind) = *(str + ip_ind);
            res_ind++;
        }
        ip_ind++;
    }

    /* After above step string is stringiitg.
       Removing extra iitg after string*/
    *(str + res_ind) = '\0';

    return str;
}

/* Function removes duplicate characters from the string
   This function work in-place and fills null characters
   in the extra space left */
char *removeDups(char *str)
{
    int n = strlen(str);

    // Sort the character array
    sort(str, str+n);

    // Remove duplicates from sorted
    return removeDupsSorted(str);
}

/* Driver program to test removeDups */
int main()
{
    char str[] = "geeksforgeeks";
    cout << removeDups(str);
    return 0;
}
```

C

```
// C++ program to remove duplicates, the order of
// characters is not maintained in this program
#include <stdio.h>
#include <stdlib.h>

/* Function to remove duplicates in a sorted array */
char *removeDupsSorted(char *str);

/* Utility function to sort array A[] */
void quickSort(char A[], int si, int ei);

/* Function removes duplicate characters from the string
   This function work in-place and fills null characters
   in the extra space left */
char *removeDups(char *str)
{
    int len = strlen(str);
    quickSort(str, 0, len-1);
    return removeDupsSorted(str);
}

/* Function to remove duplicates in a sorted array */
char *removeDupsSorted(char *str)
{
    int res_ind = 1, ip_ind = 1;

    /* In place removal of duplicate characters*/
    while (*(str + ip_ind))
    {
        if (*(str + ip_ind) != *(str + ip_ind - 1))
        {
            *(str + res_ind) = *(str + ip_ind);
            res_ind++;
        }
        ip_ind++;
    }

    /* After above step string is stringiitg.
       Removing extra iitg after string*/
    *(str + res_ind) = '\0';

    return str;
}

/* Driver program to test removeDups */
int main()
{
    char str[] = "geeksforgeeks";
    printf("%s", removeDups(str));
    getchar();
    return 0;
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SORTING
   PURPOSE */
void exchange(char *a, char *b)
{
    char temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int partition(char A[], int si, int ei)
{
    char x = A[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
```

```

{
    if (A[j] <= x)
    {
        i++;
        exchange(&A[i], &A[j]);
    }
}
exchange (&A[i + 1], &A[ei]);
return (i + 1);
}

/* Implementation of Quick Sort
A[] --> Array to be sorted
si --> Starting index
ei --> Ending index
*/
void quickSort(char A[], int si, int ei)
{
    int pi; /* Partitioning index */
    if (si < ei)
    {
        pi = partition(A, si, ei);
        quickSort(A, si, pi - 1);
        quickSort(A, pi + 1, ei);
    }
}

```

]

Python

```

# Python program to remove duplicates, the order of
# characters is not maintained in this program

# Utility function to convert string to list
def toMutable(string):
    temp = []
    for x in string:
        temp.append(x)
    return temp

# Utility function to convert string to list
def toString(List):
    return ''.join(List)

# Function to remove duplicates in a sorted array
def removeDupsSorted(List):
    res_ind = 1
    ip_ind = 1

    # In place removal of duplicate characters
    while ip_ind != len(List):
        if List[ip_ind] != List[ip_ind-1]:
            List[res_ind] = List[ip_ind]
            res_ind += 1
        ip_ind+=1

    # After above step string is stringiittg.
    # Removing extra iittg after string
    string = toString(List[0:res_ind])

    return string

# Function removes duplicate characters from the string
# This function work in-place and fills null characters
# in the extra space left
def removeDups(string):
    # Convert string to list
    List = toMutable(string)

    # Sort the character list
    List.sort()

```

```

# Remove duplicates from sorted
return removeDupsSorted(List)

# Driver program to test the above functions
string = "geeksforgeeks"
print removeDups(string)

# This code is contributed by Bhavya Jain

```

Output:

efgkors

Time Complexity: O(nlogn) If we use some nlogn sorting algorithm instead of quicksort.

METHOD 2 (Use Hashing)

Algorithm:

```

1: Initialize:
    str = "test string" /* input string */
    ip_ind = 0      /* index to keep track of location of next
                      character in input string */
    res_ind = 0      /* index to keep track of location of
                      next character in the resultant string */
    bin_hash[0..255] = {0,0,...} /* Binary hash to see if character is
                      already processed or not */

2: Do following for each character *(str + ip_ind) in input string:
    (a) if bin_hash is not set for *(str + ip_ind) then
        // if program sees the character *(str + ip_ind) first time
        (i) Set bin_hash for *(str + ip_ind)
        (ii) Move *(str + ip_ind) to the resultant string.
            This is done in-place.
        (iii) res_ind++
    (b) ip_ind++

/* String obtained after this step is "te sringng" */

3: Remove extra characters at the end of the resultant string.
/* String obtained after this step is "te sring" */

```

Implementation:

C

```

#include <stdio.h>
#include <stdlib.h>
#define NO_OF_CHARS 256
#define bool int

/* Function removes duplicate characters from the string
   This function work in-place and fills null characters
   in the extra space left */
char *removeDups(char *str)
{
    bool bin_hash[NO_OF_CHARS] = {0};
    int ip_ind = 0, res_ind = 0;
    char temp;

    /* In place removal of duplicate characters*/
    while (*(str + ip_ind))
    {
        temp = *(str + ip_ind);
        if (bin_hash[temp] == 0)
        {
            bin_hash[temp] = 1;
            *(str + res_ind) = *(str + ip_ind);
            res_ind++;
        }
        ip_ind++;
    }

    /* After above step string is stringiitg.
       Removing extra iittg after string*/
    *(str + res_ind) = '\0';
}

```

```

    return str;
}

/* Driver program to test removeDups */
int main()
{
    char str[] = "geeksforgeeks";
    printf("%s", removeDups(str));
    getchar();
    return 0;
}

```

Java

```

// Java prigram to remove duplicates
import java.util.*;

class RemoveDuplicates
{
    /* Function removes duplicate characters from the string
    This function work in-place */
    void removeDuplicates(String str)
    {
        LinkedHashSet<Character> lhs = new LinkedHashSet<>();
        for(int i=0;i<str.length();i++)
            lhs.add(str.charAt(i));

        // print string after deleting duplicate elements
        for(Character ch : lhs)
            System.out.print(ch);
    }

    /* Driver program to test removeDuplicates */
    public static void main(String args[])
    {
        String str = "geeksforgeeks";
        RemoveDuplicates r = new RemoveDuplicates();
        r.removeDuplicates(str);
    }
}

// This code has been contributed by Amit Khandelwal (Amit Khandelwal 1)

```

Python

```

# Python program to remvoe duplicate characters from an
# input string
NO_OF_CHARS = 256

# Since strings in Python are immutable and cannot be changed
# This utility function will convert the string to list
def toMutable(string):
    List = []
    for i in string:
        List.append(i)
    return List

# Utility function that changes list to string
def toString(List):
    return ''.join(List)

# Function removes duplicate characters from the string
# This function work in-place and fills null characters
# in the extra space left
def removeDups(string):
    bin_hash = [0] * NO_OF_CHARS
    ip_ind = 0
    res_ind = 0
    temp = ""
    mutableString = toMutable(string)

    for ip in range(0, len(mutableString)):
        if bin_hash[ord(mutableString[ip])] == 0:
            bin_hash[ord(mutableString[ip])] = 1
            res_ind = ip
            temp += mutableString[ip]
        else:
            if mutableString[ip] != mutableString[res_ind]:
                bin_hash[ord(mutableString[ip])] = 1
                res_ind = ip
                temp += mutableString[ip]
            else:
                bin_hash[ord(mutableString[ip])] = 1
                res_ind = ip
                temp += mutableString[ip]

```

```

# In place removal of duplicate characters
while ip_ind != len(mutableString):
    temp = mutableString[ip_ind]
    if bin_hash[ord(temp)] == 0:
        bin_hash[ord(temp)] = 1
        mutableString[res_ind] = mutableString[ip_ind]
        res_ind+=1
    ip_ind+=1

# After above step string is stringiitg.
# Removing extra iittg after string
return toString(mutableString[0:res_ind])

# Driver program to test the above functions
string = "geeksforgeeks"
print removeDups(string)

# A shorter version for this program is as follows
# import collections
# print ''.join(collections.OrderedDict.fromkeys(string))

# This code is contributed by Bhavya Jain

```

Output:

geksfor

Time Complexity: O(n)

Important Points:

- Method 1 doesn't maintain order of characters as original string, but method 2 does.
- It is assumed that number of possible characters in input string are 256. NO_OF_CHARS should be changed accordingly.
- calloc is used instead of malloc for memory allocations of counting array (count) to initialize allocated memory to '\0'. malloc() followed by memset() could also be used.
- Above algorithm also works for an integer array inputs if range of the integers in array is given. Example problem is to find maximum occurring number in an input array given that the input array contain integers only between 1000 to 1100

GATE CS Corner Company Wise Coding Practice

Strings

Search an element in a sorted and rotated array

An element in a sorted array can be found in $O(\log n)$ time via binary search. But suppose we rotate an ascending order sorted array at some pivot unknown to you beforehand. So for instance, 1 2 3 4 5 might become 3 4 5 1 2. Devise a way to find an element in the rotated array in $O(\log n)$ time.

3	4	5	1	2
---	---	---	---	---

Input : arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};

key = 3

Output : Found at index 8

Input : arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};

key = 30

Output : Not found

Input : arr[] = {30, 40, 50, 10, 20}

key = 10

Output : Found at index 3

We strongly recommend that you click here and practice it, before moving on to the solution.

All solutions provided here assume that all elements in array are distinct.

The idea is to find the pivot point, divide the array in two sub-arrays and call binary search.

The main idea for finding pivot is – for a sorted (in increasing order) and pivoted array, pivot element is the only element for which next element to it is smaller than it.

Using above criteria and binary search methodology we can get pivot element in O(logn) time

```
Input arr[] = {3, 4, 5, 1, 2}
Element to Search = 1
1) Find out pivot point and divide the array in two
   sub-arrays. (pivot = 2) /*Index of 5*/
2) Now call binary search for one of the two sub-arrays.
   (a) If element is greater than 0th element then
       search in left array
   (b) Else Search in right array
       (1 will go in else as 1 If element is found in selected sub-array then return index
Else return -1.
```

Implementation:

```
/* Program to search an element in a sorted and pivoted array*/
#include <stdio.h>

int findPivot(int[], int, int);
int binarySearch(int[], int, int, int);

/* Searches an element key in a pivoted sorted array arr[]
   of size n */
int pivotedBinarySearch(int arr[], int n, int key)
{
    int pivot = findPivot(arr, 0, n-1);

    // If we didn't find a pivot, then array is not rotated at all
    if (pivot == -1)
        return binarySearch(arr, 0, n-1, key);

    // If we found a pivot, then first compare with pivot and then
    // search in two subarrays around pivot
    if (arr[pivot] == key)
        return pivot;
    if (arr[0] <= key)
        return binarySearch(arr, 0, pivot-1, key);
    return binarySearch(arr, pivot+1, n-1, key);
}

/* Function to get pivot. For array 3, 4, 5, 6, 1, 2 it returns
   3 (index of 6) */
int findPivot(int arr[], int low, int high)
{
    // base cases
    if (high < low) return -1;
    if (high == low) return low;

    int mid = (low + high)/2; /*low + (high - low)/2;*/
    if (mid < high && arr[mid] > arr[mid + 1])
        return mid;
    if (mid > low && arr[mid] < arr[mid - 1])
        return (mid-1);
    if (arr[low] >= arr[mid])
        return findPivot(arr, low, mid-1);
    return findPivot(arr, mid + 1, high);
}

/* Standard Binary Search function*/
int binarySearch(int arr[], int low, int high, int key)
{
    if (high < low)
        return -1;
    int mid = (low + high)/2; /*low + (high - low)/2;*/
    if (key == arr[mid])
        return mid;
    if (key > arr[mid])
        return mid+1;
    if (key < arr[mid])
        return mid;
```

```

    return binarySearch(arr, (mid + 1), high, key);
    return binarySearch(arr, low, (mid - 1), key);
}

/* Driver program to check above functions */
int main()
{
    // Let us search 3 in below array
    int arr1[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};
    int n = sizeof(arr1)/sizeof(arr1[0]);
    int key = 3;
    printf("Index: %d\n", pivotedBinarySearch(arr1, n, key));
    return 0;
}

```

Output:

Index of the element is 8

Time Complexity O(logn). Thanks to Ajay Mishra for initial solution.

Improved Solution:

We can search an element in one pass of Binary Search. The idea is to search

- 1) Find middle point mid = (l + h)/2
- 2) If key is present at middle point, return mid.
- 3) Else If arr[l..mid] is sorted
 - a) If key to be searched lies in range from arr[l] to arr[mid], recur for arr[l..mid].
 - b) Else recur for arr[mid+1..r]
- 4) Else (arr[mid+1..r] must be sorted)
 - a) If key to be searched lies in range from arr[mid+1] to arr[r], recur for arr[mid+1..r].
 - b) Else recur for arr[l..mid]

Below is C++ implementation of above idea.

```

// Search an element in sorted and rotated array using
// single pass of Binary Search
#include<bits/stdc++.h>
using namespace std;

// Returns index of key in arr[l..h] if key is present,
// otherwise returns -1
int search(int arr[], int l, int h, int key)
{
    if (l > h) return -1;

    int mid = (l+h)/2;
    if (arr[mid] == key) return mid;

    /* If arr[l..mid] is sorted */
    if (arr[l] <= arr[mid])
    {
        /* As this subarray is sorted, we can quickly
         * check if key lies in half or other half */
        if (key >= arr[l] && key <= arr[mid])
            return search(arr, l, mid-1, key);

        return search(arr, mid+1, h, key);
    }

    /* If arr[l..mid] is not sorted, then arr[mid... r]
     * must be sorted*/
    if (key >= arr[mid] && key <= arr[h])
        return search(arr, mid+1, h, key);

    return search(arr, l, mid-1, key);
}

// Driver program
int main()
{
    int arr[] = {4, 5, 6, 7, 8, 9, 1, 2, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    int key = 6
}

```

```
int i = search(arr, 0, n-1, key);
if (i != -1) cout << "Index: " << i << endl;
else cout << "Key not found\n";
}
```

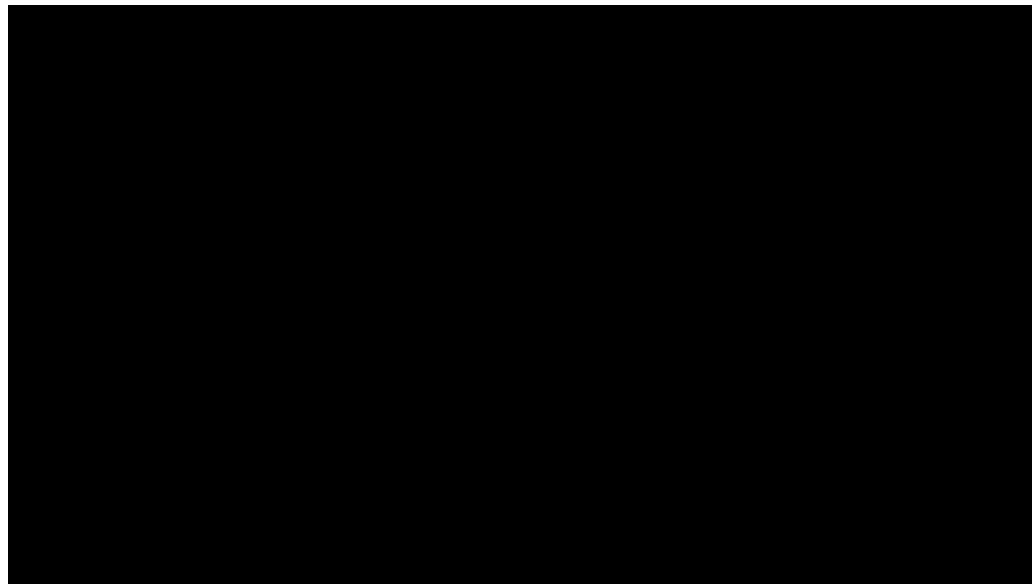
Output:

```
Index: 2
```

Thanks to [Gaurav Ahirwar](#) for suggesting above solution.

How to handle duplicates?

It doesn't look possible to search in $O(\log n)$ time in all cases when duplicates are allowed. For example consider searching 0 in {2, 2, 2, 2, 2, 2, 2, 2, 0, 2} and {2, 0, 2, 2, 2, 2, 2, 2, 2, 2}. It doesn't look possible to decide whether to recur for left half or right half by doing constant number of comparisons at the middle.



Similar Articles:

[Find the minimum element in a sorted and rotated array](#)

[Given a sorted and rotated array, find if there is a pair with a given sum.](#)

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Searching
array
Binary-Search
rotation

Add two numbers represented by linked lists | Set 2

Given two numbers represented by two linked lists, write a function that returns sum list. The sum list is linked list representation of addition of two input numbers. It is not allowed to modify the lists. Also, not allowed to use explicit extra space (Hint: Use Recursion).

Example

```
Input:
First List: 5->6->3 // represents number 563
Second List: 8->4->2 // represents number 842
Output
Resultant list: 1->4->0->5 // represents number 1405
```

We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

We have discussed a solution [here](#) which is for linked lists where least significant digit is first node of lists and most significant digit is last node. In this problem, most significant node is first node and least significant digit is last node and we are not allowed to modify the lists. Recursion is used here to calculate sum from right to left.

Following are the steps.

- 1) Calculate sizes of given two linked lists.
- 2) If sizes are same, then calculate sum using recursion. Hold all nodes in recursion call stack till the rightmost node, calculate sum of rightmost nodes and forward carry to left side.
- 3) If size is not same, then follow below steps:
 -a) Calculate difference of sizes of two linked lists. Let the difference be *diff*
 -b) Move *diff* nodes ahead in the bigger linked list. Now use step 2 to calculate sum of smaller list and right sub-list (of same size) of larger list. Also, store the carry of this sum.
 -c) Calculate sum of the carry (calculated in previous step) with the remaining left sub-list of larger list. Nodes of this sum are added at the beginning of sum list obtained previous step.

Following is C implementation of the above approach.

```
// A recursive program to add two linked lists

#include <stdio.h>
#include <stdlib.h>

// A linked List Node
struct node
{
    int data;
    struct node* next;
};

typedef struct node node;

/* A utility function to insert a node at the beginning of linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* A utility function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

// A utility function to swap two pointers
void swapPointer( node** a, node** b )
{
    node* t = *a;
    *a = *b;
    *b = t;
}

/* A utility function to get size of linked list */
int getSize(struct node *node)
{
    int size = 0;
    while (node != NULL)
    {
        node = node->next;
        size++;
    }
    return size;
}
```

```

// Adds two linked lists of same size represented by head1 and head2 and returns
// head of the resultant linked list. Carry is propagated while returning from
// the recursion
node* addSameSize(node* head1, node* head2, int* carry)
{
    // Since the function assumes linked lists are of same size,
    // check any of the two head pointers
    if (head1 == NULL)
        return NULL;

    int sum;

    // Allocate memory for sum node of current two nodes
    node* result = (node *)malloc(sizeof(node));

    // Recursively add remaining nodes and get the carry
    result->next = addSameSize(head1->next, head2->next, carry);

    // add digits of current nodes and propagated carry
    sum = head1->data + head2->data + *carry;
    *carry = sum / 10;
    sum = sum % 10;

    // Assign the sum to current node of resultant list
    result->data = sum;

    return result;
}

// This function is called after the smaller list is added to the bigger
// list's sublist of same size. Once the right sublist is added, the carry
// must be added to left side of larger list to get the final result.
void addCarryToRemaining(node* head1, node* cur, int* carry, node** result)
{
    int sum;

    // If diff. number of nodes are not traversed, add carry
    if (head1 != cur)
    {
        addCarryToRemaining(head1->next, cur, carry, result);

        sum = head1->data + *carry;
        *carry = sum/10;
        sum %= 10;

        // add this node to the front of the result
        push(result, sum);
    }
}

// The main function that adds two linked lists represented by head1 and head2.
// The sum of two lists is stored in a list referred by result
void addList(node* head1, node* head2, node** result)
{
    node *cur;

    // first list is empty
    if (head1 == NULL)
    {
        *result = head2;
        return;
    }

    // second list is empty
    else if (head2 == NULL)
    {
        *result = head1;
        return;
    }

    int size1 = getSize(head1);
    int size2 = getSize(head2) ;

    int carry = 0;
}

```

```

// Add same size lists
if (size1 == size2)
    *result = addSameSize(head1, head2, &carry);

else
{
    int diff = abs(size1 - size2);

    // First list should always be larger than second list.
    // If not, swap pointers
    if (size1 < size2)
        swapPointer(&head1, &head2);

    // move diff. number of nodes in first list
    for (cur = head1; diff--; cur = cur->next);

    // get addition of same size lists
    *result = addSameSize(cur, head2, &carry);

    // get addition of remaining first list and carry
    addCarryToRemaining(head1, cur, &carry, result);
}

// if some carry is still there, add a new node to the front of
// the result list. e.g. 999 and 87
if (carry)
    push(result, carry);
}

// Driver program to test above functions
int main()
{
    node *head1 = NULL, *head2 = NULL, *result = NULL;

    int arr1[] = {9, 9, 9};
    int arr2[] = {1, 8};

    int size1 = sizeof(arr1) / sizeof(arr1[0]);
    int size2 = sizeof(arr2) / sizeof(arr2[0]);

    // Create first list as 9->9->9
    int i;
    for (i = size1-1; i >= 0; --i)
        push(&head1, arr1[i]);

    // Create second list as 1->8
    for (i = size2-1; i >= 0; --i)
        push(&head2, arr2[i]);

    addList(head1, head2, &result);

    printList(result);

    return 0;
}

```

Output:

1 0 1 7

Time Complexity: O(m+n) where m and n are the sizes of given two linked lists.

Related Article : [Add two numbers represented by linked lists | Set 1](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Program to print last 10 lines

Given some text lines in one string, each line is separated by '\n' character. Print the last ten lines. If number of lines is less than 10, then print all lines.

Source: Microsoft Interview | Set 10

Following are the steps

- 1) Find the last occurrence of DELIM or '\n'
- 2) Initialize target position as last occurrence of '\n' and count as 0 , and do following while count 2.a) Find the next instance of '\n' and update target position
.....2.b) Skip '\n' and increment count of '\n' and update target position
- 3) Print the sub-string from target position.

```
/* Program to print the last 10 lines. If number of lines is less
than 10, then print all lines. */

#include <stdio.h>
#include <string.h>
#define DELIM '\n'

/* Function to print last n lines of a given string */
void print_last_lines(char *str, int n)
{
    /* Base case */
    if (n <= 0)
        return;

    size_t cnt = 0; // To store count of '\n' or DELIM
    char *target_pos = NULL; // To store the output position in str

    /* Step 1: Find the last occurrence of DELIM or '\n' */
    target_pos = strrchr(str, DELIM);

    /* Error if '\n' is not present at all */
    if (target_pos == NULL)
    {
        fprintf(stderr, "ERROR: string doesn't contain '\n' character\n");
        return;
    }

    /* Step 2: Find the target position from where we need to print the string */
    while (cnt < n)
    {
        // Step 2.a: Find the next instance of '\n'
        while (str < target_pos && *target_pos != DELIM)
            --target_pos;

        /* Step 2.b: skip '\n' and increment count of '\n' */
        if (*target_pos == DELIM)
            --target_pos, ++cnt;

        /* str < target_pos means str has less than 10 '\n' characters,
         so break from loop */
        else
            break;
    }

    /* In while loop, target_pos is decremented 2 times, that's why target_pos + 2 */
    if (str < target_pos)
        target_pos += 2;

    // Step 3: Print the string from target_pos
    printf("%s\n", target_pos);
}

// Driver program to test above function
int main(void)
{
    char *str1 = "str1\nstr2\nstr3\nstr4\nstr5\nstr6\nstr7\nstr8\nstr9"
                "\nstr10\nstr11\nstr12\nstr13\nstr14\nstr15\nstr16\nstr17"
                "\nstr18\nstr19\nstr20\nstr21\nstr22\nstr23\nstr24\nstr25";
    char *str2 = "str1\nstr2\nstr3\nstr4\nstr5\nstr6\nstr7";
    char *str3 = "\n";
    char *str4 = "";
}
```

```

print_last_lines(str1, 10);
printf("-----\n");

print_last_lines(str2, 10);
printf("-----\n");

print_last_lines(str3, 10);
printf("-----\n");

print_last_lines(str4, 10);
printf("-----\n");

return 0;
}

```

Output:

```

str16
str17
str18
str19
str20
str21
str22
str23
str24
str25
-----
str1
str2
str3
str4
str5
str6
str7
-----
-----
ERROR: string doesn't contain '\n' character
-----
```

This article is compiled by **Narendra Kangarkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

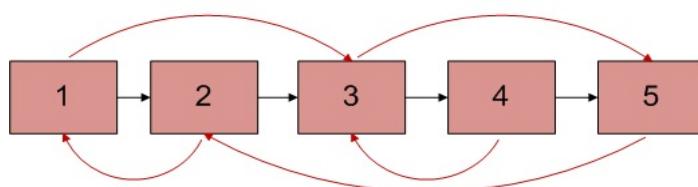
GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Clone a linked list with next and random pointer | Set 1

You are given a Double Link List with one pointer of each node pointing to the next node just like in a single link list. The second pointer however CAN point to any node in the list and not just the previous node. Now write a program in **O(n)** time to duplicate this list. That is, write a program which will create a copy of this list.

Let us call the second pointer as arbit pointer as it can point to any arbitrary node in the linked list.



Arbitrary pointers are shown in red and next pointers in black

Figure 1

Method 1 (Uses O(n) extra space)

This method stores the next and arbitrary mappings (of original list) in an array first, then modifies the original Linked List (to create copy).

creates a copy. And finally restores the original list.

- 1) Create all nodes in copy linked list using next pointers.
 - 3) Store the node and its next pointer mappings of original linked list.
 - 3) Change next pointer of all nodes in original linked list to point to the corresponding node in copy linked list.
- Following diagram shows status of both Linked Lists after above 3 steps. The red arrow shows arbit pointers and black arrow shows next pointers.

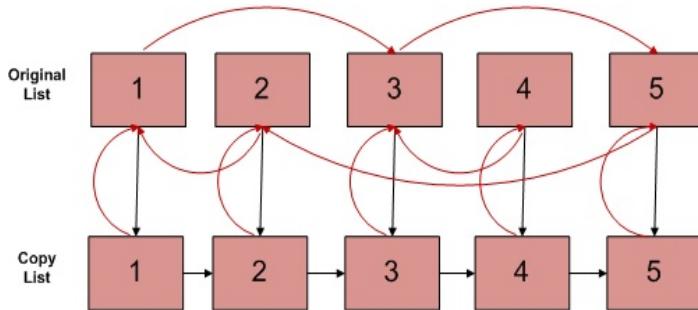


Figure 2

- 4) Change the arbit pointer of all nodes in copy linked list to point to corresponding node in original linked list.
- 5) Now construct the arbit pointer in copy linked list as below and restore the next pointer of nodes in the original linked list.

```
copy_list_node->arbit =  
    copy_list_node->arbit->arbit->next;  
copy_list_node = copy_list_node->next;
```

- 6) Restore the next pointers in original linked list from the stored mappings(in step 2).

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

Method 2 (Uses Constant Extra Space)

Thanks to Saravanan Mani for providing this solution. This solution works using constant space.

- 1) Create the copy of node 1 and insert it between node 1 & node 2 in original Linked List, create the copy of 2 and insert it between 2 & 3.. Continue in this fashion, add the copy of N after the Nth node
- 2) Now copy the arbitrary link in this fashion

```
original->next->arbitrary = original->arbitrary->next; /*TRAVERSE  
TWO NODES*/
```

- This works because `original->next` is nothing but copy of `original` and `Original->arbitrary->next` is nothing but copy of `arbitrary`.
- 3) Now restore the original and copy linked lists in this fashion in a single loop.

```
original->next = original->next->next;  
copy->next = copy->next->next;
```

- 4) Make sure that last element of `original->next` is `NULL`.

Refer below post for implementation of this method.

Clone a linked list with next and random pointer in $O(1)$ space

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

Refer Following Post for Hashing based Implementation.

Clone a linked list with next and random pointer | Set 2

Asked by Varun Bhatia. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Linked Lists

Connect nodes at same level

Write a function to connect all the adjacent nodes at the same level in a binary tree. Structure of the given Binary Tree node is like following.

```

struct node{
    int data;
    struct node* left;
    struct node* right;
    struct node* nextRight;
}

```

Initially, all the nextRight pointers point to garbage values. Your function should set these pointers to point next right for each node.

Example

Input Tree

```

A
 \
B C
/\ \
D E F

```

Output Tree

```

A-->NULL
 \
B-->C-->NULL
/\ \
D-->E-->F-->NULL

```

Method 1 (Extend Level Order Traversal or BFS)

Consider the method 2 of [Level Order Traversal](#). The method 2 can easily be extended to connect nodes of same level. We can augment queue entries to contain level of nodes also which is 0 for root, 1 for root's children and so on. So a queue node will now contain a pointer to a tree node and an integer level. When we enqueue a node, we make sure that correct level value for node is being set in queue. To set nextRight, for every node N, we dequeue the next node from queue, if the level number of next node is same, we set the nextRight of N as address of the dequeued node, otherwise we set nextRight of N as NULL.

Time Complexity: O(n)

Method 2 (Extend Pre Order Traversal)

This approach works only for [Complete Binary Trees](#). In this method we set nextRight in Pre Order fashion to make sure that the nextRight of parent is set before its children. When we are at node p, we set the nextRight of its left and right children. Since the tree is complete tree, nextRight of p's left child (`p->left->nextRight`) will always be p's right child, and nextRight of p's right child (`p->right->nextRight`) will always be left child of p's nextRight (if p is not the rightmost node at its level). If p is the rightmost node, then nextRight of p's right child will be NULL.

C

```

#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *left;
    struct node *right;
    struct node *nextRight;
};

void connectRecur(struct node* p);

// Sets the nextRight of root and calls connectRecur() for other nodes
void connect (struct node *p)
{
    // Set the nextRight for root
    p->nextRight = NULL;

    // Set the next right for rest of the nodes (other than root)
    connectRecur(p);
}

/* Set next right of all descendants of p.
   Assumption: p is a compete binary tree */
void connectRecur(struct node* p)
{
    // Base case

```

```

if (!p)
    return;

// Set the nextRight pointer for p's left child
if (p->left)
    p->left->nextRight = p->right;

// Set the nextRight pointer for p's right child
// p->nextRight will be NULL if p is the right most child at its level
if (p->right)
    p->right->nextRight = (p->nextRight) ? p->nextRight->left : NULL;

// Set nextRight for other nodes in pre order fashion
connectRecur(p->left);
connectRecur(p->right);
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->nextRight = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
       10
      / \
     8   2
      /
     3
    */
    struct node *root = newnode(10);
    root->left = newnode(8);
    root->right = newnode(2);
    root->left->left = newnode(3);

    // Populates nextRight pointer in all nodes
    connect(root);

    // Let us check the values of nextRight pointers
    printf("Following are populated nextRight pointers in the tree "
           "(-1 is printed if there is no nextRight) \n");
    printf("nextRight of %d is %d \n", root->data,
           root->nextRight? root->nextRight->data : -1);
    printf("nextRight of %d is %d \n", root->left->data,
           root->left->nextRight? root->left->nextRight->data : -1);
    printf("nextRight of %d is %d \n", root->right->data,
           root->right->nextRight? root->right->nextRight->data : -1);
    printf("nextRight of %d is %d \n", root->left->left->data,
           root->left->left->nextRight? root->left->left->nextRight->data : -1);

    getchar();
    return 0;
}

```

Java

```

// Java program to connect nodes at same level using extended
// pre-order traversal

// A binary tree node

```

```

class Node
{
    int data;
    Node left, right, nextRight;

    Node(int item)
    {
        data = item;
        left = right = nextRight = null;
    }
}

class BinaryTree
{
    Node root;

    // Sets the nextRight of root and calls connectRecur() for other nodes
    void connect(Node p)
    {

        // Set the nextRight for root
        p.nextRight = null;

        // Set the next right for rest of the nodes (other than root)
        connectRecur(p);
    }

    /* Set next right of all descendants of p.
     * Assumption: p is a complete binary tree */
    void connectRecur(Node p)
    {
        // Base case
        if (p == null)
            return;

        // Set the nextRight pointer for p's left child
        if (p.left != null)
            p.left.nextRight = p.right;

        // Set the nextRight pointer for p's right child
        // p->nextRight will be NULL if p is the right most child
        // at its level
        if (p.right != null)
            p.right.nextRight = (p.nextRight != null) ?
                p.nextRight.left : null;

        // Set nextRight for other nodes in pre order fashion
        connectRecur(p.left);
        connectRecur(p.right);
    }

    // Driver program to test above functions
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();

        /* Constructed binary tree is
           10
           / \
          8   2
          /
          3
         */
        tree.root = new Node(10);
        tree.root.left = new Node(8);
        tree.root.right = new Node(2);
        tree.root.left.left = new Node(3);

        // Populates nextRight pointer in all nodes
        tree.connect(tree.root);

        // Let us check the values of nextRight pointers
        System.out.println("Following are populated nextRight pointers in "
            + "the tree" + "(-1 is printed if there is no nextRight)");
        int a = tree.root.nextRight != null ? tree.root.nextRight.data : -1;
    }
}

```

```

System.out.println("nextRight of " + tree.root.data + " is "
+ a);
int b = tree.root.left.nextRight != null ?
        tree.root.left.nextRight.data : -1;
System.out.println("nextRight of " + tree.root.left.data + " is "
+ b);
int c = tree.root.right.nextRight != null ?
        tree.root.right.nextRight.data : -1;
System.out.println("nextRight of " + tree.root.right.data + " is "
+ c);
int d = tree.root.left.left.nextRight != null ?
        tree.root.left.left.nextRight.data : -1;
System.out.println("nextRight of " + tree.root.left.left.data + " is "
+ d);

}

}

// This code has been contributed by Mayank Jaiswal

```

Thanks to Dhanya for suggesting this approach.

Time Complexity: O(n)

Why doesn't method 2 work for trees which are not Complete Binary Trees?

Let us consider following tree as an example. In Method 2, we set the nextRight pointer in pre order fashion. When we are at node 4, we set the nextRight of its children which are 8 and 9 (the nextRight of 4 is already set as node 5). nextRight of 8 will simply be set as 9, but nextRight of 9 will be set as NULL which is incorrect. We can't set the correct nextRight, because when we set nextRight of 9, we only have nextRight of node 4 and ancestors of node 4, we don't have nextRight of nodes in right subtree of root.



See [Connect nodes at same level using constant extra space](#) for more solutions.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Trees

Lowest Common Ancestor in a Binary Tree | Set 1

Given a binary tree (not a binary search tree) and two values say n1 and n2, write a program to find the least common ancestor.

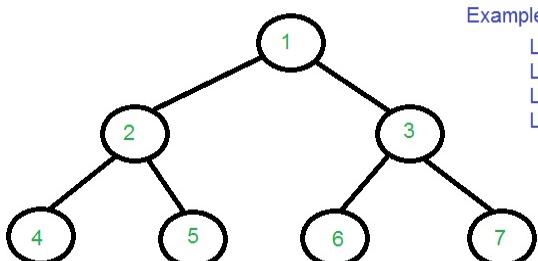
Following is definition of LCA from Wikipedia:

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source [Wiki](#))

Examples

$LCA(4, 5) = 2$
 $LCA(4, 6) = 1$
 $LCA(3, 4) = 1$
 $LCA(2, 4) = 2$



We have discussed an efficient solution to find [LCA in Binary Search Tree](#). In Binary Search Tree, using BST properties, we can find LCA in

$O(h)$ time where h is height of tree. Such an implementation is not possible in Binary Tree as keys Binary Tree nodes don't follow any order. Following are different approaches to find LCA in Binary Tree.

Method 1 (By Storing root to n1 and root to n2 paths):

Following is simple $O(n)$ algorithm to find LCA of $n1$ and $n2$.

- 1) Find path from root to $n1$ and store it in a vector or array.
- 2) Find path from root to $n2$ and store it in another vector or array.
- 3) Traverse both paths till the values in arrays are same. Return the common element just before the mismatch.

Following is C++ implementation of above algorithm.

C++

```
// A O(n) solution to find LCA of two given values n1 and n2
#include <iostream>
#include <vector>
using namespace std;

// A Binary Tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// Utility function creates a new binary tree node with given key
Node * newNode(int k)
{
    Node *temp = new Node;
    temp->key = k;
    temp->left = temp->right = NULL;
    return temp;
}

// Finds the path from root node to given root of the tree, Stores the
// path in a vector path[], returns true if path exists otherwise false
bool findPath(Node *root, vector<int> &path, int k)
{
    // base case
    if (root == NULL) return false;

    // Store this node in path vector. The node will be removed if
    // not in path from root to k
    path.push_back(root->key);

    // See if the k is same as root's key
    if (root->key == k)
        return true;

    // Check if k is found in left or right sub-tree
    if ( (root->left && findPath(root->left, path, k)) ||
        (root->right && findPath(root->right, path, k)) )
        return true;

    // If not present in subtree rooted with root, remove root from
    // path[] and return false
    path.pop_back();
    return false;
}

// Returns LCA if node n1, n2 are present in the given binary tree,
// otherwise return -1
int findLCA(Node *root, int n1, int n2)
{
    // to store paths to n1 and n2 from the root
    vector<int> path1, path2;

    // Find paths from root to n1 and root to n1. If either n1 or n2
    // is not present, return -1
    if ( !findPath(root, path1, n1) || !findPath(root, path2, n2) )
        return -1;
```

```

/* Compare the paths to get the first different value */
int i;
for (i = 0; i < path1.size() && i < path2.size(); i++)
    if (path1[i] != path2[i])
        break;
return path1[i-1];
}

// Driver program to test above functions
int main()
{
    // Let us create the Binary Tree shown in above diagram.
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5);
    cout << "\nLCA(4, 6) = " << findLCA(root, 4, 6);
    cout << "\nLCA(3, 4) = " << findLCA(root, 3, 4);
    cout << "\nLCA(2, 4) = " << findLCA(root, 2, 4);
    return 0;
}

```

Python

```

# O(n) solution to find LCS of two given values n1 and n2

# A binary tree node
class Node:
    # Constructor to create a new binary node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    # Finds the path from root node to given root of the tree.
    # Stores the path in a list path[], returns true if path
    # exists otherwise false
    def findPath( root, path, k):

        # Base Case
        if root is None:
            return False

        # Store this node in path vector. The node will be
        # removed if not in path from root to k
        path.append(root.key)

        # See if the k is same as root's key
        if root.key == k :
            return True

        # Check if k is found in left or right sub-tree
        if ((root.left != None and findPath(root.left, path, k)) or
            (root.right != None and findPath(root.right, path, k))):
            return True

        # If not present in subtree rooted with root, remove
        # root from path and return False

        path.pop()
        return False

    # Returns LCA if node n1 , n2 are present in the given
    # binary tree otherwise return -1
    def findLCA(root, n1, n2):

        # To store paths to n1 and n2 from the root
        path1 = []
        path2 = []

```

```
# Find paths from root to n1 and root to n2.
```

```
# If either n1 or n2 is not present , return -1
```

```
if (not findPath(root, path1, n1) or not findPath(root, path2, n2)):
```

```
    return -1
```

```
# Compare the paths to get the first different value
```

```
i = 0
```

```
while(i < len(path1) and i < len(path2)):
```

```
    if path1[i] != path2[i]:
```

```
        break
```

```
    i += 1
```

```
return path1[i-1]
```

```
# Driver program to test above function
```

```
# Let's create the Binary Tree shown in above diagram
```

```
root = Node(1)
```

```
root.left = Node(2)
```

```
root.right = Node(3)
```

```
root.left.left = Node(4)
```

```
root.left.right = Node(5)
```

```
root.right.left = Node(6)
```

```
root.right.right = Node(7)
```

```
print "LCA(4, 5) = %d" %(findLCA(root, 4, 5))
```

```
print "LCA(4, 6) = %d" %(findLCA(root, 4, 6))
```

```
print "LCA(3, 4) = %d" %(findLCA(root, 3, 4))
```

```
print "LCA(2, 4) = %d" %(findLCA(root, 2, 4))
```

```
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
LCA(4, 5) = 2
```

```
LCA(4, 6) = 1
```

```
LCA(3, 4) = 1
```

```
LCA(2, 4) = 2
```

Time Complexity: Time complexity of the above solution is O(n). The tree is traversed twice, and then path arrays are compared.

Thanks to *Ravi Chandra Enaganti* for suggesting the initial solution based on this method.

Method 2 (Using Single Traversal)

The method 1 finds LCA in O(n) time, but requires three tree traversals plus extra spaces for path arrays. If we assume that the keys n1 and n2 are present in Binary Tree, we can find LCA using single traversal of Binary Tree and without extra storage for path arrays.

The idea is to traverse the tree starting from root. If any of the given keys (n1 and n2) matches with root, then root is LCA (assuming that both keys are present). If root doesn't match with any of the keys, we recur for left and right subtree. The node which has one key present in its left subtree and the other key present in right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA also, otherwise LCA lies in right subtree.

C

```
/* Program to find LCA of n1 and n2 using one traversal of Binary Tree */
```

```
#include <iostream>
```

```
using namespace std;
```

```
// A Binary Tree Node
```

```
struct Node
```

```
{
```

```
    struct Node *left, *right;
```

```
    int key;
```

```
};
```

```
// Utility function to create a new tree Node
```

```
Node* newNode(int key)
```

```
{
```

```
    Node *temp = new Node;
```

```
    temp->key = key;
```

```
    temp->left = temp->right = NULL;
```

```
    return temp;
```

```
}
```

```

// This function returns pointer to LCA of two given values n1 and n2.
// This function assumes that n1 and n2 are present in Binary Tree
struct Node *findLCA(struct Node* root, int n1, int n2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
    // ancestor of other, then the ancestor key becomes LCA
    if (root->key == n1 || root->key == n2)
        return root;

    // Look for keys in left and right subtrees
    Node *left_lca = findLCA(root->left, n1, n2);
    Node *right_lca = findLCA(root->right, n1, n2);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca && right_lca) return root;

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != NULL)? left_lca: right_lca;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5)->key;
    cout << "\nLCA(4, 6) = " << findLCA(root, 4, 6)->key;
    cout << "\nLCA(3, 4) = " << findLCA(root, 3, 4)->key;
    cout << "\nLCA(2, 4) = " << findLCA(root, 2, 4)->key;
    return 0;
}

```

Java

```

//Java implementation to find lowest common ancestor of
// n1 and n2 using one traversal of binary tree

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

public class BinaryTree
{
    //Root of the Binary Tree
    Node root;

    Node findLCA(int n1, int n2)
    {
        return findLCA(root, n1, n2);
    }
}

```

```

// This function returns pointer to LCA of two given
// values n1 and n2. This function assumes that n1 and
// n2 are present in Binary Tree
Node findLCA(Node node, int n1, int n2)
{
    // Base case
    if (node == null)
        return null;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
    // ancestor of other, then the ancestor key becomes LCA
    if (node.data == n1 || node.data == n2)
        return node;

    // Look for keys in left and right subtrees
    Node left_lca = findLCA(node.left, n1, n2);
    Node right_lca = findLCA(node.right, n1, n2);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca!=null && right_lca!=null)
        return node;

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != null) ? left_lca : right_lca;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
    tree.root.right.right = new Node(7);
    System.out.println("LCA(4, 5) = " +
        tree.findLCA(4, 5).data);
    System.out.println("LCA(4, 6) = " +
        tree.findLCA(4, 6).data);
    System.out.println("LCA(3, 4) = " +
        tree.findLCA(3, 4).data);
    System.out.println("LCA(2, 4) = " +
        tree.findLCA(2, 4).data);
}
}

```

Python

```

# Python program to find LCA of n1 and n2 using one
# traversal of Binary tree

# A binary tree node
class Node:

    # Constructor to create a new tree node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    # This function returns pointer to LCA of two given
    # values n1 and n2
    # This function assumes that n1 and n2 are present in
    # Binary Tree
    def findLCA(root, n1, n2):

```

```

# Base Case
if root is None:
    return None

# If either n1 or n2 matches with root's key, report
# the presence by returning root (Note that if a key is
# ancestor of other, then the ancestor key becomes LCA
if root.key == n1 or root.key == n2:
    return root

# Look for keys in left and right subtrees
left_lca = findLCA(root.left, n1, n2)
right_lca = findLCA(root.right, n1, n2)

# If both of the above calls return Non-NULL, then one key
# is present in once subtree and other is present in other,
# So this node is the LCA
if left_lca and right_lca:
    return root

# Otherwise check if left subtree or right subtree is LCA
return left_lca if left_lca is not None else right_lca

# Driver program to test above function

# Let us create a binary tree given in the above example
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
print "LCA(4,5) = ", findLCA(root, 4, 5).key
print "LCA(4,6) = ", findLCA(root, 4, 6).key
print "LCA(3,4) = ", findLCA(root, 3, 4).key
print "LCA(2,4) = ", findLCA(root, 2, 4).key

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

LCA(4, 5) = 2
LCA(4, 6) = 1
LCA(3, 4) = 1
LCA(2, 4) = 2

```

Thanks to *Atul Singh* for suggesting this solution.

Time Complexity: Time complexity of the above solution is O(n) as the method does a simple tree traversal in bottom up fashion. Note that the above method assumes that keys are present in Binary Tree. If one key is present and other is absent, then it returns the present key as LCA (Ideally should have returned NULL). We can extend this method to handle all cases by passing two boolean variables v1 and v2. v1 is set as true when n1 is present in tree and v2 is set as true if n2 is present in tree.

C

```

/* Program to find LCA of n1 and n2 using one traversal of Binary Tree.
It handles all cases even when n1 or n2 is not there in Binary Tree */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};

// Utility function to create a new tree Node
Node* newNode(int key)

```

```

{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// This function returns pointer to LCA of two given values n1 and n2.
// v1 is set as true by this function if n1 is found
// v2 is set as true by this function if n2 is found
struct Node *findLCAUtil(struct Node* root, int n1, int n2, bool &v1, bool &v2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report the presence
    // by setting v1 or v2 as true and return root (Note that if a key
    // is ancestor of other, then the ancestor key becomes LCA)
    if (root->key == n1)
    {
        v1 = true;
        return root;
    }
    if (root->key == n2)
    {
        v2 = true;
        return root;
    }

    // Look for keys in left and right subtrees
    Node *left_lca = findLCAUtil(root->left, n1, n2, v1, v2);
    Node *right_lca = findLCAUtil(root->right, n1, n2, v1, v2);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca && right_lca) return root;

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != NULL)? left_lca: right_lca;
}

// Returns true if key k is present in tree rooted with root
bool find(Node *root, int k)
{
    // Base Case
    if (root == NULL)
        return false;

    // If key is present at root, or in left subtree or right subtree,
    // return true;
    if (root->key == k || find(root->left, k) || find(root->right, k))
        return true;

    // Else return false
    return false;
}

// This function returns LCA of n1 and n2 only if both n1 and n2 are present
// in tree, otherwise returns NULL;
Node *findLCA(Node *root, int n1, int n2)
{
    // Initialize n1 and n2 as not visited
    bool v1 = false, v2 = false;

    // Find lca of n1 and n2 using the technique discussed above
    Node *lca = findLCAUtil(root, n1, n2, v1, v2);

    // Return LCA only if both n1 and n2 are present in tree
    if (v1 && v2 || v1 && find(lca, n2) || v2 && find(lca, n1))
        return lca;

    // Else return NULL
    return NULL;
}

```

```

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    Node *lca = findLCA(root, 4, 5);
    if (lca != NULL)
        cout << "LCA(4, 5) = " << lca->key;
    else
        cout << "Keys are not present ";

    lca = findLCA(root, 4, 10);
    if (lca != NULL)
        cout << "\nLCA(4, 10) = " << lca->key;
    else
        cout << "\nKeys are not present ";

    return 0;
}

```

Java

```

// Java implementation to find lowest common ancestor of
// n1 and n2 using one traversal of binary tree
// It also handles cases even when n1 and n2 are not there in Tree

/* Class containing left and right child of current node and key */
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

public class BinaryTree
{
    // Root of the Binary Tree
    Node root;
    static boolean v1 = false, v2 = false;

    // This function returns pointer to LCA of two given
    // values n1 and n2.
    // v1 is set as true by this function if n1 is found
    // v2 is set as true by this function if n2 is found
    Node findLCAUtil(Node node, int n1, int n2)
    {
        // Base case
        if (node == null)
            return null;

        // If either n1 or n2 matches with root's key, report the presence
        // by setting v1 or v2 as true and return root (Note that if a key
        // is ancestor of other, then the ancestor key becomes LCA)
        if (node.data == n1)
        {
            v1 = true;
            return node;
        }
        if (node.data == n2)
        {
            v2 = true;
        }
    }
}

```

```

        return node;
    }

    // Look for keys in left and right subtrees
    Node left_lca = findLCAUtil(node.left, n1, n2);
    Node right_lca = findLCAUtil(node.right, n1, n2);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca != null && right_lca != null)
        return node;

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != null) ? left_lca : right_lca;
}

// Finds lca of n1 and n2 under the subtree rooted with 'node'
Node findLCA(int n1, int n2)
{
    // Initialize n1 and n2 as not visited
    v1 = false;
    v2 = false;

    // Find lca of n1 and n2 using the technique discussed above
    Node lca = findLCAUtil(root, n1, n2);

    // Return LCA only if both n1 and n2 are present in tree
    if (v1 && v2)
        return lca;

    // Else return NULL
    return null;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTreeNode tree = new BinaryTreeNode();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
    tree.root.right.right = new Node(7);

    Node lca = tree.findLCA(4, 5);
    if (lca != null)
        System.out.println("LCA(4, 5) = " + lca.data);
    else
        System.out.println("Keys are not present");

    lca = tree.findLCA(4, 10);
    if (lca != null)
        System.out.println("LCA(4, 10) = " + lca.data);
    else
        System.out.println("Keys are not present");
}
}

```

Python

```

"""
Program to find LCA of n1 and n2 using one traversal of
Binary tree
It handles all cases even when n1 or n2 is not there in tree
"""

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

```

```

def __init__(self, key):
    self.key = key
    self.left = None
    self.right = None

# This function return pointer to LCA of two given values
# n1 and n2
# v1 is set as true by this function if n1 is found
# v2 is set as true by this function if n2 is found
def findLCAUtil(root, n1, n2, v):

    # Base Case
    if root is None:
        return None

    # If either n1 or n2 matches ith root's key, report
    # the presence by setting v1 or v2 as true and return
    # root (Note that if a key is ancestor of other, then
    # the ancestor key becomes LCA)
    if root.key == n1 :
        v[0] = True
        return root

    if root.key == n2:
        v[1] = True
        return root

    # Look for keys in left and right subtree
    left_lca = findLCAUtil(root.left, n1, n2, v)
    right_lca = findLCAUtil(root.right, n1, n2, v)

    # If both of the above calls return Non-NULL, then one key
    # is present in once subtree and other is present in other,
    # So this node is the LCA
    if left_lca and right_lca:
        return root

    # Otherwise check if left subtree or right subtree is LCA
    return left_lca if left_lca is not None else right_lca

def find(root, k):

    # Base Case
    if root is None:
        return False

    # If key is present at root, or if left subtree or right
    # subtree , return true
    if (root.key == k or find(root.left, k) or
        find(root.right, k)):
        return True

    # Else return false
    return False

# This function returns LCA of n1 and n2 onlue if both
# n1 and n2 are present in tree, otherwise returns None
def findLCA(root, n1, n2):

    # Initialize n1 and n2 as not visited
    v = [False, False]

    # Find lac of n1 and n2 using the technique discussed above
    lca = findLCAUtil(root, n1, n2, v)

    # Returns LCA only if both n1 and n2 are present in tree
    if (v[0] and v[1] or v[0] and find(lca, n2) or v[1] and
        find(lca, n1)):
        return lca

    # Else return None
    return None

# Driver program to test above function

```

```

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

lca = findLCA(root, 4, 5)

if lca is not None:
    print "LCA(4, 5) = ", lca.key
else :
    print "Keys are not present"

lca = findLCA(root, 4, 10)
if lca is not None:
    print "LCA(4,10) = ", lca.key
else:
    print "Keys are not present"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```
LCA(4, 5) = 2
Keys are not present
```

Thanks to Dhruv for suggesting this extended solution.

You may like to see below articles as well :

[LCA using Parent Pointer](#)

[Lowest Common Ancestor in a Binary Search Tree.](#)

[Find LCA in Binary Tree using RMQ](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

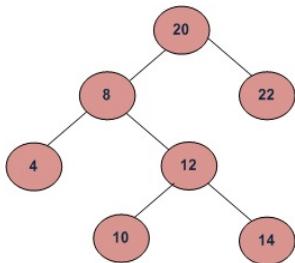
Trees
LCA

Lowest Common Ancestor in a Binary Search Tree.

Given values of two nodes in a Binary Search Tree, write a c program to find the Lowest Common Ancestor (LCA). You may assume that both the values exist in the tree.

The function prototype should be as follows:

```
struct node *lca(node* root, int n1, int n2)
n1 and n2 are two given values in the tree with given root.
```



For example, consider the BST in diagram, LCA of 10 and 14 is 12 and LCA of 8 and 14 is 8.

Following is definition of LCA from Wikipedia:

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors

may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source [Wiki](#))

Solutions:

If we are given a BST where every node has **parent pointer**, then LCA can be easily determined by traversing up using parent pointer and printing the first intersecting node.

We can solve this problem using BST properties. We can **recursively traverse** the BST from root. The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2, i.e., $n_1 < n < n_2$ or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that $n_1 < n_2$). So just recursively traverse the BST in, if node's value is greater than both n1 and n2 then our LCA lies in left side of the node, if it's smaller than both n1 and n2, then LCA lies on right side. Otherwise root is LCA (assuming that both n1 and n2 are present in BST)

C

```
// A recursive C program to find LCA of two nodes n1 and n2.
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node* left, *right;
};

/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    if (root == NULL) return NULL;

    // If both n1 and n2 are smaller than root, then LCA lies in left
    if (root->data > n1 && root->data > n2)
        return lca(root->left, n1, n2);

    // If both n1 and n2 are greater than root, then LCA lies in right
    if (root->data < n1 && root->data < n2)
        return lca(root->right, n1, n2);

    return root;
}

/* Helper function that allocates a new node with the given data.*/
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

/* Driver program to test lca() */
int main()
{
    // Let us construct the BST shown in the above figure
    struct node *root      = newNode(20);
    root->left       = newNode(8);
    root->right      = newNode(22);
    root->left->left    = newNode(4);
    root->left->right   = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);

    int n1 = 10, n2 = 14;
    struct node *t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    n1 = 14, n2 = 8;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);
}
```

```

n1 = 10, n2 = 22;
t = lca(root, n1, n2);
printf("LCA of %d and %d is %d \n", n1, n2, t->data);

getchar();
return 0;
}

```

Java

```

// Recursive Java program to print lca of two nodes

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Function to find LCA of n1 and n2. The function assumes that both
       n1 and n2 are present in BST */
    Node lca(Node node, int n1, int n2)
    {
        if (node == null)
            return null;

        // If both n1 and n2 are smaller than root, then LCA lies in left
        if (node.data > n1 && node.data > n2)
            return lca(node.left, n1, n2);

        // If both n1 and n2 are greater than root, then LCA lies in right
        if (node.data < n1 && node.data < n2)
            return lca(node.right, n1, n2);

        return node;
    }

    /* Driver program to test lca() */
    public static void main(String args[])
    {
        // Let us construct the BST shown in the above figure
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(20);
        tree.root.left = new Node(8);
        tree.root.right = new Node(22);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(12);
        tree.root.left.right.left = new Node(10);
        tree.root.left.right.right = new Node(14);

        int n1 = 10, n2 = 14;
        Node t = tree.lca(tree.root, n1, n2);
        System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

        n1 = 14;
        n2 = 8;
        t = tree.lca(tree.root, n1, n2);
        System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

        n1 = 10;
        n2 = 22;
        t = tree.lca(tree.root, n1, n2);
        System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);
    }
}

```

```
}
```

```
// This code has been contributed by Mayank Jaiswal
```

Python

```
# A recursive python program to find LCA of two nodes
# n1 and n2

# A Binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Function to find LCA of n1 and n2. The function assumes
# that both n1 and n2 are present in BST
def lca(root, n1, n2):

    # Base Case
    if root is None:
        return None

    # If both n1 and n2 are smaller than root, then LCA
    # lies in left
    if(root.data > n1 and root.data > n2):
        return lca(root.left, n1, n2)

    # If both n1 and n2 are greater than root, then LCA
    # lies in right
    if(root.data < n1 and root.data < n2):
        return lca(root.right, n1, n2)

    return root

# Driver program to test above function

# Let us construct the BST shown in the figure
root = Node(20)
root.left = Node(8)
root.right = Node(22)
root.left.left = Node(4)
root.left.right = Node(12)
root.left.right.left = Node(10)
root.left.right.right = Node(14)

n1 = 10 ; n2 = 14
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

n1 = 14 ; n2 = 8
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

n1 = 10 ; n2 = 22
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20
```

Time complexity of above solution is O(h) where h is height of tree. Also, the above solution requires O(h) extra space in function call stack for recursive function calls. We can avoid extra space using **iterative solution**.

```
/* Function to find LCA of n1 and n2. The function assumes that both
n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    while (root != NULL)
    {
        // If both n1 and n2 are smaller than root, then LCA lies in left
        if (root->data > n1 && root->data > n2)
            root = root->left;

        // If both n1 and n2 are greater than root, then LCA lies in right
        else if (root->data < n1 && root->data < n2)
            root = root->right;

        else break;
    }
    return root;
}
```

See [this](#) for complete program.

You may like to see below articles as well :

[Lowest Common Ancestor in a Binary Tree](#)

[LCA using Parent Pointer](#)

[Find LCA in Binary Tree using RMQ](#)

Exercise

The above functions assume that n1 and n2 both are in BST. If n1 and n2 are not present, then they may return incorrect result. Extend the above solutions to return NULL if n1 or n2 or both not present in BST.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Trees
LCA

Run Length Encoding

Given an input string, write a function that returns the [Run Length Encoded](#) string for the input string.

For example, if the input string is “wwwwwwaaadxxxxxx”, then the function should return “w7a3d1e1x6”.

Algorithm:

- Pick the first character from source string.
- Append the picked character to the destination string.
- Count the number of subsequent occurrences of the picked character and append the count to destination string.
- Pick the next character and repeat steps b) c) and d) if end of string is NOT reached.

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAX_RLEN 50

/* Returns the Run Length Encoded string for the
source string src */
char *encode(char *src)
{
    int rLen;
    char count[MAX_RLEN];
    int len = strlen(src);

    /* If all characters in the source string are different,
    then size of destination string would be twice of input string.
    For example if the src is "abcd", then dest would be "a1b1c1d1"
    For other inputs, size would be less than twice. */
    char *dest = (char *)malloc(sizeof(char)*(len*2 + 1));
```

```

int i, j = 0, k;

/* traverse the input string one by one */
for(i = 0; i < len; i++)
{
    /* Copy the first occurrence of the new character */
    dest[j++] = src[i];

    /* Count the number of occurrences of the new character */
    rLen = 1;
    while(i + 1 < len && src[i] == src[i+1])
    {
        rLen++;
        i++;
    }

    /* Store rLen in a character array count[] */
    sprintf(count, "%d", rLen);

    /* Copy the count[] to destination */
    for(k = 0; *(count+k); k++, j++)
    {
        dest[j] = count[k];
    }
}

/*terminate the destination string */
dest[j] = '\0';
return dest;
}

/*driver program to test above function */
int main()
{
    char str[] = "geeksforgeeks";
    char *res = encode(str);
    printf("%s", res);
    getchar();
}

```

Time Complexity: O(n)

References:

http://en.wikipedia.org/wiki/Run-length_encoding

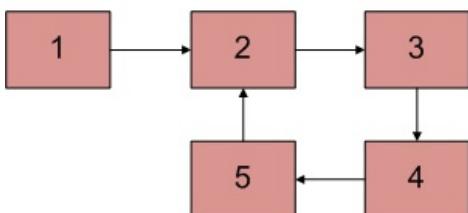
Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Strings

Detect and Remove Loop in a Linked List

Write a function *detectAndRemoveLoop()* that checks whether a given Linked List contains loop and if loop is present then removes the loop and returns true. And if the list doesn't contain loop then returns false. Below diagram shows a linked list with a loop. *detectAndRemoveLoop()* must change the below list to 1->2->3->4->5->NULL.



We also recommend to read following post as a prerequisite of the solution discussed here.

[Write a C function to detect loop in a linked list](#)

Before trying to remove the loop, we must detect it. Techniques discussed in the above post can be used to detect loop. To remove loop, all we need to do is to get pointer to the last node of the loop. For example, node with value 5 in the above diagram. Once we have pointer to the last node, we can make the next of this node as NULL and loop is gone.

We can easily use Hashing or Visited node techniques (discussed in the above mentioned post) to get the pointer to the last node. Idea is simple: the very first node whose next is already visited (or hashed) is the last node.

We can also use Floyd Cycle Detection algorithm to detect and remove the loop. In the Floyd's algo, the slow and fast pointers meet at a loop node. We can use this loop node to remove cycle. There are following two different ways of removing loop when Floyd's algorithm is used for Loop detection.

Method 1 (Check one by one)

We know that Floyd's Cycle detection algorithm terminates when fast and slow pointers meet at a common point. We also know that this common point is one of the loop nodes (2 or 3 or 4 or 5 in the above diagram). We store the address of this in a pointer variable say ptr2. Then we start from the head of the Linked List and check for nodes one by one if they are reachable from ptr2. When we find a node that is reachable, we know that this node is the starting node of the loop in Linked List and we can get pointer to the previous of this node.

C

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. Used by detectAndRemoveLoop() */
void removeLoop(struct node * , struct node *);

/* This function detects and removes loop in the list
   If loop was there in the list then it returns 1,
   otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indicate that there is no loop*/
    return 0;
}

/* Function to remove loop.
loop_node --> Pointer to one of the loop nodes
head --> Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
    struct node *ptr1;
    struct node *ptr2;

    /* Set a pointer to the beginning of the Linked List and
       move it one by one to find the first node which is
       part of the Linked List */
    ptr1 = head;
    while (1)
    {
```

```

/* Now start a pointer from loop_node and check if it ever
   reaches ptr2 */
ptr2 = loop_node;
while (ptr2->next != loop_node && ptr2->next != ptr1)
    ptr2 = ptr2->next;

/* If ptr2 reached ptr1 then there is a loop. So break the
   loop */
if (ptr2->next == ptr1)
    break;

/* If ptr2 didn't reach ptr1 then try the next node after ptr1 */
ptr1 = ptr1->next;
}

/* After the end of loop ptr2 is the last node of the loop. So
   make next of ptr2 as NULL */
ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

struct node *newNode(int key)
{
    struct node *temp = new struct node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Driver program to test above function*/
int main()
{
    struct node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);
    return 0;
}

```

Java

```

// Java program to detect and remove loop in linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }
}
```

```

        }

// Function that detects loop in the list
int detectAndRemoveLoop(Node node) {
    Node slow = node, fast = node;
    while (slow != null && fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;

        // If slow and fast meet at same point then loop is present
        if (slow == fast) {
            removeLoop(slow, node);
            return 1;
        }
    }
    return 0;
}

// Function to remove loop
void removeLoop(Node loop, Node curr) {
    Node ptr1 = null, ptr2 = null;

    /* Set a pointer to the begining of the Linked List and
       move it one by one to find the first node which is
       part of the Linked List */
    ptr1 = curr;
    while (1 == 1) {

        /* Now start a pointer from loop_node and check if it ever
           reaches ptr2 */
        ptr2 = loop;
        while (ptr2.next != loop && ptr2.next != ptr1) {
            ptr2 = ptr2.next;
        }

        /* If ptr2 reached ptr1 then there is a loop. So break the
           loop */
        if (ptr2.next == ptr1) {
            break;
        }

        /* If ptr2 didn't reach ptr1 then try the next node after ptr1 */
        ptr1 = ptr1.next;
    }

    /* After the end of loop ptr2 is the last node of the loop. So
       make next of ptr2 as NULL */
    ptr2.next = null;
}

// Function to print the linked list
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

// Driver program to test above functions
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(50);
    list.head.next = new Node(20);
    list.head.next.next = new Node(15);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(10);

    // Creating a loop for testing
    head.next.next.next.next = head.next.next;
    list.detectAndRemoveLoop(head);
    System.out.println("Linked List after removing loop : ");
    list.printList(head);
}
}

```

Python

```
# Python program to detect and remove loop in linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def detectAndRemoveLoop(self):
        slow_p = fast_p = self.head
        while(slow_p and fast_p and fast_p.next):
            slow_p = slow_p.next
            fast_p = fast_p.next.next

            # If slow_p and fast_p meet at some point
            # then there is a loop
            if slow_p == fast_p:
                self.removeLoop(slow_p)

        # Return 1 to indicate that loop is found
        return 1

    # Return 0 to indicate that there is no loop
    return 0

    # Function to remove loop
    # loop_node-> Pointer to one of the loop nodes
    # head --> Pointer to the start node of the
    # linked list
    def removeLoop(self, loop_node):

        # Set a pointer to the beginning of the linked
        # list and move it one by one to find the first
        # node which is part of the linked list
        ptr1 = self.head
        while(1):
            # Now start a pointer from loop_node and check
            # if it ever reaches ptr2
            ptr2 = loop_node
            while(ptr2.next!= loop_node and ptr2.next !=ptr1):
                ptr2 = ptr2.next

            # If ptr2 reached ptr1 then there is a loop.
            # So break the loop
            if ptr2.next == ptr1 :
                break

        ptr1 = ptr1.next

        # After the end of loop ptr2 is the last node of
        # the loop. So make next of ptr2 as NULL
        ptr2.next = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
```

```

def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program
llist = LinkedList()
llist.push(10)
llist.push(4)
llist.push(15)
llist.push(20)
llist.push(50)

# Create a loop for testing
llist.head.next.next.next.next = llist.head.next.next

llist.detectAndRemoveLoop()

print "Linked List after removing loop"
llist.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Linked List after removing loop
50 20 15 4 10

```

Method 2 (Better Solution)

This method is also dependent on Floyd's Cycle detection algorithm.

- 1) Detect Loop using Floyd's Cycle detection algo and get the pointer to a loop node.
- 2) Count the number of nodes in loop. Let the count be k.
- 3) Fix one pointer to the head and another to kth node from head.
- 4) Move both pointers at the same pace, they will meet at loop starting node.
- 5) Get pointer to the last node of loop and make next of it as NULL.

Thanks to WgpShashank for suggesting this method.

C

```

#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
   If loop was there in the list then it returns 1,
   otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {

```

```

removeLoop(slow_p, list);

    /* Return 1 to indicate that loop is found */
    return 1;
}
}

/* Return 0 to indicate that there is no loop*/
return 0;
}

/* Function to remove loop.
loop_node --> Pointer to one of the loop nodes
head --> Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
    struct node *ptr1 = loop_node;
    struct node *ptr2 = loop_node;

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2)
    {
        ptr1 = ptr1->next;
        k++;
    }

    // Fix one pointer to head
    ptr1 = head;

    // And the other pointer to k nodes after head
    ptr2 = head;
    for (i = 0; i < k; i++)
        ptr2 = ptr2->next;

    /* Move both pointers at the same pace,
    they will meet at loop starting node */
    while (ptr2 != ptr1)
    {
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }

    // Get pointer to the last node
    ptr2 = ptr2->next;
    while (ptr2->next != ptr1)
        ptr2 = ptr2->next;

    /* Set the next node of the loop ending node
    to fix the loop */
    ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

struct node *newNode(int key)
{
    struct node *temp = new struct node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Driver program to test above function*/
int main()
{
    struct node *head = newNode(50);
}

```

```

head->next = newNode(20);
head->next->next = newNode(15);
head->next->next->next = newNode(4);
head->next->next->next->next = newNode(10);

/* Create a loop for testing */
head->next->next->next->next->next = head->next->next;

detectAndRemoveLoop(head);

printf("Linked List after removing loop \n");
printList(head);
return 0;
}

```

Java

```

// Java program to detect and remove loop in linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    // Function that detects loop in the list
    int detectAndRemoveLoop(Node node) {
        Node slow = node, fast = node;
        while (slow != null && fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;

            // If slow and fast meet at same point then loop is present
            if (slow == fast) {
                removeLoop(slow, node);
                return 1;
            }
        }
        return 0;
    }

    // Function to remove loop
    void removeLoop(Node loop, Node head) {
        Node ptr1 = loop;
        Node ptr2 = loop;

        // Count the number of nodes in loop
        int k = 1, i;
        while (ptr1.next != ptr2) {
            ptr1 = ptr1.next;
            k++;
        }

        // Fix one pointer to head
        ptr1 = head;

        // And the other pointer to k nodes after head
        ptr2 = head;
        for (i = 0; i < k; i++) {
            ptr2 = ptr2.next;
        }

        /* Move both pointers at the same pace,
        they will meet at loop starting node */
    }
}

```

```

        while (ptr2 != ptr1) {
            ptr1 = ptr1.next;
            ptr2 = ptr2.next;
        }

        // Get pointer to the last node
        ptr2 = ptr2.next;
        while (ptr2.next != ptr1) {
            ptr2 = ptr2.next;
        }

        /* Set the next node of the loop ending node
        to fix the loop */
        ptr2.next = null;
    }

    // Function to print the linked list
    void printList(Node node) {
        while (node != null) {
            System.out.print(node.data + " ");
            node = node.next;
        }
    }

    // Driver program to test above functions
    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.head = new Node(50);
        list.head.next = new Node(20);
        list.head.next.next = new Node(15);
        list.head.next.next.next = new Node(4);
        list.head.next.next.next.next = new Node(10);

        // Creating a loop for testing
        head.next.next.next.next = head.next.next;
        list.detectAndRemoveLoop(head);
        System.out.println("Linked List after removing loop : ");
        list.printList(head);
    }
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to detect and remove loop in linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def detectAndRemoveLoop(self):
        slow_p = fast_p = self.head

        while(slow_p and fast_p and fast_p.next):
            slow_p = slow_p.next
            fast_p = fast_p.next.next

            # If slow_p and fast_p meet at some point then
            # there is a loop
            if slow_p == fast_p:

```

```

    self.removeLoop(slow_p)

    # Return 1 to indicate that loop is found
    return 1

    # Return 0 to indicate that there is no loop
    return 0

    # Function to remove loop
    # loop_node --> pointer to one of the loop nodes
    # head --> Pointer to the start node of the linked list
    def removeLoop(self, loop_node):
        ptr1 = loop_node
        ptr2 = loop_node

        # Count the number of nodes in loop
        k = 1
        while(ptr1.next != ptr2):
            ptr1 = ptr1.next
            k += 1

        # Fix one pointer to head
        ptr1 = self.head

        # And the other pointer to k nodes after head
        ptr2 = self.head
        for i in range(k):
            ptr2 = ptr2.next

        # Move both pointers at the same place
        # they will meet at loop starting node
        while(ptr2 != ptr1):
            ptr1 = ptr1.next
            ptr2 = ptr2.next

        # Get pointer to the last node
        ptr2 = ptr2.next
        while(ptr2.next != ptr1):
            ptr2 = ptr2.next

        # Set the next node of the loop ending node
        # to fix the loop
        ptr2.next = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

    # Driver program
    llist = LinkedList()
    llist.push(10)
    llist.push(4)
    llist.push(15)
    llist.push(20)
    llist.push(50)

    # Create a loop for testing
    llist.head.next.next.next.next = llist.head.next.next

    llist.detectAndRemoveLoop()

    print "Linked List after removing loop"
    llist.printList()

    # This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

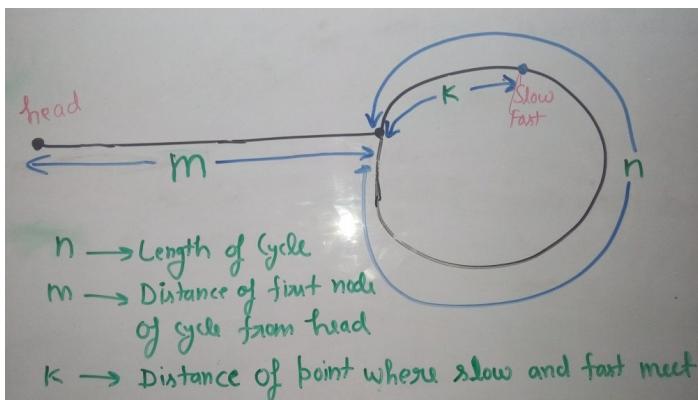
Linked List after removing loop
50 20 15 4 10

Method 3 (Optimized Method 2: Without Counting Nodes in Loop)

We do not need to count number of nodes in Loop. After detecting the loop, if we start slow pointer from head and move both slow and fast pointers at same speed until fast don't meet, they would meet at the beginning of linked list.

How does this work?

Let slow and fast meet at some point after Floyd's Cycle finding algorithm. Below diagram shows the situation when cycle is found.



We can conclude below from above diagram

Distance traveled by fast pointer = $2 * (\text{Distance traveled by slow pointer})$

$$(m + n*x + k) = 2*(m + n*y + k)$$

Note that before meeting the point shown above, fast was moving at twice speed.

$x \rightarrow$ Number of complete cyclic rounds made by fast pointer before they meet first time

$y \rightarrow$ Number of complete cyclic rounds made by slow pointer before they meet first time

From above equation, we can conclude below

$$m + k = (x - 2y)*n$$

Which means **m+k is a multiple of n**.

So if we start moving both pointers again at **same speed** such that one pointer (say slow) begins from head node of linked list and other pointer (say fast) begins from meeting point. When slow pointer reaches beginning of linked list (has made m steps). Fast pointer would have made also moved m steps as they are now moving same pace. Since m+k is a multiple of n and fast starts from k, they would meet at the beginning. Can they meet before also? No because slow pointer enters the cycle first time after m steps.

C++

```
// C++ program to detect and remove loop
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int key;
    struct Node *next;
};

Node *newNode(int key)
{
    Node *temp = new Node;
```

```

temp->key = key;
temp->next = NULL;
return temp;
}

// A utility function to print a linked list
void printList(Node *head)
{
    while (head != NULL)
    {
        cout << head->key << " ";
        head = head->next;
    }
    cout << endl;
}

void detectAndRemoveLoop(Node *head)
{
    Node *slow = head;
    Node *fast = head->next;

    // Search for loop using slow and fast pointers
    while (fast && fast->next)
    {
        if (slow == fast)
            break;
        slow = slow->next;
        fast = fast->next->next;
    }

    /* If loop exists */
    if (slow == fast)
    {
        slow = head;
        while (slow != fast->next)
        {
            slow = slow->next;
            fast = fast->next;
        }

        /* since fast->next is the looping point */
        fast->next = NULL; /* remove loop */
    }
}

/* Driver program to test above function*/
int main()
{
    Node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    return 0;
}

```

Java

```

// Java program to detect and remove loop in linked list

class LinkedList {

    static Node head;

```

```

static class Node {

    int data;
    Node next;

    Node(int d) {
        data = d;
        next = null;
    }
}

// Function that detects loop in the list
void detectAndRemoveLoop(Node node) {
    Node slow = node;
    Node fast = node.next;

    // Search for loop using slow and fast pointers
    while (fast != null && fast.next != null) {
        if (slow == fast) {
            break;
        }
        slow = slow.next;
        fast = fast.next.next;
    }

    /* If loop exists */
    if (slow == fast) {
        slow = node;
        while (slow != fast.next) {
            slow = slow.next;
            fast = fast.next;
        }

        /* since fast->next is the looping point */
        fast.next = null; /* remove loop */
    }
}

// Function to print the linked list
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

// Driver program to test above functions
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(50);
    list.head.next = new Node(20);
    list.head.next.next = new Node(15);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(10);

    // Creating a loop for testing
    head.next.next.next.next = head.next.next;
    list.detectAndRemoveLoop(head);
    System.out.println("Linked List after removing loop : ");
    list.printList(head);
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to detect and remove loop

# Node class

```

```

class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    def detectAndRemoveLoop(self):
        slow = self.head
        fast = self.head.next

        # Search for loop using slow and fast pointers
        while(fast is not None):
            if fast.next is None:
                break
            if slow == fast :
                break
            slow = slow.next
            fast = fast.next.next

        # if loop exists
        if slow == fast :
            slow = self.head
            while(slow != fast.next):
                slow = slow.next
                fast = fast.next

            # Since fast.next is the looping point
            fast.next = None # Remove loop

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

# Driver program
llist = LinkedList()
llist.head = Node(50)
llist.head.next = Node(20)
llist.head.next.next = Node(15)
llist.head.next.next.next = Node(4)
llist.head.next.next.next.next = Node(10)

#Create a loop for testing
llist.head.next.next.next.next = llist.head.next.next

llist.detectAndRemoveLoop()

print "Linked List after removing loop"
llist.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

Linked List after removing loop
50 20 15 4 10

Thanks to Gaurav Ahirwar for suggesting above solution.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Linked Lists

Find the repeating and the missing | Added 3 new methods

Given an unsorted array of size n. Array elements are in range from 1 to n. One number from set {1, 2, ...n} is missing and one number occurs twice in array. Find these two numbers.

Examples:

```
arr[] = {3, 1, 3}  
Output: 2, 3 // 2 is missing and 3 occurs twice
```

```
arr[] = {4, 3, 6, 2, 1, 1}  
Output: 1, 5 // 5 is missing and 1 occurs twice
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Method 1 (Use Sorting)

- 1) Sort the input array.
- 2) Traverse the array and check for missing and repeating.

Time Complexity: O(nLogn)

Thanks to LoneShadow for suggesting this method.

Method 2 (Use count array)

- 1) Create a temp array temp[] of size n with all initial values as 0.
- 2) Traverse the input array arr[], and do following for each arr[i]
.....a) if(temp[arr[i]] == 0) temp[arr[i]] = 1;
.....b) if(temp[arr[i]] == 1) output "arr[i]" //repeating
- 3) Traverse temp[] and output the array element having value as 0 (This is the missing element)

Time Complexity: O(n)

Auxiliary Space: O(n)

Method 3 (Use elements as Index and mark the visited places)

Traverse the array. While traversing, use absolute value of every element as index and make the value at this index as negative to mark it visited. If something is already marked negative then this is the repeating element. To find missing, traverse the array again and look for a positive value.

```
#include<stdio.h>  
#include<stdlib.h>  
  
void printTwoElements(int arr[], int size)  
{  
    int i;  
    printf("\n The repeating element is");  
  
    for(i = 0; i < size; i++)  
    {  
        if(arr[abs(arr[i])-1] > 0)  
            arr[abs(arr[i])-1] = -arr[abs(arr[i])-1];  
        else  
            printf(" %d ", abs(arr[i]));  
    }  
  
    printf("\nand the missing element is");  
    for(i=0; i<size; i++)  
    {  
        if(arr[i]>0)  
            printf("%d",i+1);  
    }
```

```

    }

/* Driver program to test above function */
int main()
{
    int arr[] = {7, 3, 4, 5, 5, 6, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    printTwoElements(arr, n);
    return 0;
}

```

Time Complexity: O(n)

Thanks to Manish Mishra for suggesting this method.

Method 4 (Make two equations)

Let x be the missing and y be the repeating element.

1) Get sum of all numbers.

Sum of array computed $S = n(n+1)/2 - x + y$

2) Get product of all numbers.

Product of array computed $P = 1*2*3*...*n * y / x$

3) The above two steps give us two equations, we can solve the equations and get the values of x and y.

Time Complexity: O(n)

Thanks to disappearedng for suggesting this solution.

This method can cause arithmetic overflow as we calculate product and sum of all array elements. See [this](#) for changes suggested by john to reduce the chances of overflow.

Method 5 (Use XOR)

Let x and y be the desired output elements.

Calculate XOR of all the array elements.

```
xor1 = arr[0]^arr[1]^arr[2].....arr[n-1]
```

XOR the result with all numbers from 1 to n

```
xor1 = xor1^1^2^....^n
```

In the result xor1, all elements would nullify each other except x and y. All the bits that are set in xor1 will be set in either x or y. So if we take any set bit (We have chosen the rightmost set bit in code) of xor1 and divide the elements of the array in two sets – one set of elements with same bit set and other set with same bit not set. By doing so, we will get x in one set and y in another set. Now if we do XOR of all the elements in first set, we will get x, and by doing same in other set we will get y.

```

#include <stdio.h>
#include <stdlib.h>

/* The output of this function is stored at *x and *y */
void getTwoElements(int arr[], int n, int *x, int *y)
{
    int xor1; /* Will hold xor of all elements and numbers from 1 to n */
    int set_bit_no; /* Will have only single set bit of xor1 */
    int i;
    *x = 0;
    *y = 0;

    xor1 = arr[0];

    /* Get the xor of all array elements elements */
    for(i = 1; i < n; i++)
        xor1 = xor1^arr[i];

    /* XOR the previous result with numbers from 1 to n*/
    for(i = 1; i <= n; i++)
        xor1 = xor1^i;

    /* Get the rightmost set bit in set_bit_no */
    set_bit_no = xor1 & ~(xor1-1);

    /* Now divide elements in two sets by comparing rightmost set
       bit of xor1 with bit at same position in each element. Also, get XORs
       of two sets. The two XORs are the output elements. */

```

```

The following two for loops serve the purpose */
for(i = 0; i < n; i++)
{
    if(arr[i] & set_bit_no)
        *x = *x ^ arr[i]; /* arr[i] belongs to first set */
    else
        *y = *y ^ arr[i]; /* arr[i] belongs to second set */
}
for(i = 1; i <= n; i++)
{
    if(i & set_bit_no)
        *x = *x ^ i; /* i belongs to first set */
    else
        *y = *y ^ i; /* i belongs to second set */
}

/* Now *x and *y hold the desired output elements */
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1, 3, 4, 5, 5, 6, 2};
    int *x = (int *)malloc(sizeof(int));
    int *y = (int *)malloc(sizeof(int));
    int n = sizeof(arr)/sizeof(arr[0]);
    getTwoElements(arr, n, x, y);
    printf(" The two elements are %d and %d", *x, *y);
    getchar();
}

```

Time Complexity: O(n)

This method doesn't cause overflow, but it doesn't tell which one occurs twice and which one is missing. We can add one more step that checks which one is missing and which one is repeating. This can be easily done in O(n) time.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Searching
XOR

How to determine if a binary tree is height-balanced?

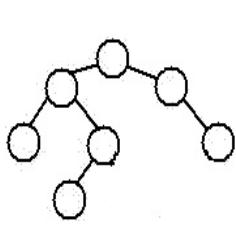
A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of "much farther" and different amounts of work to keep them balanced.

Consider a height-balancing scheme where following conditions should be checked to determine if a binary tree is balanced.

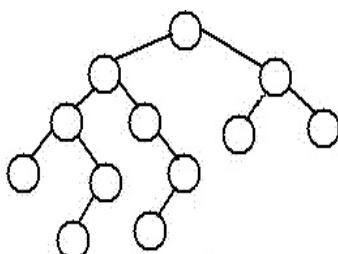
An empty tree is height-balanced. A non-empty binary tree T is balanced if:

- 1) Left subtree of T is balanced
- 2) Right subtree of T is balanced
- 3) The difference between heights of left subtree and right subtree is not more than 1.

The above height-balancing scheme is used in AVL trees. The diagram below shows two trees, one of them is height-balanced and other is not. The second tree is not height-balanced because height of left subtree is 2 more than height of right subtree.



A height-balanced Tree



Not a height-balanced tree

To check if a tree is height-balanced, get the height of left and right subtrees. Return true if difference between heights is not more than 1 and

left and right subtrees are balanced, otherwise return false.

C

```
/* C program to check if a tree is height-balanced or not */
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Returns the height of a binary tree */
int height(struct node* node);

/* Returns true if binary tree with root as root is height-balanced */
bool isBalanced(struct node *root)
{
    int lh; /* for height of left subtree */
    int rh; /* for height of right subtree */

    /* If tree is empty then return true */
    if(root == NULL)
        return 1;

    /* Get the height of left and right sub trees */
    lh = height(root->left);
    rh = height(root->right);

    if( abs(lh-rh) <= 1 &&
        isBalanced(root->left) &&
        isBalanced(root->right))
        return 1;

    /* If we reach here then tree is not height-balanced */
    return 0;
}

/* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION */

/* returns maximum of two integers */
int max(int a, int b)
{
    return (a >= b)? a: b;
}

/* The function Compute the "height" of a tree. Height is the
   number of nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    /* base case tree is empty */
    if(node == NULL)
        return 0;

    /* If tree is not empty then height = 1 + max of left
       height and right heights */
    return 1 + max(height(node->left), height(node->right));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
```

```

node->left = NULL;
node->right = NULL;

return(node);
}

int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->left->left = newNode(8);

    if(isBalanced(root))
        printf("Tree is balanced");
    else
        printf("Tree is not balanced");

    getchar();
    return 0;
}

```

Java

```

/* Java program to determine if binary tree is
height balanced or not */

/* A binary tree node has data, pointer to left child,
and a pointer to right child */
class Node
{
    int data;
    Node left, right;
    Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Returns true if binary tree with root as root is height-balanced */
    boolean isBalanced(Node node)
    {
        int lh; /* for height of left subtree */

        int rh; /* for height of right subtree */

        /* If tree is empty then return true */
        if (node == null)
            return true;

        /* Get the height of left and right sub trees */
        lh = height(node.left);
        rh = height(node.right);

        if (Math.abs(lh - rh) <= 1
            && isBalanced(node.left)
            && isBalanced(node.right))
            return true;

        /* If we reach here then tree is not height-balanced */
        return false;
    }

    /* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION */
    /* The function Compute the "height" of a tree. Height is the
       number of nodes along the longest path from the root node

```

```

    down to the farthest leaf node.*/
int height(Node node)
{
    /* base case tree is empty */
    if (node == null)
        return 0;

    /* If tree is not empty then height = 1 + max of left
    height and right heights */
    return 1 + Math.max(height(node.left), height(node.right));
}

public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.left.left.left = new Node(8);

    if(tree.isBalanced(tree.root))
        System.out.println("Tree is balanced");
    else
        System.out.println("Tree is not balanced");
}
}

```

// This code has been contributed by Mayank Jaiswal(mayank_24)

Output:

Tree is not balanced

Time Complexity: O(n^2) Worst case occurs in case of skewed tree.

Optimized implementation: Above implementation can be optimized by calculating the height in the same recursion rather than calling a height() function separately. Thanks to Amar for suggesting this optimized version. This optimization reduces time complexity to O(n).

C

```

/* program to check if a tree is height-balanced or not */
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* The function returns true if root is balanced else false
   The second parameter is to store the height of tree.
   Initially, we need to pass a pointer to a location with value
   as 0. We can also write a wrapper over this function */
bool isBalanced(struct node *root, int* height)
{
    /* lh --> Height of left subtree
       rh --> Height of right subtree */
    int lh = 0, rh = 0;

    /* l will be true if left subtree is balanced
       and r will be true if right subtree is balanced */
    int l = 0, r = 0;

    if(root == NULL)

```

```

{
    *height = 0;
    return 1;
}

/* Get the heights of left and right subtrees in lh and rh
   And store the returned values in l and r */
l = isBalanced(root->left, &lh);
r = isBalanced(root->right,&rh);

/* Height of current node is max of heights of left and
   right subtrees plus 1*/
*height = (lh > rh? lh: rh) + 1;

/* If difference between heights of left and right
   subtrees is more than 2 then this node is not balanced
   so return 0 */
if((lh - rh >= 2) || (rh - lh >= 2))
    return 0;

/* If this node is balanced and left and right subtrees
   are balanced then return true */
else return l&&r;
}

/* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION */

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main()
{
    int height = 0;

    /* Constructed binary tree is
       1
      / \
     2   3
    / \ /
   4   5 6
  /
 7
 */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->left->left->left = newNode(7);

    if(isBalanced(root, &height))
        printf("Tree is balanced");
    else
        printf("Tree is not balanced");

    getchar();
    return 0;
}

```

Java

```

/* Java program to determine if binary tree is
height balanced or not */

/* A binary tree node has data, pointer to left child,
and a pointer to right child */
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

// A wrapper class used to modify height across
// recursive calls.
class Height
{
    int height = 0;
}

class BinaryTree {

    Node root;

    /* Returns true if binary tree with root as root is height-balanced */
    boolean isBalanced(Node root, Height height)
    {
        /* If tree is empty then return true */
        if (root == null)
        {
            height.height = 0;
            return true;
        }

        /* Get heights of left and right sub trees */
        Height lheight = new Height(), rheight = new Height();
        boolean l = isBalanced(root.left, lheight);
        boolean r = isBalanced(root.right, rheight);
        int lh = lheight.height, rh = rheight.height;

        /* Height of current node is max of heights of
           left and right subtrees plus 1*/
        height.height = (lh > rh? lh: rh) + 1;

        /* If difference between heights of left and right
           subtrees is more than 2 then this node is not balanced
           so return 0 */
        if ((lh - rh >= 2) ||
            (rh - lh >= 2))
            return false;

        /* If this node is balanced and left and right subtrees
           are balanced then return true */
        else return l && r;
    }

    /* The function Compute the "height" of a tree. Height is the
       number of nodes along the longest path from the root node
       down to the farthest leaf node.*/
    int height(Node node)
    {
        /* base case tree is empty */
        if (node == null)
            return 0;

        /* If tree is not empty then height = 1 + max of left
           height and right heights */
        return 1 + Math.max(height(node.left), height(node.right));
    }

    public static void main(String args[])
}

```

```

{
    Height height = new Height();

    /* Constructed binary tree is
        1
        / \
        2   3
        / \ /
        4   5 6
        /
        7   */

    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.right = new Node(6);
    tree.root.left.left.left = new Node(7);

    if (tree.isBalanced(tree.root, height))
        System.out.println("Tree is balanced");
    else
        System.out.println("Tree is not balanced");
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)

```

Time Complexity: O(n)

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Trees
Self-Balancing-BST

Program to validate an IP address

Write a program to Validate an IPv4 Address.

According to Wikipedia, [IPv4 addresses](#) are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255, separated by dots, e.g., 172.16.254.1

Following are steps to check whether a given string is valid IPv4 address or not:

step 1) Parse string with “.” as delimiter using “[strtok\(\)](#)” function.

e.g. `ptr = strtok(str, DELIM);`

step 2)

-a) If ptr contains any character which is not digit then return 0
-b) Convert “ptr” to decimal number say ‘NUM’
-c) If NUM is not in range of 0-255 return 0
-d) If NUM is in range of 0-255 and ptr is non-NULL increment “dot_counter” by 1
-e) if ptr is NULL goto step 3 else goto step 1

step 3) if dot_counter != 3 return 0 else return 1.

```

// Program to check if a given string is valid IPv4 address or not
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define DELIM "."

/* return 1 if string contain only digits, else return 0 */
int valid_digit(char *ip_str)
{
    while (*ip_str) {
        if (*ip_str >= '0' && *ip_str <= '9')
            ++ip_str;
    }
}

```

```

        else
            return 0;
    }
    return 1;
}

/* return 1 if IP string is valid, else return 0 */
int is_valid_ip(char *ip_str)
{
    int i, num, dots = 0;
    char *ptr;

    if (ip_str == NULL)
        return 0;

    // See following link for strtok()
    // http://pubs.opengroup.org/onlinepubs/009695399/functions/strtok_r.html
    ptr = strtok(ip_str, DELIM);

    if (ptr == NULL)
        return 0;

    while (ptr) {

        /* after parsing string, it must contain only digits */
        if (!valid_digit(ptr))
            return 0;

        num = atoi(ptr);

        /* check for valid IP */
        if (num >= 0 && num <= 255) {
            /* parse remaining string */
            ptr = strtok(NULL, DELIM);
            if (ptr != NULL)
                ++dots;
        } else
            return 0;
    }

    /* valid IP string must contain 3 dots */
    if (dots != 3)
        return 0;
    return 1;
}

// Driver program to test above functions
int main()
{
    char ip1[] = "128.0.0.1";
    char ip2[] = "125.16.100.1";
    char ip3[] = "125.512.100.1";
    char ip4[] = "125.512.100.abc";
    is_valid_ip(ip1)? printf("Valid\n"): printf("Not valid\n");
    is_valid_ip(ip2)? printf("Valid\n"): printf("Not valid\n");
    is_valid_ip(ip3)? printf("Valid\n"): printf("Not valid\n");
    is_valid_ip(ip4)? printf("Valid\n"): printf("Not valid\n");
    return 0;
}

```

Output:

Valid
Valid
Not valid
Not valid

This article is compiled by **Narendra Kangalkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Two nodes of a BST are swapped, correct the BST

Two of the nodes of a Binary Search Tree (BST) are swapped. Fix (or correct) the BST.

Input Tree:

```
10
 / \
5   8
 / \
2   20
```

In the above tree, nodes 20 and 8 must be swapped to fix the tree.

Following is the output tree

```
10
 / \
5   20
 / \
2   8
```

The inorder traversal of a BST produces a sorted array. So a **simple method** is to store inorder traversal of the input tree in an auxiliary array. Sort the auxiliary array. Finally, insert the auxiliary array elements back to the BST, keeping the structure of the BST same. Time complexity of this method is $O(n\log n)$ and auxiliary space needed is $O(n)$.

We can solve this in $O(n)$ time and with a single traversal of the given BST. Since inorder traversal of BST is always a sorted array, the problem can be reduced to a problem where two elements of a sorted array are swapped. There are two cases that we need to handle:

1. The swapped nodes are not adjacent in the inorder traversal of the BST.

For example, Nodes 5 and 25 are swapped in {3 5 7 8 10 15 20 25}.

The inorder traversal of the given tree is 3 25 7 8 10 15 20 5

If we observe carefully, during inorder traversal, we find node 7 is smaller than the previous visited node 25. Here save the context of node 25 (previous node). Again, we find that node 5 is smaller than the previous node 20. This time, we save the context of node 5 (current node). Finally swap the two node's values.

2. The swapped nodes are adjacent in the inorder traversal of BST.

For example, Nodes 7 and 8 are swapped in {3 5 7 8 10 15 20 25}.

The inorder traversal of the given tree is 3 5 8 7 10 15 20 25

Unlike case #1, here only one point exists where a node value is smaller than previous node value. e.g. node 7 is smaller than node 8.

How to Solve? We will maintain three pointers, *first*, *middle* and *last*. When we find the first point where current node value is smaller than previous node value, we update the *first* with the previous node & *middle* with the current node. When we find the second point where current node value is smaller than previous node value, we update the *last* with the current node. In case #2, we will never find the second point. So, *last* pointer will not be updated. After processing, if the *last* node value is null, then two swapped nodes of BST are adjacent.

Following is C implementation of the given code.

```
// Two nodes in the BST's swapped, correct the BST.
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node *left, *right;
};

// A utility function to swap two integers
void swap( int* a, int* b )
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node *)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Function to correct the BST after two nodes are swapped
void correctBST( struct node** root )
{
    if( *root == NULL )
        return;

    struct node* first = NULL;
    struct node* middle = NULL;
    struct node* last = NULL;

    struct node* prev = NULL;
    struct node* curr = *root;
    struct node* next = curr->right;

    while( curr != NULL )
    {
        if( curr->data < prev->data )
        {
            if( first == NULL )
                first = prev;
            if( middle == NULL )
                middle = curr;
            if( last == NULL )
                last = curr;
        }

        prev = curr;
        curr = next;
        if( curr != NULL )
            next = curr->right;
    }

    if( first == middle )
    {
        if( last == NULL )
            return;
        else
            swap( &last->data, &curr->data );
    }
    else
    {
        if( first == curr )
            swap( &first->data, &curr->data );
        else
            swap( &middle->data, &curr->data );
    }
}
```

```

node->data = data;
node->left = NULL;
node->right = NULL;
return(node);
}

// This function does inorder traversal to find out the two swapped nodes.
// It sets three pointers, first, middle and last. If the swapped nodes are
// adjacent to each other, then first and middle contain the resultant nodes
// Else, first and last contain the resultant nodes
void correctBSTUtil( struct node* root, struct node** first,
                     struct node** middle, struct node** last,
                     struct node** prev )
{
    if( root )
    {
        // Recur for the left subtree
        correctBSTUtil( root->left, first, middle, last, prev );

        // If this node is smaller than the previous node, it's violating
        // the BST rule.
        if (*prev && root->data < (*prev)->data)
        {
            // If this is first violation, mark these two nodes as
            // 'first' and 'middle'
            if ( !first )
            {
                *first = *prev;
                *middle = root;
            }

            // If this is second violation, mark this node as last
            else
                *last = root;
        }

        // Mark this node as previous
        *prev = root;

        // Recur for the right subtree
        correctBSTUtil( root->right, first, middle, last, prev );
    }
}

// A function to fix a given BST where two nodes are swapped. This
// function uses correctBSTUtil() to find out two nodes and swaps the
// nodes to fix the BST
void correctBST( struct node* root )
{
    // Initialize pointers needed for correctBSTUtil()
    struct node *first, *middle, *last, *prev;
    first = middle = last = prev = NULL;

    // Set the pointers to find out two nodes
    correctBSTUtil( root, &first, &middle, &last, &prev );

    // Fix (or correct) the tree
    if( first && last )
        swap( &(first->data), &(last->data) );
    else iff( first && middle ) // Adjacent nodes swapped
        swap( &(first->data), &(middle->data) );

    // else nodes have not been swapped, passed tree is really BST.
}

/* A utility function to print Inorder traversal */
void printlnorder(struct node* node)
{
    if (node == NULL)
        return;
    printlnorder(node->left);
    printf("%d ", node->data);
    printlnorder(node->right);
}

```

```

/* Driver program to test above functions*/
int main()
{
    /*      6
           / \
          10  2
         / \ /
        1  3 7 12
    10 and 2 are swapped
*/

struct node *root = newNode(6);
root->left    = newNode(10);
root->right   = newNode(2);
root->left->left = newNode(1);
root->left->right= newNode(3);
root->right->right = newNode(12);
root->right->left = newNode(7);

printf("Inorder Traversal of the original tree \n");
printInorder(root);

correctBST(root);

printf("\nInorder Traversal of the fixed tree \n");
printInorder(root);

return 0;
}

```

Output:

```

Inorder Traversal of the original tree
1 10 3 6 7 2 12
Inorder Traversal of the fixed tree
1 2 3 6 7 10 12

```

Time Complexity: O(n)

See [this](#) for different test cases of the above code.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

[Binary Search Tree](#)

Draw a circle without floating point arithmetic

Given a radius of a circle, draw the circle without using floating point arithmetic.

Following program uses a simple concept. Let the radius of the circle be r. Consider a square of size $(2r+1) \times (2r+1)$ around the circle to be drawn. Now walk through every point inside the square. For every point (x, y) , if (x, y) lies inside the circle ($x^2 + y^2 \leq r^2$)

```

#include <stdio.h>
void drawCircle(int r) { // Consider a rectangle of size N*N int N = 2*r+1; int x, y; // Coordinates inside the rectangle // Draw a square of size N*N. for (int i = 0; i < N; i++) { for (int j = 0; j < N; j++) { // Start from the left most corner point x = i-r; y = j-r; // If this point is inside the circle, print it if (x*x + y*y <= r*r+1 ) printf("."); else // If outside the circle, print space printf(" "); printf(" "); } printf("\n"); } } // Driver Program to test above function int main() { drawCircle(8); return 0; }

```

Output:



Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Articles



December 27, 2016

Position: Software Engineer – 2(Big Data Team, Bangalore) I applied through referral. Got a call within a week. Round-1(Skype-Bangalore) (~45Mins) The interviewer directly asked me to explain about my current project. 5-6 questions from there. My project was based on cloud computing so, He asked me some basic questions about cloud computing. About three layers(IAAS, PAAS,... [Read More »](#)

Experienced
Interview Experiences
Microsoft

Top topics for Interview Preparation for Software Developer

Most Important ones

- A step by step guide for placement preparation Set 1 – Basic
- A step by step guide for placement preparation Set 2- Advanced
- Top 10 algorithms in Interview Questions
- Top 25 Interview Questions
- Commonly asked Interview Questions for Front End Developers

Competitive Programming Contests

- How to begin with Competitive Programming
- Top 10 Algorithms and Data Structures for Competitive Programming
- How to prepare for Google Asia Pacific University (APAC) Test ?

Subject Wise

- 10 Most asked Questions from Java Programmers
- Commonly Asked C Programming Interview Questions | Set 1
- Commonly Asked C Programming Interview Questions | Set 2
- Commonly Asked Java Programming Interview Questions | Set 1
- Commonly Asked OOP Interview Questions
- Commonly Asked C++ Interview Questions
- Commonly asked DBMS interview questions | Set 1
- Commonly asked DBMS interview questions | Set 2
- Commonly Asked Operating Systems Interview Questions | Set 1
- Commonly Asked Data Structure Interview Questions.
- Commonly Asked Algorithm Interview Questions
- Commonly asked Computer Networks Interview Questions

Company Wise

- Amazon's most asked interview questions Set 1 Set 2
- Microsoft's most asked interview questions Set 1 Set 2
- Accenture's most asked Interview Questions

Prepare Yourself

- 10 mistakes people tend to do in an Interview
- 6 Tips to Prepare Behavioural Interview Questions
- Top 5 Common Mistakes in Technical On-site Interviews

Company Wise Coding Practice Topic Wise Coding Practice

Last Minute Notes – GATE 2017

Top 5 Topics for Each Section of GATE CS Syllabus

GATE CS MOCK 2017

Previous year papers GATE CS, solutions and explanations year-wise and topic-wise.

This page contains GATE CS Preparation Notes / Tutorials on Mathematics, Digital Logic, Computer Organization and Architecture, Programming and Data Structures, Algorithms, Theory of Computation, Compiler Design, Operating Systems, DBMS (Database Management Systems), and Computer Networks listed according to the GATE CS 2017 official Syllabus.

GATE 2017 SYLLABUS	LEARN	PRACTICE	EXTERNAL RESOURCES
Section1: Engineering Mathematics <u>Discrete Mathematics:</u> Propositional and first order logic. Sets, relations, functions, partial orders and lattices. Groups. <u>Graphs:</u> connectivity, matching, coloring. <u>Combinatorics:</u> counting, recurrence relations, generating functions. <u>Linear Algebra:</u> Matrices, determinants, system of linear equations, eigenvalues and eigenvectors, LU decomposition. <u>Calculus:</u> Limits, continuity and differentiability. Maxima and minima. Mean value theorem. Integration. <u>Probability:</u> Random variables. Uniform, normal, exponential, poisson and binomial distributions. Mean, median, mode and standard deviation. Conditional probability and Bayes theorem.	Discrete Mathematics: Proposition Logic Introduction Propositional Equivalence Predicates and Quantifiers Set Theory Introduction Set Theory Set Operations Relations and their types Relations and their representations Groups (New) Combinatorics: Recurrence relations Pigeonhole Principle Linear Algebra: Matrix Introduction Eigen Values and Eigen Vectors L U Decomposition Calculus: Lagrange's Mean Value Theorem Mean Value Theorem Rolle's Theorem Probability: Random Variables Mean Variance And Standard Deviation Conditional Probability Bayes's formula for Conditional Probability	Set Theory & Algebra Linear Algebra Numerical Methods and Calculus Graph Theory Combinatorics Propositional and First Order Logic	Set theory,Algebra,Mathematical Logic Discrete Mathematics-NPTEL Video Lectures Book PDF-Schaum's Graph Theory Video Lectures-IISc Bangalore Lecture Notes-MIT
Section 2: Digital Logic Boolean algebra. Combinational and sequential circuits. Minimization. Number representations and computer arithmetic (fixed and floating point).	Flip-flop types and their Conversion Half Adder Half Subtractor K-Map Counters Synchronous Sequential Circuits (New) Number System and base conversions (New) Floating Point Representation (New) Last Minute Notes	Digital Logic & Number representation(28)	Video Lectures-NPTEL Notes-Number System-Swarthmore
Section 3: Computer Organization and Architecture Machine instructions and addressing modes. ALU, data-path and control unit.	Machine Instructions Addressing Modes Computer Arithmetic Set – 1 Computer Arithmetic Set – 2 Pipelining Set 1 (Execution, Stages	Computer Organization and Architecture(33)	Book PDF- Carl Hamacher Book PDF-Morris Mano

<p>Instruction pipelining.</p> <p><u>Memory hierarchy</u>: cache, main memory and secondary storage; I/O interface (interrupt and DMA mode).</p>	<p>(and Throughput)</p> <p>Pipelining Set 2 (Dependencies and Data Hazard)</p> <p>Pipelining Set 3 (Types and Stalling)</p> <p>Memory hierarchy:</p> <p>Cache Memory</p> <p>Cache Organization Introduction</p> <p>I/O Interface (Interrupt and DMA Mode)</p>		
<p>Section 4: Programming and Data Structures</p> <p>Programming in C.</p> <p>Recursion.</p> <p>Arrays, stacks, queues, linked lists, trees, binary search trees, binary heaps, graphs.</p>	<p>C Programming</p> <p>Recursion</p> <p>Recursive functions</p> <p>Tail Recursion</p> <p>Arrays</p> <p>Stack</p> <p>Queue</p> <p>Linked List</p> <p>Binary Tree</p> <p>Binary Search Tree</p> <p>Binary Heap</p> <p>Graph</p>	<p>C Language</p> <p>Recursion</p> <p>Linked List</p> <p>Stack</p> <p>Queue</p> <p>Binary Trees</p> <p>Tree Traversals</p> <p>Binary Search Trees</p> <p>Balanced Binary Search</p> <p>Trees</p> <p>Array</p> <p>Heap</p> <p>Graph</p> <p>Graph Traversals</p> <p>B and B+ Trees</p> <p>Misc</p> <p>Data Structures and Algorithm</p>	<p>Video lectures-IITD</p> <p>Book- Introduction to Algorithms by Cormen, Thomas H.</p>
<p>Section 5: Algorithms</p> <p>Searching, sorting, hashing.</p> <p>Asymptotic worst case time and space complexity.</p> <p><u>Algorithm design techniques</u>: greedy, dynamic programming and divide-and-conquer.</p> <p>Graph search, minimum spanning trees, shortest paths.</p>	<p>Searching and Sorting</p> <p>Hashing</p> <p>Analysis of Algorithms</p> <p>Greedy Algorithms</p> <p>Dynamic Programming</p> <p>Divide and Conquer</p> <p>Graph Algorithms (Search Algorithms)</p> <p>Minimum Spanning Tree:</p> <p>Prims Minimum Spanning tree</p> <p>Prims Minimum Spanning Tree for adjacency list representation</p> <p>Kruskals Minimum Spanning tree</p> <p>Shortest Paths:</p> <p>Dijkstras Shortest Path Algorithm</p> <p>Dijkstra's Algorithm for Adjacency List Representation</p> <p>Bellman–Ford Algorithm</p> <p>Floyd Warshall Algorithm</p> <p>Shortest Path in Directed Acyclic Graph</p> <p>Shortest path with exactly k edges in a directed and weighted graph</p>	<p>Searching</p> <p>Sorting</p> <p>Hash</p> <p>Analysis of Algorithms</p> <p>Analysis of Algorithms (Recurrences)</p> <p>Divide and Conquer</p> <p>Greedy Algorithms</p> <p>Dynamic Programming</p> <p>Backtracking</p> <p>Graph Shortest Paths</p> <p>Graph Minimum Spanning Tree</p> <p>NP Complete</p> <p>Misc</p> <p>Data Structures and Algorithm</p>	<p>Video lectures-IITD</p> <p>Book- Introduction to Algorithms by Cormen, Thomas H.</p>
<p>Section 6: Theory of Computation</p> <p>Regular expressions and finite automata.</p> <p>Context-free grammars and push-down automata.</p> <p>Regular and context-free languages, pumping lemma.</p> <p>Turing machines and undecidability.</p>	<p>Finite Automata Introduction</p> <p>Chomsky Hierarchy</p> <p>Pumping Lemma</p> <p>Designing Finite Automata from Regular Expression</p> <p>Regular Expressions, Regular Grammar and</p> <p>Regular Languages</p> <p>Pushdown Automata</p> <p>Closure Properties of Context Free Languages</p> <p>Conversion from NFA to DFA</p> <p>Minimization of DFA</p> <p>Mealy and Moore Machines</p> <p>Decidability</p> <p>Turing Machine</p> <p>Ambiguity in CFG and CFL</p> <p>Simplifying Context Free Grammars</p> <p>Recursive and Recursive Enumerable Languages</p> <p>Undecidability and Reducibility</p> <p>Last Minute Notes</p>	<p>Regular languages and finite automata</p> <p>Context free languages and Push-down automata</p> <p>Recursively enumerable sets and Turing machines</p> <p>Undecidability</p> <p>Automata Theory</p>	<p>Web Resource-ArsDigita University</p> <p>Sample Problems and Solutions-Loyola Univ</p>

Section 7: Compiler Design Lexical analysis, parsing, syntax-directed translation. Runtime environments. Intermediate code generation.	Lexical Analysis Introduction to Syntax Analysis Parsing Set 1 Parsing Set 2 Parsing Set 3 Syntax Directed Translation Intermediate Code Generation Runtime Environments Classification of Context Free Grammars Ambiguous Grammar Why FIRST and FOLLOW? FIRST Set in Syntax Analysis FOLLOW Set in Syntax Analysis	Lexical analysis Parsing and Syntax directed translation Code Generation and Optimization Compiler Design	Lecture Slides-Stanford Book-Aho and Ullman Dragon Book Lecture Notes Video Lectures-Stanford
Section 8: Operating System Processes, threads, inter-process communication, concurrency and synchronization. Deadlock. CPU scheduling. Memory management and virtual memory. File systems.	Process Management Introduction Process Scheduling Process scheduler Disk Scheduling Process Synchronization Introduction Process Synchronization Monitors Deadlock Introduction Deadlock Prevention And Avoidance Deadlock Detection And Recovery Memory Management Partition Allocation Method Page Replacement Algorithm User Thread VS Kernel Thread Multi threading Model Inter-Process Communication Fork System Call Paging Segmentation Banker's Algorithm Readers-Writers Problem Set 1 (Introduction and Readers Preference Solution) Difference between Priority Inversion and Priority Inheritance File Systems Virtual Memory Operating System Notes Last Minute Notes Commonly Asked Interview Question	Process Management CPU Scheduling Memory Management Input Output Systems Operating Systems	Web resource- VirginiaTech Univ. Lecture Slides- Silberschatz, Galvin, Gagne Video Lectures-IIT KGP Practice Problems and Solutions- William Stallings
Section 9: Databases ER-model. Relational model: relational algebra, tuple calculus, SQL. Integrity constraints, normal forms. File organization, indexing (e.g., B and B+ trees). Transactions and concurrency control.	Need for DBMS Relational Model and Algebra : Relational Model Introduction and Codd Rules Keys in Relational Model (Candidate, Super, Primary, Alternate and Foreign) Relational Algebra-Overview Relational Algebra-Basic Operators Relational Algebra-Extended Operators Normalization: Lossless Decomposition Dependency Preserving Decomposition Attribute Closure/Candidate Key-Functional Dependencies Database Normalization Introduction Database Normal Form Equivalence of Functional Dependencies Find the highest normal form of a relation ER Model : ER Model Minimization of ER Diagram	ER and Relational Models Database Design (Normal Forms) SQL Transactions and concurrency control Sequential files, indexing, B & B+ trees Database Management Systems	Lecture Slides-Silberschatz, Korth and Sudarshan Lecture Slides-Raghu Ramakrishnan and Johannes Gehrke Lecture Slides-Stanford DBMS course Video Lectures-IIT KGP

	Mapping from ER Model to Relational Model SQL: SQL Inner VS Outer Join Having Vs Where Clause Nested Queries in SQL Concurrency: ACID Properties Concurrency Control View equal schedule Conflict Serializability Recoverability of Schedules Misc: Indexing in Databases Set 1 How to solve Relational Algebra problems for GATE		
Section 10: Computer Networks	Network Topologies (New) LAN Technologies Network Devices IP Addressing Introduction and Classful Addressing IP Addressing Classless Addressing Network Layer Introduction Network Layer IPv4 Datagram Fragmentation and Delay Longest Prefix Matching in Routers Why DNS uses UDP not TCP Error Detection Congestion Control Leaky Bucket Algorithm (New) Stop and Wait ARQ Sliding Window Protocol Set 1 (Sender Side) Sliding Window Protocol Set 2 (Receiver Side) Difference between http and https DNS SMTP ICMP (New) Circuit Switching VS Packet Switching Basics of Wi-Fi Digital Signatures and Certificates Commonly asked Computer Networks Interview Questions Set 1	Data Link Layer Network Layer Transport Layer Misc Topics in Computer Networks Application Layer Network Security Computer Networks	Video Lectures by University of Washington. Lecture Notes-Prof. Dheeraj Sanghi, IIT Kanpur Web Resources on Computer Networks by Andrew S. Tanenbaum.

Previous year papers GATE CS, solutions and explanations year-wise and topic-wise.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above!

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.

All about GATE CS Preparation for 2017 aspirants. The page contains solutions of previous year GATE CS papers with explanations, topic wise Quizzes, notes/tutorials and important links for preparation.

GATE CS MOCK 2017

GATE CS Notes/Tutorials (According to Official GATE 2017 Syllabus)

Previous Years' questions/answers/explanation for GATE CS

- GATE-CS-2017 (Set 1)
- GATE-CS-2017 (Set 2)
- GATE-CS-2016 (Set 1)
- GATE-CS-2016 (Set 2)
- GATE-CS-2015 (Set 1)
- GATE-CS-2015 (Set 2)
- GATE-CS-2015 (Set 3)
- GATE-CS-2014-(Set-1)
- GATE-CS-2014-(Set-2)
- GATE-CS-2014-(Set-3)
- GATE CS 2013
- GATE CS 2012
- GATE CS 2011
- GATE CS 2010
- GATE-CS-2009
- GATE CS 2008
- GATE-CS-2007
- GATE-CS-2006
- GATE-CS-2005
- GATE-CS-2004
- GATE-CS-2003
- GATE-CS-2002
- GATE-CS-2001
- GATE-CS-2000

Previous Years' questions/answers/explanation for GATE IT

- GATE-IT-2004
- GATE-IT-2006
- GATE-IT-2008
- GATE-IT-2005
- GATE-IT-2007

Topic-wise Mock Quizzes for GATE CS

Data Structures and Algorithms

Linked List
Stack
Queue
Binary Trees
Binary Search Trees
Balanced Binary Search Trees
Graph
Hash
Array
Misc
B and B+ Trees
Heap
Tree Traversals
Analysis of Algorithms
Sorting
Divide and Conquer
Greedy Algorithms
Dynamic Programming
Backtracking
Misc
NP Complete
Searching
Analysis of Algorithms (Recurrences)
Recursion
Bit Algorithms
Graph Traversals
Graph Shortest Paths
Graph Minimum Spanning Tree
Data Structures and Algorithm
C Language

Operating Systems
Operating System Notes
Process Management
CPU Scheduling
Memory Management
Input Output Systems

DBMS

ER and Relational Models
Database Design (Normal Forms)
SQL
Transactions and concurrency control
Sequential files, indexing, B & B+ trees
Database Management Systems

Compiler Design

Lexical analysis
Parsing and Syntax directed translation
Code Generation and Optimization
Compiler Design

Computer Networks

Data Link Layer
Network Layer
Transport Layer
Misc Topics in Computer Networks
Application Layer
Network Security
Computer Networks

Theory of Computation

Regular languages and finite automata
Context free languages and Push-down automata
Recursively enumerable sets and Turing machines
Undecidability
Automata Theory

Aptitude

Probability
English
General Aptitude

Engineering Mathematics

[Operating Systems](#)

Computer Organization and Architecture

[Digital Logic & Number representation\(28\)](#)

[Computer Organization and Architecture\(33\)](#)

[Sample GATE Mock Test](#)

[Set Theory & Algebra](#)

[Linear Algebra](#)

[Numerical Methods and Calculus](#)

[Graph Theory](#)

[Combinatorics](#)

[Propositional and First Order Logic](#)

Important Links:

[Solutions of GATE CS 2016 Mock Test](#)

[Top 5 Topics for for Section of GATE CS Syllabus](#)

[How to prepare in Last 10 days for GATE?](#)

[GATE CS Topic wise External Reference Links](#)

[Previous Year GATE Official Question Papers](#)

[Last Minute Notes](#)

[GATE CS 2016 Official Papers](#)

[GATE CS 2017 Dates](#)

[GATE CS 2017 Syllabus](#)

Please write comments if you find anything incorrect or wish to share more information for GATE CS preparation.

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.

[Load Comments](#)

Accenture's most asked Interview Questions

- 1) What is/are your favorite subject(s)? There may be many questions on the subject told.
- 2) Differences between C and C++.
- 3) What is include in a C program?
- 4) What is Dynamic Memory Allocation, example?
- 5) Differences between C/C++ and Java?
- 6) Simple programs like Bubble Sort, sum of a simple series, etc.
- 7) What is OOP?
- 8) What are encapsulation, Inheritance, polymorphism and abstraction?
- 9) Explain Runtime Polymorphism
- 10) What is BCNF?
- 11) What are inner and outer joins? Examples of both.
- 12) Questions on keys in DBMS like primary key, super key, difference between primary key and unique?
- 13) OSI Layers?
- 14) Why Accenture?
- 16) Are you ready to work in night shifts?
- 17) About final year project?
- 18) Your strengths and weaknesses?
- 19) Do you know a language other than C, C++ and Java?
- 20) Background questions like AIEEE rank, year gap, family members, etc

Also refer [Tips for a Successful Interview with Accenture](#)

Please write comments if any common question is missed.

If you like GeeksQuiz and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Articles

Commonly Asked OOP Interview Questions | Set 1

What is Object Oriented Programming?

Object Oriented Programming (OOP) is a programming paradigm where the complete software operates as a bunch of objects talking to each other. An object is a collection of data and methods that operate on its data.

Why OOP?

The main advantage of OOP is better manageable code that covers following.

- 1) The overall understanding of the software is increased as the distance between the language spoken by developers and that spoken by users.
- 2) Object orientation eases maintenance by the use of encapsulation. One can easily change the underlying representation by keeping the methods same.

OOP paradigm is mainly useful for relatively big software. See [this](#) for a complete example that shows advantages of OOP over procedural programming.

What are main features of OOP?

Encapsulation

Polymorphism

Inheritance

What is encapsulation?

Encapsulation is referred to one of the following two notions.

- 1) Data hiding: A language feature to restrict access to members of an object. For example, private and protected members in C++.
- 2) Bundling of data and methods together: Data and methods that operate on that data are bundled together.

What is Polymorphism? How is it supported by C++?

Polymorphism means that some code or operations or objects behave differently in different contexts. In C++, following features support polymorphism.

Compile Time Polymorphism: Compile time polymorphism means compiler knows which function should be called when a polymorphic call is made. C++ supports compiler time polymorphism by supporting features like templates, function overloading and default arguments.

Run Time Polymorphism: Run time polymorphism is supported by virtual functions. The idea is, [virtual functions](#) are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.

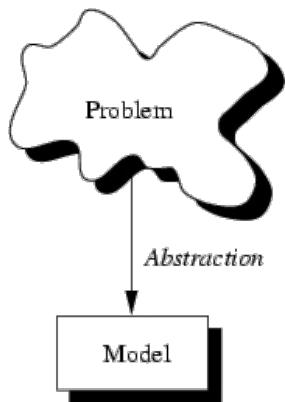
What is Inheritance? What is the purpose?

The idea of inheritance is simple, a class is based on another class and uses data and implementation of the other class.

The purpose of inheritance is Code Reuse.

What is Abstraction?

The first thing with which one is confronted when writing programs is the problem. Typically we are confronted with "real-life" problems and we want to make life easier by providing a program for the problem. However, real-life problems are nebulous and the first thing we have to do is to try to understand the problem to separate necessary from unnecessary details: We try to obtain our own abstract view, or model, of the problem. This process of modeling is called abstraction (Source <http://gd.tuwien.ac.at/languages/c/c++oop-pmueller/node4.html#SECTION0041000000000000000>)



See [the source](#) for a complete example and more details of abstraction.

References:

<http://gd.tuwien.ac.at/languages/c/c++oop-pmueller/tutorial.html>

You may like to see following:

[Commonly Asked C Programming Interview Questions | Set 1](#)
[Commonly Asked C Programming Interview Questions | Set 2](#)
[Amazon's most asked interview questions](#)
[Microsoft's most asked interview questions](#)
[Accenture's most asked Interview Questions](#)
[Commonly Asked OOP Interview Questions](#)
[Commonly Asked C++ Interview Questions](#)
[Commonly asked DBMS interview questions | Set 1](#)
[Commonly asked DBMS interview questions | Set 2](#)
[Commonly Asked Operating Systems Interview Questions | Set 1](#)
[Commonly Asked Data Structure Interview Questions.](#)
[Commonly Asked Algorithm Interview Questions](#)
[Commonly asked Computer Networks Interview Questions](#)
[Top 10 algorithms in Interview Questions](#)

We will soon be covering more OOP Questions. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: C++

Commonly Asked C Programming Interview Questions | Set 1

What is the difference between declaration and definition of a variable/function

Ans: Declaration of a variable/function simply declares that the variable/function exists somewhere in the program but the memory is not allocated for them. But the declaration of a variable/function serves an important role. And that is the type of the variable/function. Therefore, when a variable is declared, the program knows the data type of that variable. In case of function declaration, the program knows what are the arguments to that functions, their data types, the order of arguments and the return type of the function. So that's all about declaration. Coming to the definition, when we define a variable/function, apart from the role of declaration, it also allocates memory for that variable/function. Therefore, we can think of definition as a super set of declaration. (or declaration as a subset of definition). From this explanation, it should be obvious that a variable/function can be declared any number of times but it can be defined only once. (Remember the basic principle that you can't have two locations of the same variable/function).

```

// This is only declaration. y is not allocated memory by this statement
extern int y;

// This is both declaration and definition, memory to x is allocated by this statement.
int x;

```

What are different storage class specifiers in C?

Ans: auto, register, static, extern

What is scope of a variable? How are variables scoped in C?

Ans: Scope of a variable is the part of the program where the variable may directly be accessible. In C, all identifiers are lexically (or statically) scoped. See [this](#) for more details.

How will you print “Hello World” without semicolon?

Ans:

```
int main(void)
{
    if (printf("Hello World")) ;
}
```

See [print “Geeks for Geeks” without using a semicolon for answer.](#)

When should we use pointers in a C program?

1. To get address of a variable
2. *For achieving pass by reference in C:* Pointers allow different functions to share and modify their local variables.
3. *To pass large structures* so that complete copy of the structure can be avoided.
- C
4. *To implement “linked” data structures* like linked lists and binary trees.

What is NULL pointer?

Ans: NULL is used to indicate that the pointer doesn't point to a valid location. Ideally, we should initialize pointers as NULL if we don't know their value at the time of declaration. Also, we should make a pointer NULL when memory pointed by it is deallocated in the middle of a program.

What is Dangling pointer?

Ans: Dangling Pointer is a pointer that doesn't point to a valid memory location. Dangling pointers arise when an object is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory. Following are examples.

```
// EXAMPLE 1
int *ptr = (int *)malloc(sizeof(int));
.....
.....
free(ptr);

// ptr is a dangling pointer now and operations like following are invalid
*ptr = 10; // or printf("%d", *ptr);
```

```
// EXAMPLE 2
int *ptr = NULL
{
    int x = 10;
    ptr = &x;
}
// x goes out of scope and memory allocated to x is free now.
// So ptr is a dangling pointer now.
```

What is memory leak? Why it should be avoided

Ans: Memory leak occurs when programmers create a memory in heap and forget to delete it. Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

```
/* Function with memory leak */
#include <stdlib.h>

void f()
{
    int *ptr = (int *) malloc(sizeof(int));

    /* Do some work */

    return; /* Return without freeing ptr*/
}
```

What are local static variables? What is their use?

Ans:A local static variable is a variable whose lifetime doesn't end with a function call where it is declared. It extends for the lifetime of complete program. All calls to the function share the same copy of local static variables. Static variables can be used to count the number of times a function is called. Also, static variables get the default value as 0. For example, the following program prints “0 1”

```

#include <stdio.h>
void fun()
{
    // static variables get the default value as 0.
    static int x;
    printf("%d ", x);
    x = x + 1;
}

int main()
{
    fun();
    fun();
    return 0;
}
// Output: 0 1

```

What are static functions? What is their use?

Ans:In C, functions are global by default. The “static” keyword before a function name makes it static. Unlike global functions in C, access to static functions is restricted to the file where they are declared. Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files. See [this](#) for examples and more details.

- [Commonly Asked C Programming Interview Questions | Set 2](#)
- [Practices Quizzes on C](#)
- [C articles](#)

You may also like:

[Commonly Asked C Programming Interview Questions | Set 2](#)
[Commonly Asked Java Programming Interview Questions | Set 1](#)
[Amazon's most asked interview questions](#)
[Microsoft's most asked interview questions](#)
[Accenture's most asked Interview Questions](#)
[Commonly Asked OOP Interview Questions](#)
[Commonly Asked C++ Interview Questions](#)
[Commonly asked DBMS interview questions | Set 1](#)
[Commonly asked DBMS interview questions | Set 2](#)
[Commonly Asked Operating Systems Interview Questions | Set 1](#)
[Commonly Asked Data Structure Interview Questions.](#)
[Commonly Asked Algorithm Interview Questions](#)
[Commonly asked Computer Networks Interview Questions](#)
[Top 10 algorithms in Interview Questions](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: C

Commonly Asked C Programming Interview Questions | Set 2

This post is second set of [Commonly Asked C Programming Interview Questions | Set 1](#)

What are main characteristics of C language?

C is a procedural language. The main features of C language include low-level access to memory, simple set of keywords, and clean style. These features make it suitable for system programming like operating system or compiler development.

What is difference between i++ and ++i?

- 1) The expression ‘i++’ returns the old value and then increments i. The expression ++i increments the value and returns new value.
- 2) Precedence of postfix ++ is higher than that of prefix ++.
- 3) Associativity of postfix ++ is left to right and associativity of prefix ++ is right to left.
- 4) In C++, ++i can be used as l-value, but i++ cannot be. In C, they both cannot be used as l-value.

See [Difference between ++*p, *p++ and *++p](#) for more details.

What is l-value?

l-value or location value refers to an expression that can be used on left side of assignment operator. For example in expression "a = 3", a is l-value and 3 is r-value.

l-values are of two types:

"nonmodifiable l-value" represent a l-value that can not be modified. const variables are "nonmodifiable l-value".

"modifiable l-value" represent a l-value that can be modified.

What is the difference between array and pointer?

See [Array vs Pointer](#)

How to write your own sizeof operator?

```
#define my_sizeof(type) (char *)(&type+1)-(char *)(&type)
```

See [Implement your own sizeof](#) for more details.

How will you print numbers from 1 to 100 without using loop?

We can use recursion for this purpose.

```
/* Prints numbers from 1 to n */
void printNos(unsigned int n)
{
    if(n > 0)
    {
        printNos(n-1);
        printf("%d ", n);
    }
}
```

What is volatile keyword?

The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

Objects declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time. See [Understanding "volatile" qualifier in C](#) for more details.

Can a variable be both const and volatile?

yes, the const means that the variable cannot be assigned a new value. The value can be changed by other code or pointer. For example the following program works fine.

```
int main(void)
{
    const volatile int local = 10;
    int *ptr = (int*) &local;
    printf("Initial value of local : %d \n", local);
    *ptr = 100;
    printf("Modified value of local: %d \n", local);
    return 0;
}
```

- Practices [Quizzes on C](#)
- [C articles](#)

We will soon be publishing more sets of commonly asked C programming questions.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: C

Commonly Asked C++ Interview Questions | Set 1

What are the differences between C and C++?

- 1) C++ is a kind of superset of C, most of C programs except few exceptions (See [this](#) and [this](#)) work in C++ as well.
- 2) C is a procedural programming language, but C++ supports both procedural and Object Oriented programming.
- 3) Since C++ supports object oriented programming, it supports features like function overloading, templates, inheritance, virtual functions, friend functions. These features are absent in C.
- 4) C++ supports exception handling at language level, in C exception handling is done in traditional if-else style.
- 5) C++ supports [references](#), C doesn't.
- 6) In C, scanf() and printf() are mainly used input/output. C++ mainly uses streams to perform input and output operations. cin is standard input stream and cout is standard output stream.

There are many more differences, above is a list of main differences.

What are the differences between references and pointers?

Both references and pointers can be used to change local variables of one function inside another function. Both of them can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain.

Despite above similarities, there are following differences between references and pointers.

References are less powerful than pointers

- 1) Once a reference is created, it cannot be later made to reference another object; it cannot be reseated. This is often done with pointers.
- 2) References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.
- 3) A reference must be initialized when declared. There is no such restriction with pointers

Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc. In Java, references don't have above restrictions, and can be used to implement all data structures. References being more powerful in Java, is the main reason Java doesn't need pointers.

References are safer and easier to use:

- 1) Safer: Since references must be initialized, wild references like wild pointers are unlikely to exist. It is still possible to have references that don't refer to a valid location (See questions 5 and 6 in the below exercise)
- 2) Easier to use: References don't need dereferencing operator to access the value. They can be used like normal variables. '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator ('.'), unlike pointers where arrow operator ('->') is needed to access members.

What are virtual functions – Write an example?

[Virtual functions](#) are used with inheritance, they are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime. Virtual keyword is used to make a function virtual.

Following things are necessary to write a C++ program with runtime polymorphism (use of virtual functions)

- 1) A base class and a derived class.
- 2) A function with same name in base class and derived class.
- 3) A pointer or reference of base class type pointing or referring to an object of derived class.

For example, in the following program bp is a pointer of type Base, but a call to bp->show() calls show() function of Derived class, because bp points to an object of Derived class.

```
#include<iostream>
using namespace std;

class Base {
public:
    virtual void show() { cout<<"In Base \n"; }
};

class Derived: public Base {
public:
    void show() { cout<<"In Derived \n"; }
};

int main(void) {
    Base *bp = new Derived;
    bp->show(); // RUN-TIME POLYMORPHISM
    return 0;
}
```

Output:

In Derived

What is this pointer?

The ['this'](#) pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. 'this' pointer is a constant pointer that holds the memory address of the current object. 'this' pointer is not available in static

member functions as static member functions can be called without any object (with class name).

Can we do “delete this”?

See <http://www.geeksforgeeks.org/delete-this-in-c/>

What are VTABLE and VPTR?

vtable is a table of function pointers. It is maintained per class.

vptr is a pointer to vtable. It is maintained per object (See this for an example).

Compiler adds additional code at two places to maintain and use vtable and vptr.

1) Code in every constructor. This code sets vptr of the object being created. This code sets vptr to point to vtable of the class.

2) Code with polymorphic function call (e.g. bp->show() in above code). Wherever a polymorphic call is made, compiler inserts code to first look for vptr using base class pointer or reference (In the above example, since pointed or referred object is of derived type, vptr of derived class is accessed). Once vptr is fetched, vtable of derived class can be accessed. Using vtable, address of derived derived class function show() is accessed and called.

You may also like:

- Practice [Quizzes](#) on C++
- C/C++ [Articles](#)

We will soon be covering more C++. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: C++

Commonly Asked Java Programming Interview Questions | Set 1

Why is Java called the ‘Platform Independent Programming Language’?

Platform independence means that execution of your program does not depend on type of operating system(it could be any : Linux, windows, Mac ..etc). So compile code only once and run it on any System (In C/C++, we need to compile the code for every machine on which we run it). Java is both compiler(javac) and interpreter(jvm) based language. Your java source code is first compiled into byte code using javac compiler. This byte code can be easily converted to equivalent machine code using JVM. JVM(Java Virtual Machine) is available in all operating systems we install. Hence, byte code generated by javac is universal and can be converted to machine code on any operating system, this is the reason why java is platform independent.

Explain Final keyword in java?

Final keyword in java is used to restrict usage of variable, class and method.

Variable: Value of Final variable is constant, you can not change it.

Method: you can't override a Final method.

Class: you can't inherit from Final class.

Refer [this](#) for details

When is the super keyword used?

super keyword is used to refer:

- immediate parent class constructor,
- immediate parent class variable,
- immediate parent class method.

Refer [this](#) for details.

What is the difference between StringBuffer and String?

String is an Immutable class, i.e. you can not modify its content once created. While StringBuffer is a mutable class, means you can change its content later. Whenever we alter content of String object, it creates a new string and refer to that, it does not modify the existing one. This is the reason that the performance with StringBuffer is better than with String.

Refer [this](#) for details.

Why multiple inheritance is not supported in java?

Java supports multiple inheritance but not through classes, it supports only through its interfaces. The reason for not supporting multiple inheritance is to avoid the conflict and complexity arises due to it and keep Java a Simple Object Oriented Language. If we recall [this](#) in C++, there is a special case of multiple inheritance (diamond problem) where you have a multiple inheritance with two classes which have methods in conflicts. So, Java developers decided to avoid such conflicts and didn't allow multiple inheritance through classes at all.

Can a top level class be private or protected?

Top level classes in java can't be private or protected, but inner classes in java can. The reason for not making a top level class as private is very obvious, because nobody can see a private class and thus they can not use it. Declaring a class as protected also doesn't make any sense. The only difference between default visibility and protected visibility is that we can use it in any package by inheriting it. Since in java there is no such concept of package inheritance, defining a class as protected is no different from default.

What is the difference between 'throw' and 'throws' in Java Exception Handling?

Following are the differences between two:

- throw keyword is used to throw Exception from any method or static block whereas throws is used to indicate that which Exception can possibly be thrown by this method
- If any method throws checked Exception, then caller can either handle this exception(using try catch block)or can re throw it by declaring another 'throws' clause in method declaration.
- throw clause can be used in any part of code where you feel a specific exception needs to be thrown to the calling method

E.g.

```
throw  
throw new Exception("You have some exception")  
throw new IOException("Connection failed!!")  
throws  
throws IOException, NullPointerException, ArithmeticException
```

What is finalize() method?

Unlike c++ , we don't need to destroy objects explicitly in Java. 'Garbage Collector' does that automatically for us. Garbage Collector checks if no references to an object exist, that object is assumed to be no longer required, and the memory occupied by the object can be freed. Sometimes an object can hold non-java resources such as file handle or database connection, then you want to make sure these resources are also released before object is destroyed. To perform such operation Java provide protected void finalize() in object class. You can override this method in your class and do the required tasks. Right before an object is freed, the java run time calls the finalize() method on that object. Refer [this](#) for more details.

Difference in Set and List interface?

Set and List both are child interface of Collection interface. There are following two main differences between them

- List can hold duplicate values but Set doesn't allow this.
- In List interface data is present in the order you inserted but in the case of Set insertion order is not preserved.

What will happen if you put System.exit(0) on try or catch block? Will finally block execute?

By Calling System.exit(0) in try or catch block, we can skip the finally block. System.exit(int) method can throw a SecurityException. If System.exit(0) exits the JVM without throwing that exception then finally block will not execute. But, if System.exit(0) does throw security exception then finally block will be executed.

- Practice [Quizzes](#) of Java
- [Java Articles](#)

This article is compiled by **Dharmesh Singh**.

You may like to see following:

[Commonly Asked C Programming Interview Questions | Set 1](#)
[Commonly Asked C Programming Interview Questions | Set 2](#)
[Amazon's most asked interview questions](#)
[Microsoft's most asked interview questions](#)
[Accenture's most asked Interview Questions](#)
[Commonly Asked OOP Interview Questions](#)
[Commonly Asked C++ Interview Questions](#)
[Commonly asked DBMS interview questions | Set 1](#)
[Commonly asked DBMS interview questions | Set 2](#)
[Commonly Asked Operating Systems Interview Questions | Set 1](#)
[Commonly Asked Data Structure Interview Questions.](#)
[Commonly Asked Algorithm Interview Questions](#)
[Commonly asked Computer Networks Interview Questions](#)
[Top 10 algorithms in Interview Questions](#)

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Java

Commonly Asked Java Programming Interview Questions | Set 2

[Frequently Asked Java Interview Questions | Set 1](#)

Can we Overload or Override static methods in java ?

- **Overriding :** Overriding is related to run-time polymorphism. A subclass (or derived class) provides a specific implementation of a method in superclass (or base class) at runtime.
- **Overloading:** Overloading is related to compile time (or static) polymorphism. This feature allows different methods to have same name, but different signatures, especially number of input parameters and type of input parameters.
- **Can we overload static methods?** The answer is '**Yes**'. We can have two or more static methods with same name, but differences in input parameters
- **Can we Override static methods in java?** We can declare static methods with same signature in subclass, but it is not considered overriding as there won't be any run-time polymorphism. Hence the answer is '**No**'. Static methods cannot be overridden because method overriding only occurs in the context of dynamic (i.e. runtime) lookup of methods. Static methods (by their name) are looked up statically (i.e. at compile-time).

[Read more](#)

Why the main method is static in java?

The method is static because otherwise there would be ambiguity: which constructor should be called? Especially if your class looks like this:

```
public class JavaClass
```

```
{
protected JavaClass(int x)
{
}
public void main(String[] args)
{
}
}
```

Should the JVM call new JavaClass(int)? What should it pass for x? If not, should the JVM instantiate JavaClass without running any constructor method? because that will special-case your entire class – sometimes you have an instance that hasn't been initialized, and you have to check for it in every method that could be called. There are just too many edge cases and ambiguities for it to make sense for the JVM to have to instantiate a class before the entry point is called. That's why main is static.

What happens if you remove static modifier from the main method?

Program compiles successfully . But at runtime throws an error "NoSuchMethodError".

What is the scope of variables in Java in following cases?

- **Member Variables** (Class Level Scope) : The member variables must be declared inside class (outside any function). They can be directly accessed anywhere in class
- **Local Variables** (Method Level Scope) : Variables declared inside a method have method level scope and can't be accessed outside the method.
- **Loop Variables** (Block Scope) : A variable declared inside pair of brackets "{" and "}" in a method has scope within the brackets only.

Read [more](#)

What is “this” keyword in java?

Within an instance method or a constructor, this is a reference to the current object — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using this.

Usage of this keyword

- Used to refer current class instance variable.
- To invoke current class constructor.
- It can be passed as an argument in the method call.
- It can be passed as argument in the constructor call.
- Used to return the current class instance.
- Used to invoke current class method (implicitly)

What is an abstract class? How abstract classes are similar or different in Java from C++?

Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods.

- Like C++, in Java, an instance of an abstract class cannot be created, we can have references of abstract class type though.
- Like C++, an abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created
- In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.
- Abstract classes can also have final methods (methods that cannot be overridden). For example, the following program compiles and runs fine.

Read [more](#)

Which class is the superclass for every class ?

Object class

Can we overload main() method?

The main method in Java is no extra-terrestrial method. Apart from the fact that main() is just like any other method & can be overloaded in a

similar manner, JVM always looks for the method signature to launch the program.

- The normal main method acts as an entry point for the JVM to start the execution of program.
- We can overload the main method in Java. But the program doesn't execute the overloaded main method when we run your program, we need to call the overloaded main method from the actual main method only.

Read [more](#)

What is object cloning?

Object cloning means to create an exact copy of the original object. If a class needs to support cloning, it must implement `java.lang.Cloneable` interface and override `clone()` method from `Object` class. Syntax of the `clone()` method is :

```
protected Object clone() throws CloneNotSupportedException
```

If the object's class doesn't implement `Cloneable` interface then it throws an exception '`CloneNotSupportedException`' .

Read [more](#)

How is inheritance in C++ different from Java?

1. In Java, all classes inherit from the `Object` class directly or indirectly. Therefore, there is always a single inheritance tree of classes in Java, and `Object` class is root of the tree.
2. In Java, members of the grandparent class are not directly accessible. See [this G-Fact](#) for more details.
3. The meaning of protected member access specifier is somewhat different in Java. In Java, protected members of a class "A" are accessible in other class "B" of same package, even if B doesn't inherit from A (they both have to be in the same package).
4. Java uses `extends` keyword for inheritance. Unlike C++, Java doesn't provide an inheritance specifier like public, protected or private. Therefore, we cannot change the protection level of members of base class in Java, if some data member is public or protected in base class then it remains public or protected in derived class. Like C++, private members of base class are not accessible in derived class. Unlike C++, in Java, we don't have to remember those rules of inheritance which are combination of base class access specifier and inheritance specifier.
5. In Java, methods are virtual by default. In C++, we explicitly use `virtual` keyword. See [this G-Fact](#) for more details.
6. Java uses a separate keyword `interface` for interfaces, and `abstract` keyword for abstract classes and abstract functions.
7. Unlike C++, Java doesn't support multiple inheritance. A class cannot inherit from more than one class. A class can implement multiple interfaces though.
8. In C++, default constructor of parent class is automatically called, but if we want to call parametrized constructor of a parent class, we must use [Initializer list](#). Like C++, default constructor of the parent class is automatically called in Java, but if we want to call parameterized constructor then we must use `super` to call the parent constructor.

See examples [here](#)

Why method overloading is not possible by changing the return type in java?

In C++ and Java, functions can not be overloaded if they differ only in the return type . The return type of functions is not a part of the mangled name which is generated by the compiler for uniquely identifying each function. The No of arguments, Type of arguments & Sequence of arguments are the parameters which are used to generate the unique mangled name for each function. It is on the basis of these unique mangled names that compiler can understand which function to call even if the names are same(overloading).

Can we override private methods in Java?

No, a private method cannot be overridden since it is not visible from any other class. Read [more](#)

What is blank final variable?

A final variable in Java can be assigned a value only once, we can assign a value either in declaration or later.

```
final int i = 10;  
i = 30; // Error because i is final.
```

A **blank final** variable in Java is a `final` variable that is not initialized during declaration. Below is a simple example of blank final.

```
// A simple blank final example  
final int i;  
i = 30;
```

Read [more](#)

What is “super” keyword in java?

The super keyword in java is a reference variable that is used to refer parent class objects. The keyword “super” came into the picture with the concept of Inheritance. Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

Various scenarios of using java super Keyword:

- super is used to refer immediate parent instance variable
- super is used to call parent class method
- super() is used to call immediate parent constructor

Read [more](#)

What is static variable in Java?

The static keyword in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

- variable (also known as class variable)
- method (also known as class method)
- block
- nested class

Differences between HashMap and HashTable in Java.

1. HashMap is non synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code whereas Hashtable is synchronized. It is thread-safe and can be shared with many threads.
2. HashMap allows one null key and multiple null values whereas Hashtable doesn't allow any null key or value.
3. HashMap is generally preferred over HashTable if thread synchronization is not needed

Read [more](#)

How are Java objects stored in memory?

In Java, all objects are dynamically allocated on **Heap**. This is different from C++ where objects can be allocated memory either on Stack or on Heap. In C++, when we allocate object using new(), the object is allocated on Heap, otherwise on Stack if not global or static.

In Java, when we only declare a variable of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use new(). So the object is always allocated memory on heap. Read [more](#)

What are C++ features missing in Java?

Try to answer this on your own before seeing the answer – [here](#).

See also

- [Java Multiple Choice Questions](#)
- [Practice Coding Questions](#)
- [Java articles](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Java

Commonly asked DBMS interview questions | Set 1

[What are advantages of DBMS over traditional file based systems?](#)

Ans: Database management systems were developed to handle the following difficulties of typical file-processing systems supported by conventional operating systems.

1. Data redundancy and inconsistency
2. Difficulty in accessing data
3. Data isolation – multiple files and formats
4. Integrity problems
5. Atomicity of updates
6. Concurrent access by multiple users
7. Security problems

Source: <http://cs.nyu.edu/courses/spring01/G22.2433-001/mod1.2.pdf>

What are super, primary, candidate and foreign keys?

Ans: A **superkey** is a set of attributes of a relation schema upon which all attributes of the schema are functionally dependent. No two rows can have the same value of super key attributes.

A **Candidate key** is minimal superkey, i.e., no proper subset of Candidate key attributes can be a superkey.

A **Primary Key** is one of the candidate keys. One of the candidate keys is selected as most important and becomes the primary key. There cannot be more than one primary keys in a table.

Foreign key is a field (or collection of fields) in one table that uniquely identifies a row of another table. See [this](#) for an example.

What is the difference between primary key and unique constraints?

Ans: Primary key cannot have NULL value, the unique constraints can have NULL values. There is only one primary key in a table, but there can be multiple unique constraints.

What is database normalization?

Ans: It is a process of analyzing the given relation schemas based on their functional dependencies and primary keys to achieve the following desirable properties:

- 1) Minimizing Redundancy
- 2) Minimizing the Insertion, Deletion, And Update Anomalies

Relation schemas that do not meet the properties are decomposed into smaller relation schemas that could meet desirable properties.

Source: <http://cs.tsu.edu/ghemri/CS346/ClassNotes/Normalization.pdf>

What is SQL?

SQL is Structured Query Language designed for inserting and modifying in a **relational database system**.

What are the differences between DDL, DML and DCL in SQL?

Ans: Following are some details of three.

DDL stands for Data Definition Language. SQL queries like CREATE, ALTER, DROP and RENAME come under this.

DML stands for Data Manipulation Language. SQL queries like SELECT, INSERT and UPDATE come under this.

DCL stands for Data Control Language. SQL queries like GRANT and REVOKE come under this.

What is the difference between having and where clause?

Ans: HAVING is used to specify a condition for a group or an aggregate function used in select statement. The WHERE clause selects before grouping. The HAVING clause selects rows after grouping. Unlike HAVING clause, the WHERE clause cannot contain aggregate functions. (See [this](#) for examples).

See [Having vs Where Clause?](#) for more details

How to print duplicate rows in a table?

Ans: See <http://quiz.geeksforgeeks.org/how-to-print-duplicate-rows-in-a-table/>

What is Join?

Ans: An SQL Join is used to combine data from two or more tables, based on a common field between them. For example, consider the following two tables.

Student Table

ENROLLNO	STUDENTNAME	ADDRESS
1000	geek1	geeksquiz1
1001	geek2	geeksquiz2
1002	geek3	geeksquiz3

StudentCourse Table

COURSEID	ENROLLNO
1	1000
2	1000
3	1000
1	1002
2	1003

Following is join query that shows names of students enrolled in different courseIDs.

```
SELECT StudentCourse.CourseID, Student.StudentName  
      FROM StudentCourse  
INNER JOIN Customers  
    ON StudentCourse.EnrollNo = Student.EnrollNo  
ORDER BY StudentCourse.CourseID;
```

The above query would produce following result.

COURSEID	STUDENTNAME
1	geek1
1	geek2
2	geek1
2	geek3
3	geek1

What is Identity?

Ans: Identity (or AutoNumber) is a column that automatically generates numeric values. A start and increment value can be set, but most DBA leave these at 1. A GUID column also generates numbers; the value of this cannot be controlled. Identity/GUID columns do not need to be indexed.

What is a view in SQL? How to create one

Ans: A **view** is a virtual table based on the result-set of an SQL statement. We can create using create view syntax.

```
CREATE VIEW view_name AS  
SELECT column_name(s)  
FROM table_name  
WHERE condition
```

What are the uses of view?

1. Views can represent a subset of the data contained in a table; consequently, a view can limit the degree of exposure of the underlying tables to the outer world: a given user may have permission to query the view, while denied access to the rest of the base table.
2. Views can join and simplify multiple tables into a single virtual table
3. Views can act as aggregated tables, where the database engine aggregates data (sum, average etc.) and presents the calculated results as part of the data
4. Views can hide the complexity of data; for example a view could appear as Sales2000 or Sales2001, transparently partitioning the actual underlying table
5. Views take very little space to store; the database contains only the definition of a view, not a copy of all the data which it presents.
6. Depending on the SQL engine used, views can provide extra security

Source: [Wiki Page](#)

What is a Trigger?

Ans: A **Trigger** is a code that associated with insert, update or delete operations. The code is executed automatically whenever the associated query is executed on a table. Triggers can be useful to maintain integrity in database.

What is a stored procedure?

Ans: A **stored procedure** is like a function that contains a set of operations compiled together. It contains a set of operations that are commonly used in an application to do some common database tasks.

What is the difference between Trigger and Stored Procedure?

Ans: Unlike Stored Procedures, Triggers cannot be called directly. They can only be associated with queries.

What is a transaction? What are ACID properties?

Ans: A Database Transaction is a set of database operations that must be treated as whole, means either all operations are executed or none of them.

An example can be bank transaction from one account to another account. Either both debit and credit operations must be executed or none of them.

ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably.

What are indexes?

Ans: A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and the use of more storage space to maintain the extra copy of data.

Data can be stored only in one order on disk. To support faster access according to different values, faster search like binary search for different values is desired. For this purpose, indexes are created on tables. These indexes need extra space on disk, but they allow faster search according to different frequently searched values.

What are clustered and non-clustered Indexes?

Ans: Clustered indexes is the index according to which data is physically stored on disk. Therefore, only one clustered index can be created on a given database table.

Non-clustered indexes don't define physical ordering of data, but logical ordering. Typically, a tree is created whose leaf point to disk records. B-Tree or B+ tree are used for this purpos

- [Commonly asked DBMS interview questions | Set 2](#)
- [Practice Quizzes on DBMS](#)
- [Last Minute Notes – DBMS](#)
- [DBMS Articles](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: DBMS

Commonly Asked Operating Systems Interview Questions | Set 1

What is a process and process table? What are different states of process

A process is an instance of program in execution. For example a Web Browser is a process, a shell (or command prompt) is a process.

The operating system is responsible for managing all the processes that are running on a computer and allocated each process a certain amount of time to use the processor. In addition, the operating system also allocates various other resources that processes will need such as computer memory or disks. To keep track of the state of all the processes, the operating system maintains a table known as the *process table*. Inside this table, every process is listed along with the resources the processes is using and the current state of the process.

Processes can be in one of three states: running, ready, or waiting. The running state means that the process has all the resources it need for execution and it has been given permission by the operating system to use the processor. Only one process can be in the running state at any given time. The remaining processes are either in a waiting state (i.e., waiting for some external event to occur such as user input or a disk access) or a ready state (i.e., waiting for permission to use the processor). In a real operating system, the waiting and ready states are implemented as queues which hold the processes in these states. The animation below shows a simple representation of the life cycle of a process (Source: <http://courses.cs.vt.edu/csonline/OS/Lessons/Processes/index.html>)

What is a Thread? What are the differences between process and thread?

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*. Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

A thread has its own program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section and OS resources like open files and signals. See <http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/threads.htm> for more details.

What is deadlock?

Deadlock is a situation when two or more processes wait for each other to finish and none of them ever finish. Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other.

Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s).

What are the necessary conditions for deadlock?

Mutual Exclusion: There is a resource that cannot be shared.

Hold and Wait: A process is holding at least one resource and waiting for another resource which is with some other process.

No Preemption: The operating system is not allowed to take a resource back from a process until process gives it back.

Circular Wait: A set of processes are waiting for each other in circular form.

What is Virtual Memory? How is it implemented?

Virtual memory creates an illusion that each user has one or more contiguous address spaces, each beginning at address zero. The sizes of such virtual address spaces is generally very high.

The idea of virtual memory is to use disk space to extend the RAM. Running processes don't need to care whether the memory is from RAM or disk. The illusion of such a large amount of memory is created by subdividing the virtual memory into smaller pieces, which can be loaded into physical memory whenever they are needed by a process.

What is Thrashing?

Thrashing is a situation when the performance of a computer degrades or collapses. Thrashing occurs when a system spends more time processing page faults than executing transactions. While processing page faults is necessary to in order to appreciate the benefits of virtual memory, thrashing has a negative affect on the system. As the page fault rate increases, more transactions need processing from the paging device. The queue at the paging device increases, resulting in increased service time for a page fault (Source: <http://cs.gmu.edu/cne/modules/vm/blue/thrash.html>)

What is Belady's Anomaly?

Bélády's anomaly is an anomaly with some page replacement policies where increasing the number of page frames results in an increase in the number of page faults. It occurs with First in First Out page replacement is used. See [the wiki page](#) for an example and more details.

Differences between mutex and semaphore?

See <http://www.geeksforgeeks.org/mutex-vs-semaphore/>

- Practice [Quizzes](#) on Operating System topics
- [Last Minute Notes – OS](#)
- [OS articles](#)

We will soon be covering more Operating System questions. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Operating Systems

Commonly Asked Data Structure Interview Questions | Set 1

What is a Data Structure?

A data structure is a way of organizing the data so that the data can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers. (Source: [Wiki Page](#))

What are linear and non linear data Structures?

- **Linear:** A data structure is said to be linear if its elements form a sequence or a linear list. Examples: Array, Linked List, Stacks and Queues
- **Non-Linear:** A data structure is said to be linear if traversal of nodes is nonlinear in nature. Example: Graph and Trees.

What are the various operations that can be performed on different Data Structures?

- **Insertion** – Add a new data item in the given collection of data items.
- **Deletion** – Delete an existing data item from the given collection of data items.
- **Traversal** – Access each data item exactly once so that it can be processed.
- **Searching** – Find out the location of the data item if it exists in the given collection of data items.
- **Sorting** – Arranging the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

How is an Array different from Linked List?

- The size of the arrays is fixed, Linked Lists are Dynamic in size.
- Inserting and deleting a new element in an array of elements is expensive, Whereas both insertion and deletion can easily be done in Linked Lists.
- Random access is not allowed in Linked Listed.
- Extra memory space for a pointer is required with each element of the Linked list.
- Arrays have better cache locality that can make a pretty big difference in performance.

What is Stack and where it can be used?

Stack is a linear data structure which follows the order LIFO(Last In First Out) or FILO(First In Last Out) for accessing elements. Basic operations of stack are : **Push, Pop , Peek**

Applications of Stack:

1. Infix to Postfix Conversion using Stack
2. Evaluation of Postfix Expression
3. Reverse a String using Stack
4. Implement two stacks in an array
5. Check for balanced parentheses in an expression

What is a Queue, how it is different from stack and how is it implemented?

Queue is a linear structure which follows the order is **First In First Out (FIFO)** to access elements. Mainly the following are basic operations on queue: **Enqueue, Dequeue, Front, Rear**

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added. Both Queues and Stacks can be implemented using Arrays and Linked Lists.

What are Infix, prefix, Postfix notations?

- **Infix notation:** X + Y – Operators are written in-between their operands. This is the usual way we write expressions. An expression such as

$$A * (B + C) / D$$
- **Postfix notation (also known as “Reverse Polish notation”):** X Y + Operators are written after their operands. The infix expression given above is equivalent to

$$A B C + * D$$
- **Prefix notation (also known as “Polish notation”):** + X Y Operators are written before their operands. The expressions given above are equivalent to

$$/* A + B C D$$

Converting between these notations: [Click here](#)

What is a Linked List and What are its types?

A linked list is a linear data structure (like arrays) where each element is a separate object. Each element (that is node) of a list is comprising of two items – the data and a reference to the next node.Types of Linked List :

1. **Singly Linked List :** In this type of linked list, every node stores address or reference of next node in list and the last node has next address or reference as NULL. For example 1->2->3->4->NULL
2. **Doubly Linked List :** Here, here are two references associated with each node, One of the reference points to the next node and one to

the previous node. Eg. NULL<-1<->2<->3->NULL

3. **Circular Linked List** : Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list. Eg. 1->2->3->1 [The next pointer of last node is pointing to the first]

Which data structures are used for BFS and DFS of a graph?

- Queue is used for BFS
- Stack is used for DFS. DFS can also be implemented using recursion (Note that recursion also uses function call stack).

Can doubly linked be implemented using a single pointer variable in every node?

Doubly linked list can be implemented using a single pointer. See [XOR Linked List – A Memory Efficient Doubly Linked List](#)

How to implement a stack using queue?

A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

- Method 1 (By making push operation costly)
- Method 2 (By making pop operation costly) See [Implement Stack using Queues](#)

How to implement a queue using stack?

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

- Method 1 (By making enQueue operation costly)
- Method 2 (By making deQueue operation costly) See [Implement Queue using Stacks](#)

Which Data Structure Should be used for implementiong LRU cache?

We use two data structures to implement an LRU Cache.

1. **Queue** which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size).The most recently used pages will be near front end and least recently pages will be near rear end.
2. **A Hash** with page number as key and address of the corresponding queue node as value. See [How to implement LRU caching scheme? What data structures should be used?](#)

How to check if a given Binary Tree is BST or not?

If inorder traversal of a binary tree is sorted, then the binary tree is BST. The idea is to simply do inorder traversal and while traversing keep track of previous key value. If current key value is greater, then continue, else return false. See [A program to check if a binary tree is BST or not](#) for more details.

Linked List Questions

- Linked List Insertion
- Linked List Deletion
- middle of a given linked list
- Nth node from the end of a Linked List

Tree Traversal Questions

- Inorder
- Preorder and Postoder Traversals
- Level order traversal

- Height of Binary Tree

Convert a DLL to Binary Tree in-place

See [In-place conversion of Sorted DLL to Balanced BST](#)

Convert Binary Tree to DLL in-place

See [Convert a given Binary Tree to Doubly Linked List | Set 1](#), [Convert a given Binary Tree to Doubly Linked List | Set 2](#)

Delete a given node in a singly linked list

Given only a pointer to a node to be deleted in a singly linked list, how do you delete it?

Reverse a Linked List

Write a function to reverse a linked list

Detect Loop in a Linked List

Write a C function to detect loop in a linked list.

Which data structure is used for dictionary and spell checker?

Data Structure for Dictionary and Spell Checker?

You may also like

- Practice Quizzes on Data Structures
- Last Minute Notes – DS
- Common Interview Puzzles
- Amazon's most asked interview questions
- Microsoft's most asked interview questions
- Accenture's most asked Interview Questions
- Commonly Asked OOP Interview Questions
- Commonly Asked C++ Interview Questions,
- Commonly Asked C Programming Interview Questions | Set 1
- Commonly Asked C Programming Interview Questions | Set 2
- Commonly asked DBMS interview questions | Set 1
- Commonly Asked Operating Systems Interview Questions | Set 1
- Commonly Asked Data Structure Interview Questions
- Commonly Asked Algorithm Interview Questions

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Data Structures

Commonly Asked Algorithm Interview Questions | Set 1

What is an algorithm?

Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output. (Source: [Introduction to Algorithms 3rd Edition by CLRS](#))

What is time complexity of Binary Search?

Time complexity of binary search is O(Logn). See [Binary Search](#) for more details.

Can Binary Search be used for linked lists?

Since random access is not allowed in linked list, we cannot reach the middle element in O(1) time. Therefore Binary Search is not possible for linked lists. There are other ways though, refer [Skip List](#) for example.

How to find if two given rectangles overlap?

Two rectangles do not overlap if one of the following conditions is true.

- 1) One rectangle is above top edge of other rectangle.
- 2) One rectangle is on left side of left edge of other rectangle.

See [Find if two rectangles overlap](#) for more details.

How to find angle between hour and minute hands at a given time?

The idea is to take a reference point as 12. Find the angle moved by hour and minute hands, subtract the two angles to find the angle between them. See [angle between hour hand and minute hand](#) for more details

When does the worst case of QuickSort occur?

In [quickSort](#), we select a pivot element, then partition the given array around the pivot element by placing pivot element at its correct position in sorted array.

The worst case of quickSort occurs when one part after partition contains all elements and other part is empty. For example, if the input array is sorted and if last or first element is chosen as a pivot, then the worst occurs. See <http://quiz.geeksforgeeks.org/quick-sort/> for more details.

A sorted array is rotated at some unknown point, how to efficiently search an element in it?

A simple approach is linear search, but we can search in O(Logn) time using [Binary Search](#). See [Search an element in a sorted and pivoted array](#) for more details.

Other variations of this problem like [find the minimum element or maximum element in a sorted and rotated array](#).

Given a big string of characters, how to efficiently find the first unique character in it?

The efficient solution is to use character as an index in a count array. Traverse the given string and store index of first occurrence of every character, also store count of occurrences. Then traverse the count array and find the smallest index with count as 1. See [find the first unique character](#) for more details.

How to count inversions in a sorted array?

Two elements arr[i] and arr[j] in an array arr[] form an inversion if $a[i] > a[j]$ and $i < j$. How to count all inversions in an unsorted array. See [Count Inversions in an array](#) for all approaches.

Given a big array, how to efficiently find k'th largest element in it?

There can be many solutions for this. The best solution is to use min heap. We Build a Min Heap MH of the first k elements. For each element, after the kth element (arr[k] to arr[n-1]), compare it with root of MH, if the element is greater than the root then make it root and call heapify for MH, Else ignore it. Finally, MH has k largest elements and root of the MH is the kth largest element. See [k largest\(or smallest\) elements](#) for more details.

Given an array of size n with range of numbers from 1 to n+1. The array doesn't contain any duplicate, one number is missing, find the missing number.

There can be many ways to solve it. The best among is to use XOR. See [Find the missing number](#) for details. There are many variations of this problem like [find the two repeating numbers](#), [find a missing and a repeating number](#), etc.

How to write an efficient method to calculate x raise to the power n?

The idea is to use [divide an conquer](#) here to do it in O(Logn) time. See [Write a C program to calculate pow\(x,n\)](#) for more details.

Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words.

The idea is to use Dynamic Programming. See [Word Break Problem](#) for more details.

Given a row of n coins of values v₁ . . . v_n, where n is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

This is also a Dynamic Programming Question. See [Optimal Strategy for a Game](#) for more details.

You are given an array of sorted words in an arbitrary language, you need to find order (or precedence of characters) in the language.

For example if the given arrays is {"baa", "abcd", "abca", "cab", "cad"}, then order of characters is 'b', 'd', 'a', 'c'. Note that words are sorted and in the given language "baa" comes before "abcd", therefore 'b' is before 'a' in output. Similarly we can find other orders.

This can be solved using two steps: First create a graph by processing given set of words, then do [topological sorting](#) of the created graph, See [this](#) for more details.

You may also like

- [Commonly Asked Data Structure Interview Questions | Set 1](#)
- [Practice Quizzes on various Algorithm Topics](#)

- Last Minute Notes – Algo

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Algorithms

Commonly asked Computer Networks Interview Questions | Set 1

What are Unicasting, Anycasting, Multicasting and Broadcasting?

If the message is sent from a source to a single destination node, it is called Unicasting. This is typically done in networks.

If the message is sent from a source to any of the given destination nodes. This is used a lot in Content delivery Systems where we want to get content from any server.

If the message is sent to some subset of other nodes, it is called Multicasting. Used in situation when there are multiple receivers of same data. Like video conferencing, updating something on CDN servers which have replica of same data.

If the message is sent to all the nodes in a network it is called Broadcasting. This is typically used in Local networks, for examples DHCP and ARP use broadcasting.

What are layers in OSI model?

There are total 7 layers

1. Physical Layer
2. Data Link Layer
3. Network Layer
4. Transport Layer
5. Session Layer
6. Presentation Layer
7. Application Layer

What is Stop-and-Wait Protocol?

In Stop and wait protocol, a sender after sending a frame waits for acknowledgement of the frame and sends the next frame only when acknowledgement of the frame has received.

What is Piggybacking?

Piggybacking is used in bi-directional data transmission in the network layer (OSI model). The idea is to improve efficiency piggy back acknowledgement (of the received data) on the data frame (to be sent) instead of sending a separate frame.

Differences between Hub, Switch and Router?

Hub	Switch	Router
Physical Layer Device	Data Link Layer Device	Network Layer Device
Simply repeats signal to all ports	Doesn't simply repeat, but filters content by MAC or LAN address	Routes data based on IP address
Connects devices within a single LAN	Can connect multiple sub-LANs within a single LAN	Connect multiple LANS and WANS together.
Collision domain of all hosts connected through Hub remains one. i.e., if signal sent by any two devices can collide.	Switch divides collision domain, but broadcast domain of connected devices remains same.	It divides both collision and broadcast domains,

See [network devices](#) for more details.

What happens when you type a URL in web browser?

A URL may contain request to HTML, image file or any other type.

1. If content of the typed URL is in cache and fresh, then display the content.
2. Else find IP address for the domain so that a TCP connection can be setup. Browser does a DNS lookup.
3. Browser needs to know IP address for a url, so that it can setup a TCP connection. This is why browser needs DNS service. Browser first looks for URL-IP mapping browser cache, then in OS cache. If all caches are empty, then it makes a recursive query to the local DNS server. The local DNS server provides the IP address.
4. Browser sets up a TCP connection using three way handshake.
5. Browser sends a HTTP request.

6. Server has a web server like Apache, IIS running that handles incoming HTTP request and sends a HTTP response.
7. Browser receives the HTTP response and renders the content.

What is DHCP, how does it work?

1. The idea of DHCP (Dynamic Host Configuration Protocol) is to enable devices to get IP address without any manual configuration.
2. The device sends a broadcast message saying "I am new here"
3. The DHCP server sees the message and responds back to the device and typically allocates an IP address. All other devices on network ignore the message of new device as they are not DHCP server.

In Wi Fi networks, Access Points generally work as a DHCP server.

What is ARP, how does it work?

ARP stands for Address Resolution Protocol. ARP is used to find LAN address from Network address. A node typically has destination IP to send a packet, the nodes needs link layer address to send a frame over local link. The ARP protocol helps here.

1. The node sends a broadcast message to all nodes saying what is the MAC address of this IP address.
2. Node with the provided IP address replies with the MAC address.

Like DHCP, ARP is a discovery protocol, but unlike DHCP there is not server here.

See [this](#) video for detailed explanation.

Practice [Quizzes](#) for Networking

You may also like

- [Common Interview Puzzles](#)
- [Amazon's most asked interview questions](#)
- [Microsoft's most asked interview questions](#)
- [Accenture's most asked Interview Questions](#)
- [Commonly Asked OOP Interview Questions](#)
- [Commonly Asked C++ Interview Questions,](#)
- [Commonly Asked C Programming Interview Questions | Set 1](#)
- [Commonly Asked C Programming Interview Questions | Set 2](#)
- [Commonly asked DBMS interview questions | Set 1](#)
- [Commonly Asked Operating Systems Interview Questions | Set 1](#)
- [Commonly Asked Data Structure Interview Questions](#)
- [Commonly Asked Algorithm Interview Questions](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

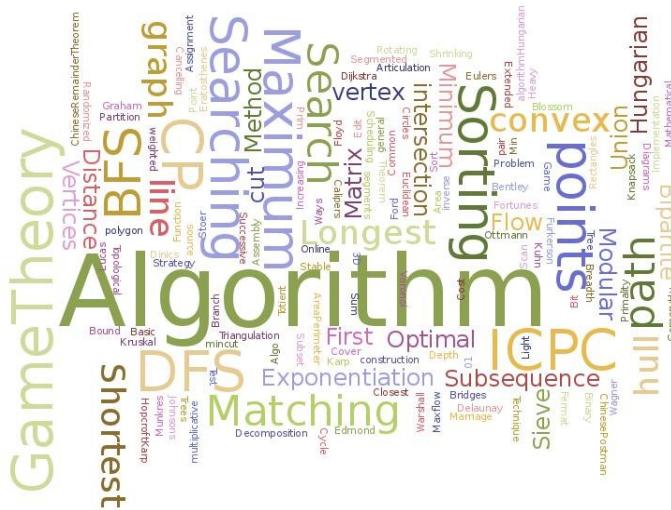
See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Computer Networks

Top 10 Algorithms and Data Structures for Competitive Programming

In this post "Important top 10 algorithms and data structures for competitive coding".

Topics :

1. [Graph algorithms](#)
2. [Dynamic programming](#)
3. [Searching and Sorting:](#)
4. [Number theory and Other Mathematical](#)
5. [Geometrical and Network Flow Algorithms](#)
6. [Data Structures](#)



The below links cover all most important algorithms and data structure topics:

Graph Algorithms

1. Breadth First Search (BFS)
 2. Depth First Search (DFS)
 3. Shortest Path from source to all vertices **Dijkstra**
 4. Shortest Path from every vertex to every other vertex **Floyd Warshall**
 5. Minimum Spanning tree **Prim**
 6. Minimum Spanning tree **Kruskal**
 7. Topological Sort
 8. Johnson's algorithm
 9. Articulation Points (or Cut Vertices) in a Graph
 10. Bridges in a graph

All Graph Algorithms

Dynamic Programming

1. Longest Common Subsequence
 2. Longest Increasing Subsequence
 3. Edit Distance
 4. Minimum Partition
 5. Ways to Cover a Distance
 6. Longest Path In Matrix
 7. Subset Sum Problem
 8. Optimal Strategy for a Game
 9. 0-1 Knapsack Problem
 10. Assembly Line Scheduling

All DP Algorithms

Searching And Sorting

1. Binary Search
 2. Quick Sort
 3. Merge Sort
 4. Order Statistics
 5. KMP algorithm
 6. Rabin karp
 7. Z's algorithm
 8. Aho Corasick String Matching
 9. Counting Sort

10. Manacher's algorithm: [Part 1](#), [Part 2](#) and [Part 3](#)

All Articles on [Searching](#), [Sorting](#) and [Pattern Searching](#).

Number theory and Other Mathematical

Prime Numbers and Prime Factorization

1. Primality Test | Set 1 (Introduction and School Method)
2. Primality Test | Set 2 (Fermat Method)
3. Primality Test | Set 3 (Miller–Rabin)
4. Sieve of Eratosthenes
5. Segmented Sieve
6. Wilson's Theorem
7. Prime Factorisation
8. Pollard's rho algorithm

Modulo Arithmetic Algorithms

1. Basic and Extended Euclidean algorithms
2. Euler's Totient Function
3. Modular Exponentiation
4. Modular Multiplicative Inverse
5. Chinese remainder theorem Introduction
6. Chinese remainder theorem and Modulo Inverse Implementation
7. $nCr \% m$ and this.

Miscellaneous:

1. Counting Inversions
2. Counting Inversions using BIT
3. logarithmic exponentiation
4. Square root of an integer
5. Heavy light Decomposition , [this](#) and [this](#)
6. Matrix Rank
7. Gaussian Elimination to Solve Linear Equations
8. Hungarian algorithm
9. Link cut
10. Mo's algorithm and [this](#)
11. Factorial of a large number in C++
12. Factorial of a large number in Java+
13. Russian Peasant Multiplication
14. Catalan Number

All Articles on Mathematical Algorithms

Geometrical and Network Flow Algorithms

1. Convex Hull
2. Graham Scan
3. Line Intersection
4. Interval Tree
5. Matrix Exponentiation and this
6. Maxflow Ford Fulkerson Algo and Edmond Karp Implementation
7. Min cut
8. Stable Marriage Problem
9. Hopcroft–Karp Algorithm for Maximum Matching
10. Dinic's algo and e-maxx

All Articles on Geometric Algorithms

Data Structures

1. Binary Indexed Tree or Fenwick tree
2. Segment Tree (RMQ, Range Sum and Lazy Propagation)
3. K-D tree (See insert, minimum and delete)
4. Union Find Disjoint Set (Cycle Detection and By Rank and Path Compression)
5. Tries
6. Suffix array (this, this and this)
7. Sparse table
8. Suffix automata
9. Suffix automata II
10. LCA and RMQ

All Articles on Advanced Data Structures.

How to Begin?

Please see [How to begin with Competitive Programming?](#)

How to Practice?

Please see <http://practice.geeksforgeeks.org/>

What are top algorithms in Interview Questions?

Top 10 algorithms in Interview Questions

How to prepare for ACM – ICPC?

[How to prepare for ACM – ICPC?](#)

This is an initial draft. We will soon be adding more links and algorithms to this post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Competitive Programming
BFS
Binary-Indexed-Tree
BIT
DFS
modular-arithmetic
Segment-Tree
sieve
Topological Sorting

Top 10 algorithms in Interview Questions

In this post "Top 10 coding problems of important topics with their solutions" are written. If you are preparing for a coding interview, going through these problems is a must.

Topics :

1. [Graph](#)
2. [Linked List](#)
3. [Dynamic Programming](#)
4. [Sorting And Searching](#)
5. [Tree / Binary Search Tree](#)
6. [Number Theory](#)
7. [BIT Manipulation](#)
8. [String / Array](#)

Graph

1. Breadth First Search (BFS)
2. Depth First Search (DFS)
3. Shortest Path from source to all vertices **Dijkstra**
4. Shortest Path from every vertex to every other vertex **Floyd Warshall**
5. To detect cycle in a Graph **Union Find**
6. Minimum Spanning tree **Prim**
7. Minimum Spanning tree **Kruskal**
8. Topological Sort
9. Boggle (Find all possible words in a board of characters)
10. Bridges in a Graph

Linked List

1. Insertion of a node in Linked List (On the basis of some constraints)
2. Delete a given node in Linked List (under given constraints)
3. Compare two strings represented as linked lists
4. Add Two Numbers Represented By Linked Lists
5. Merge A Linked List Into Another Linked List At Alternate Positions
6. Reverse A List In Groups Of Given Size
7. Union And Intersection Of 2 Linked Lists
8. Detect And Remove Loop In A Linked List
9. Merge Sort For Linked Lists
10. Select A Random Node from A Singly Linked List

Dynamic Programming

1. Longest Common Subsequence
2. Longest Increasing Subsequence
3. Edit Distance
4. Minimum Partition
5. Ways to Cover a Distance
6. Longest Path In Matrix
7. Subset Sum Problem
8. Optimal Strategy for a Game
9. 0-1 Knapsack Problem
10. Boolean Parenthesization Problem

Sorting And Searching

1. Binary Search
2. Search an element in a sorted and rotated array
3. Bubble Sort
4. Insertion Sort
5. Merge Sort
6. Heap Sort (Binary Heap)
7. Quick Sort
8. Interpolation Search
9. Find Kth Smallest/Largest Element In Unsorted Array
10. Given a sorted array and a number x, find the pair in array whose sum is closest to x

Tree / Binary Search Tree

1. Find Minimum Depth of a Binary Tree
2. Maximum Path Sum in a Binary Tree
3. Check if a given array can represent Preorder Traversal of Binary Search Tree
4. Check whether a binary tree is a full binary tree or not
5. Bottom View Binary Tree
6. Print Nodes in Top View of Binary Tree
7. Remove nodes on root to leaf paths of length < K
8. Lowest Common Ancestor in a Binary Search Tree
9. Check if a binary tree is subtree of another binary tree
10. Reverse alternate levels of a perfect binary tree

Number Theory

1. Modular Exponentiation
2. Modular multiplicative inverse
3. Primality Test | Set 2 (Fermat Method)
4. Euler's Totient Function
5. Sieve of Eratosthenes
6. Convex Hull
7. Basic and Extended Euclidean algorithms
8. Segmented Sieve
9. Chinese remainder theorem
10. Lucas Theorem

BIT Manipulation

1. Maximum Subarray XOR
2. Magic Number
3. Sum of bit differences among all pairs
4. Swap All Odds And Even Bits
5. Find the element that appears once
6. Binary representation of a given number

7. Count total set bits in all numbers from 1 to n
8. Rotate bits of a number
9. Count number of bits to be flipped to convert A to B
10. Find Next Sparse Number

String / Array

1. Reverse an array without affecting special characters
2. All Possible Palindromic Partitions
3. Count triplets with sum smaller than a given value
4. Convert array into Zig-Zag fashion
5. Generate all possible sorted arrays from alternate elements of two given sorted arrays
6. Pythagorean Triplet in an array
7. Length of the largest subarray with contiguous elements
8. Find the smallest positive integer value that cannot be represented as sum of any subset of a given array
9. Smallest subarray with sum greater than a given value
10. Stock Buy Sell to Maximize Profit

Top 10 Algorithms and Data Structures for Competitive Programming

Company-wise Practice Questions

Interview Corner

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Misc
BFS
DFS
Interview Tips
interview-preparation
placement preparation
Topological Sorting