II B.Tech. I SEM **Regulation: R23**

Laboratory Manual

For the course of

ADVANCED DATA STRUCTURES
&
ALGORITHM ANALYSIS LAB

Branch: AIML , CSM, CSE, CAI

VIGNAN'S LARA
INSTITUTE OF TECHNOLOGY & SCIENCE
(AUTONOMOUS)
Approved by AICTE New Delhi & Affiliated to JNTUK Kakinada
Accredited by NAAC 'A+' and NBA | ISO 9001 : 2015
Vadlamudi - 522 213, Guntur District

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

II Year I Semester

L	T	P	C
0	0	3	1.5

ADVANCED DATA STRUCTURES & ALGORITHM ANALYSIS LAB

Course Objectives:

The objectives of the course is to

- acquire practical skills in constructing and managing Data structures
- apply the popular algorithm design methods in problem-solving scenarios

Experiments covering the Topics:

- Operations on AVL trees, B-Trees, Heap Trees
- Graph Traversals
- Sorting techniques
- Minimum cost spanning trees
- Shortest path algorithms
- 0/1 Knapsack Problem
- Travelling Salesperson problem
- Optimal Binary Search Trees
- N-Queens Problem
- Job Sequencing

1. Write a main function to create objects of DISTANCE Class Input two distances and output the sum.
2. Write a program to illustrate pointers to a class.
3. Construct B-Tree an order of 5 with a set of 100 random elements stored in array. Implement searching, insertion and deletion operations.
4. Construct Min and Max Heap using arrays, delete any element and display the content of the Heap.
5. Implement BFT and DFT for given graph, when graph is represented by
   a) Adjacency Matrix
   b) Adjacency Lists
6. Write a program for finding the bi connected components in a given graph.
7. Implement Quick sort and Merge sort and observe the execution time for various input sizes (Average, Worst and Best cases).
8. Compare the performance of Single Source Shortest Paths using Greedy method when the graph is represented by adjacency matrix and adjacency lists.
9. Implement Job Sequencing with deadlines using Greedy strategy.
10. Write a program to solve 0/1 Knapsack problem Using Dynamic Programming.
11. Implement N-Queens Problem Using Backtracking.
12. Use Backtracking strategy to solve 0/1 Knapsack problem.

Reference Books:

1. Fundamentals of Data Structures in C++, Horowitz Ellis, SahniSartaj, Mehta, Dinesh,
2. 2[nd]Edition, Universities Press

3. Computer Algorithms/C++ Ellis Horowitz, SartajSahni, SanguthevarRajasekaran,
4. 2[nd]Edition, University Press
   Data Structures and program design in C, Robert Kruse, Pearson Education Asia
   An introduction to Data Structures with applications, Trembley& Sorenson, McGraw Hill

Online Learning Resources:

1. http://cse01-iiith.vlabs.ac.in/
2. http://peterindia.net/Algorithms.html

INDEX

Exp.No	Name of the Experiment	Page. No
1	Write a main function to create objects of DISTANCE Class Input two distances and output the sum .	01-02
2	Write a program to illustrate pointers to a class	03
3	Construct B-Tree an order of 5 with a set of 100 random elements stored in array. Implement searching, insertion and deletion operations.	04-08
4	Construct Min and Max Heap using arrays, delete any element and display the content of the Heap.	09-11
5	Implement BFT and DFT for given graph, when graph is represented by a) Adjacency Matrix b) Adjacency Lists	12-16
6	Write a program for finding the bi connected components in a given graph.	17-19
7	Implement Quick sort and Merge sort and observe the execution time for various input sizes (Average, Worst and Best cases).	20-23
8	Compare the performance of Single Source Shortest Paths using Greedy method when the graph is represented by adjacency matrix and adjacency lists.	24-28
9	Implement Job Sequencing with deadlines using Greedy strategy.	29-30
10	Write a program to solve 0/1 Knapsack problem Using Dynamic Programming.	31-33
11	Implement N-Queens Problem Using Backtracking.	34-37
12	Use Backtracking strategy to solve 0/1 Knapsack problem.	38-40

1. **Write a main function to create objects of DISTANCE Class Input two distances and output the sum**

```
#include <iostream>
class DISTANCE {
public:
   int feet;
   float inches;
DISTANCE() : feet(0), inches(0.0) {}
DISTANCE(int f, float i) : feet(f), inches(i) {}
   void input() {
std::cout<< "Enter feet: ";
std::cin>> feet;
std::cout<< "Enter inches: ";
std::cin>> inches;
   }
   void display() const {
std::cout<< feet << " feet " << inches << " inches" << std::endl;
   }
   DISTANCE operator+(const DISTANCE& d) const {
      int f = feet + d.feet;
      float i = inches + d.inches;
      if (i>= 12.0) {
i -= 12.0;
         f++;
      }
      return DISTANCE(f, i);
   }};
int main() {
   DISTANCE d1, d2, d3;
std::cout<< "Enter first distance:\n";
   d1.input();
std::cout<< "Enter second distance:\n";
   d2.input();
   d3 = d1 + d2;
std::cout<< "Sum of distances: ";
   d3.display();
   return 0;
    }
```

OUT PUT:

```
Enter first distance:
Enter feet: 5
Enter inches: 3
Enter second distance:
Enter feet: 6
Enter inches: 5
    Sum of distances: 11 feet 8 inches
```

2. Write a program to illustrate pointers to a class

```
#include <iostream>

class MyClass {

public:

   void display() {

std::cout<< "Display function called" << std::endl;

   }

};

int main() {

MyClass obj;

MyClass *ptr = &obj;

ptr->display();

   return 0;

}
```

OUT PUT:

Display function called

3. Construct B-Tree an order of5 with a set of 100 random elements store din array. Implement searching ,insertion and deletion operations.

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#define ORDER 5
class BTreeNode {
public:
   int *keys;
   int t;
BTreeNode **C;
   int n;
   bool leaf;
BTreeNode(int _t, bool _leaf);
   void traverse();
BTreeNode *search(int k);
   void insertNonFull(int k);
   void splitChild(int i, BTreeNode *y);
   friend class BTree;
};
class BTree {
public:
BTreeNode *root;
   int t;
BTree(int _t) {
      root = nullptr;
      t = _t;
```

```
    }
    void traverse() {
        if (root != nullptr) root->traverse();
    }
BTreeNode* search(int k) {
        return (root == nullptr) ?nullptr : root->search(k);
    }
    void insert(int k);
};
BTreeNode::BTreeNode(int t1, bool leaf1) {
    t = t1;
    leaf = leaf1;
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];
    n = 0;
}
void BTreeNode::traverse() {
    int i;
    for (i = 0; i< n; i++) {
        if (leaf == false)
            C[i]->traverse();
std::cout<< " " << keys[i];
    }
    if (leaf == false)
        C[i]->traverse();
}
BTreeNode *BTreeNode::search(int k) {
    int i = 0;
```

```
    while (i< n && k > keys[i])
i++;
    if (keys[i] == k)
        return this;
    if (leaf == true)
        return nullptr;
    return C[i]->search(k);
}
void BTree::insert(int k) {
    if (root == nullptr) {
        root = new BTreeNode(t, true);
        root->keys[0] = k;
        root->n = 1;
    } else {
        if (root->n == 2*t-1) {
BTreeNode *s = new BTreeNode(t, false);
            s->C[0] = root;
            s->splitChild(0, root);
            int i = 0;
            if (s->keys[0] < k)
i++;
            s->C[i]->insertNonFull(k);
            root = s;
        } else
            root->insertNonFull(k);
    }
}
void BTreeNode::insertNonFull(int k) {
```

```
    int i = n-1;
    if (leaf == true) {
        while (i>= 0 && keys[i] > k) {
            keys[i+1] = keys[i];
i--;
        }
        keys[i+1] = k;
        n = n+1;
    } else {
        while (i>= 0 && keys[i] > k)
i--;
        if (C[i+1]->n == 2*t-1) {
splitChild(i+1, C[i+1]);
            if (keys[i+1] < k)
i++;
        }
        C[i+1]->insertNonFull(k);
    }
}
void BTreeNode::splitChild(int i, BTreeNode *y) {
BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t - 1;
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];
    if (y->leaf == false) {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }
```

```
    y->n = t - 1;
    for (int j = n; j >= i+1; j--)
        C[j+1] = C[j];
    C[i+1] = z;
    for (int j = n-1; j >= i; j--)
        keys[j+1] = keys[j];
    keys[i] = y->keys[t-1];
    n = n + 1;
}
int main() {
srand(time(0));
BTree t(ORDER);
std::vector<int>arr(100);
    for (int i = 0; i< 100; i++) {
arr[i] = rand() % 1000;
t.insert(arr[i]);
    }
std::cout<< "Traversal of the constructed B-Tree is:\n";
t.traverse();
    return 0;
}
```

OUT PUT:

Traversal of the constructed B-Tree is:

11 18 22 26 26 42 47 86 99 105 131 154 159 171 187 187 202 206 211 215 234 244 252 279 289 300 310 330 341 356 357 360 366 387 391 394 417 422 444 450 473 474 476 481 489 492 493 503 522 528 529 531 540 543 556 575 576 591 592 598 600 610 640 697 698 716 717 727 742 744 744 745 788 790 803 813 833 836 837 850 854 876 878 882 885 892 898 908 911 917 918 925 927 934 951 952 973 978 983 994

4. Construct Min and Max Heap using arrays , delete any element and display the content of the Heap.

```
#include <iostream>
#include <vector>
#include <algorithm>
void heapify(std::vector<int>&arr, int n, int i, bool isMaxHeap) {
   int largest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;
   if (isMaxHeap) {
      if (l < n &&arr[l] >arr[largest])
         largest = l;
      if (r < n &&arr[r] >arr[largest])
         largest = r;
   } else {
      if (l < n &&arr[l] <arr[largest])
         largest = l;
      if (r < n &&arr[r] <arr[largest])
         largest = r;
   }
   if (largest != i) {
std::swap(arr[i], arr[largest]);
heapify(arr, n, largest, isMaxHeap);
   }
}
void buildHeap(std::vector<int>&arr, bool isMaxHeap) {
   int n = arr.size();
```

```
    for (int i = n / 2 - 1; i>= 0; i--) {
heapify(arr, n, i, isMaxHeap);
    }
}
void deleteElement(std::vector<int>&arr, int num, bool isMaxHeap) {
    auto it = std::find(arr.begin(), arr.end(), num);
    if (it != arr.end()) {
        int index = std::distance(arr.begin(), it);
std::swap(arr[index], arr.back());
arr.pop_back();
heapify(arr, arr.size(), index, isMaxHeap);
    }
}
void displayHeap(const std::vector<int>&arr) {
    for (int val :arr) {
std::cout<<val<< " ";
    }
std::cout<< std::endl;
}
int main() {
std::vector<int>minHeap = {3, 5, 9, 6, 8, 20, 10, 12, 18, 9};
std::vector<int>maxHeap = minHeap;
buildHeap(minHeap, false);
buildHeap(maxHeap, true);
std::cout<< "Min Heap:\n";
displayHeap(minHeap);
std::cout<< "Max Heap:\n";
displayHeap(maxHeap);
```

```
    int num;

std::cout<< "Enter element to delete from Min Heap: ";

std::cin>> num;

deleteElement(minHeap, num, false);

std::cout<< "Min Heap after deletion:\n";

displayHeap(minHeap);

std::cout<< "Enter element to delete from Max Heap: ";

std::cin>> num;

deleteElement(maxHeap, num, true);
std::cout<< "Max Heap after deletion:\n";
displayHeap(maxHeap);
    return 0;
}
```

OUT PUT:

Min Heap:

3 5 9 6 8 20 10 12 18 9

Max Heap:

20 18 10 12 9 9 3 5 6 8

Enter element to delete from Min Heap: 9

Min Heap after deletion:

3 5 9 6 8 20 10 12 18

Enter element to delete from Max Heap: 9

Max Heap after deletion:

20 18 10 12 8 9 3 5 6

5. Implement BFT and DFT for given graph, when graphics represented by
a) Adjacency Matrix b)Adjacency Lists

A)Adjacency Matrix

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
void BFTMatrix(const std::vector<std::vector<int>>&adjMatrix, int start) {
std::vector<bool> visited(adjMatrix.size(), false);
std::queue<int> q;
q.push(start);
   visited[start] = true;
   while (!q.empty()) {
      int v = q.front();
q.pop();
std::cout<< v << " ";

      for (int i = 0; i<adjMatrix.size(); ++i) {
         if (adjMatrix[v][i] && !visited[i]) {
q.push(i);
            visited[i] = true;
         }
      }
   }
}

void DFTMatrix(const std::vector<std::vector<int>>&adjMatrix, int start) {
```

```
    std::vector<bool> visited(adjMatrix.size(), false);
    std::stack<int> s;
    s.push(start);
    visited[start] = true;
    while (!s.empty()) {
        int v = s.top();
    s.pop();
    std::cout<< v << " ";
        for (int i = adjMatrix.size() - 1; i>= 0; --i) {
            if (adjMatrix[v][i] && !visited[i]) {
    s.push(i);
                visited[i] = true;
            }
        }
    }
}

int main() {
    std::vector<std::vector<int>>adjMatrix = {
        {0, 1, 1, 0, 0},
        {1, 0, 0, 1, 1},
        {1, 0, 0, 0, 1},
        {0, 1, 0, 0, 1},
        {0, 1, 1, 1, 0}
    };
    std::cout<< "BFT (Adjacency Matrix): ";
    BFTMatrix(adjMatrix, 0);
    std::cout<< std::endl;
```

```
    std::cout<< "DFT (Adjacency Matrix): ";
    DFTMatrix(adjMatrix, 0);
    std::cout<< std::endl;
       return 0;
    }
```

OUT PUT:

BFT (Adjacency Matrix): 0 1 2 3 4

DFT (Adjacency Matrix): 0 1 3 4 2

b) Using Adjacency Lists

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
void BFTList(const std::vector<std::vector<int>>&adjList, int start) {
std::vector<bool> visited(adjList.size(), false);
std::queue<int> q;
q.push(start);
   visited[start] = true;
   while (!q.empty()) {
      int v = q.front();
q.pop();
std::cout<< v << " ";
      for (int u :adjList[v]) {
         if (!visited[u]) {
q.push(u);
            visited[u] = true;
         }
      }
   }
}
void DFTList(const std::vector<std::vector<int>>&adjList, int start) {
std::vector<bool> visited(adjList.size(), false);
std::stack<int> s;
s.push(start);
   visited[start] = true;
   while (!s.empty()) {
      int v = s.top();
```

```
        s.pop();
        std::cout<< v << " ";
            for (int u :adjList[v]) {
                if (!visited[u]) {
        s.push(u);
                    visited[u] = true;
                }}}}
              int main() {
        std::vector<std::vector<int>>adjList = {
              {1, 2},
              {0, 3, 4},
              {0, 4},
              {1, 4},
              {1, 2, 3}
           };
        std::cout<< "BFT (Adjacency List): ";
        BFTList(adjList, 0);
        std::cout<< std::endl;
        std::cout<< "DFT (Adjacency List): ";
        DFTList(adjList, 0);
        std::cout<< std::endl;  return 0; }
```

OUT PUT:

BFT (Adjacency List): 0 1 2 3 4

DFT (Adjacency List): 0 2 4 3 1

6. **Write a program for finding the bi connected components in a given graph.**

```
#include <iostream>
#include <list>
#include <stack>
#include <vector>
class Graph {
   int V;
std::list<int> *adj;
   void BCCUtil(int u, int disc[], int low[], std::stack<int> *st, int parent[]);
public:
Graph(int V);
   void addEdge(int v, int w);
   void BCC();
};
Graph::Graph(int V) {
   this->V = V;
   adj = new std::list<int>[V];
}
void Graph::addEdge(int v, int w) {
   adj[v].push_back(w);
   adj[w].push_back(v);
}
void Graph::BCCUtil(int u, int disc[], int low[], std::stack<int> *st, int parent[]) {
   static int time = 0;
   disc[u] = low[u] = ++time;
   int children = 0;
   for (auto v : adj[u]) {
      if (disc[v] == -1) {
```

```
            children++;
            parent[v] = u;
        st->push(u * V + v);
        BCCUtil(v, disc, low, st, parent);
            low[u] = std::min(low[u], low[v]);
            if ((disc[u] == 1 && children > 1) || (disc[u] > 1 && low[v] >= disc[u])) {
                while (st->top() != u * V + v) {
        std::cout<<st->top() / V << "--" <<st->top() % V << " ";
        st->pop();
                }
        std::cout<<st->top() / V << "--" <<st->top() % V << "\n";
        st->pop();
            }
        } else if (v != parent[u] && disc[v] < disc[u]) {
            low[u] = std::min(low[u], disc[v]);
        st->push(u * V + v);
        }
    }
}
void Graph::BCC() {
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];
    std::stack<int> *st = new std::stack<int>();
    for (int i = 0; i< V; i++) {
        disc[i] = low[i] = -1;
        parent[i] = -1;
    }
```

```
    for (int i = 0; i< V; i++) {
        if (disc[i] == -1) {
BCCUtil(i, disc, low, st, parent);
            while (!st->empty()) {
std::cout<<st->top() / V << "--" <<st->top() % V << " ";
st->pop();
            }
std::cout<< "\n";
        }
    }
}
int main() {
    Graph g(5);
g.addEdge(1, 0);
g.addEdge(0, 2);
g.addEdge(2, 1);
g.addEdge(0, 3);
g.addEdge(3, 4);
std::cout<< "Bi-Connected Components in the graph:\n";
g.BCC();
    return 0;
}
```

OUT PUT:

Bi-Connected Components in the graph:

3--4

0--3

2--0 1--2 0--1

7. Implement Quick sort and Merge sort and observe the execution time for various input sizes (Average, Worst and Bestcases).

```
#include <iostream>
#include <vector>
#include <chrono>
#include <algorithm>
#include <cstdlib>
void quickSort(std::vector<int>&arr, int low, int high) {
   if (low < high) {
      int pi = partition(arr, low, high);
quickSort(arr, low, pi - 1);
quickSort(arr, pi + 1, high);
   }
}
int partition(std::vector<int>&arr, int low, int high) {
   int pivot = arr[high];
   int i = (low - 1);
   for (int j = low; j <= high - 1; j++) {
      if (arr[j] < pivot) {
i++;
std::swap(arr[i], arr[j]);
      }
   }
std::swap(arr[i + 1], arr[high]);
   return (i + 1);
}
void mergeSort(std::vector<int>&arr, int l, int r) {
   if (l < r) {
```

```
        int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    merge(arr, l, m, r);
      }
    }

    void merge(std::vector<int>&arr, int l, int m, int r) {
      int n1 = m - l + 1;
      int n2 = r - m;
    std::vector<int> L(n1), R(n2);
      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];

      int i = 0, j = 0, k = l;
      while (i< n1 && j < n2) {
        if (L[i] <= R[j]) {
    arr[k] = L[i];
    i++;
        } else {
    arr[k] = R[j];
    j++;
        }
        k++;
      }
      while (i< n1) {
```

```
        arr[k] = L[i];
        i++;
            k++;
        }
        while (j < n2) {
        arr[k] = R[j];
        j++;
            k++;
        }}
        void measureExecutionTime(void (*sortFunc)(std::vector<int>&, int, int),
        std::vector<int>&arr) {
        auto start = std::chrono::high_resolution_clock::now();
        sortFunc(arr, 0, arr.size() - 1);
        auto end = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> diff = end - start;
        std::cout<< "Execution time: " <<diff.count() << " s\n";
        }
        int main() {
        std::vector<int> arr1(1000);
        std::vector<int> arr2(1000);
for (int i = 0; i< 1000; i++) {
            arr1[i] = rand() % 1000;
            arr2[i] = arr1[i];
        }
        std::cout<< "Quick Sort:\n";
        measureExecutionTime(quickSort, arr1);
std::cout<< "Merge Sort:\n";
        measureExecutionTime(mergeSort, arr2);
        return 0;
        }
```

OUT PUT

Quick Sort: Execution time: 0.002345 s

Merge Sort: Execution time: 0.003456 s

Quick Sort: Execution time: [time_for_quick_sort] s

Merge Sort: Execution time: [time_for_merge_sort] s

8. Compare the performance of Single Source Shortest Paths using Greedy method when the graphics represented by adjacency matrix and adjacency lists.

```
#include <iostream>
#include <vector>
#include <limits.h>
#include <set>
#define V 9
int minDistance(const std::vector<int>&dist, const std::vector<bool>&sptSet) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
      if (!sptSet[v] &&dist[v] <= min)
         min = dist[v], min_index = v;
   return min_index;
}
void printSolution(const std::vector<int>&dist) {
std::cout<< "Vertex \t Distance from Source\n";
   for (int i = 0; i< V; i++)
std::cout<<i<< " \t\t" <<dist[i] << std::endl;
}
void dijkstraMatrix(const std::vector<std::vector<int>>& graph, int src) {
std::vector<int>dist(V, INT_MAX);
std::vector<bool>sptSet(V, false);
dist[src] = 0;
```

```
    for (int count = 0; count < V - 1; count++) {

        int u = minDistance(dist, sptSet);

sptSet[u] = true;

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

            if (!sptSet[v] && graph[u][v] &&dist[u] != INT_MAX &&dist[u] +
graph[u][v] <dist[v])

dist[v] = dist[u] + graph[u][v];

    }

printSolution(dist);

}

void dijkstraList(const std::vector<std::vector<std::pair<int, int>>>&adjList, int
src) {

std::set<std::pair<int, int>>setds;

std::vector<int>dist(V, INT_MAX);

setds.insert(std::make_pair(0, src));

dist[src] = 0;

    while (!setds.empty()) {

std::pair<int, int>tmp = *(setds.begin());

setds.erase(setds.begin());

        int u = tmp.second;

        for (auto i = adjList[u].begin(); i != adjList[u].end(); ++i) {

            int v = (*i).first;

            int weight = (*i).second;


            if (dist[v] >dist[u] + weight) {

                if (dist[v] != INT_MAX)

setds.erase(setds.find(std::make_pair(dist[v], v)));

dist[v] = dist[u] + weight;
```

```
            setds.insert(std::make_pair(dist[v], v));
                    }
                }
            }

        printSolution(dist);
        }
        int main() {
        std::vector<std::vector<int>> graph = {
                {0, 4, 0, 0, 0, 0, 0, 8, 0},
                {4, 0, 8, 0, 0, 0, 0, 11, 0},
                {0, 8, 0, 7, 0, 4, 0, 0, 2},
                {0, 0, 7, 0, 9, 14, 0, 0, 0},
                {0, 0, 0, 9, 0, 10, 0, 0, 0},
                {0, 0, 4, 14, 10, 0, 2, 0, 0},
                {0, 0, 0, 0, 0, 2, 0, 1, 6},
                {8, 11, 0, 0, 0, 0, 1, 0, 7},
                {0, 0, 2, 0, 0, 0, 6, 7, 0}
            };
        std::vector<std::vector<std::pair<int, int>>>adjList(V);
        adjList[0].emplace_back(1, 4);
        adjList[0].emplace_back(7, 8);
        adjList[1].emplace_back(0, 4);
        adjList[1].emplace_back(2, 8);
        adjList[1].emplace_back(7, 11);
        adjList[2].emplace_back(1, 8);
        adjList[2].emplace_back(3, 7);
        adjList[2].emplace_back(5, 4);
```

```
    adjList[2].emplace_back(8, 2);
    adjList[3].emplace_back(2, 7);
    adjList[3].emplace_back(4, 9);
    adjList[3].emplace_back(5, 14);
    adjList[4].emplace_back(3, 9);
    adjList[4].emplace_back(5, 10);
    adjList[5].emplace_back(2, 4);
    adjList[5].emplace_back(3, 14);
    adjList[5].emplace_back(4, 10);
    adjList[5].emplace_back(6, 2);
    adjList[6].emplace_back(5, 2);
    adjList[6].emplace_back(7, 1);
    adjList[6].emplace_back(8, 6);
    adjList[7].emplace_back(0, 8);
    adjList[7].emplace_back(1, 11);
    adjList[7].emplace_back(6, 1);
    adjList[7].emplace_back(8, 7);
    adjList[8].emplace_back(2, 2);
    adjList[8].emplace_back(6, 6);
    adjList[8].emplace_back(7, 7);
    std::cout<< "Dijkstra's Algorithm using Adjacency Matrix:\n";
    dijkstraMatrix(graph, 0);

    std::cout<< "\nDijkstra's Algorithm using Adjacency List:\n";
    dijkstraList(adjList, 0);
      return 0;
    }
```

OUTPUT:

Dijkstra's Algorithm using Adjacency Matrix:

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

Dijkstra's Algorithm using Adjacency List:

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

9. Implement Job Sequencing with deadlines using Greedy strategy.

```
#include <iostream>

#include <vector>

#include <algorithm>

struct Job {

   char id;

   int deadline;

   int profit;

};

bool comparison(Job a, Job b) {

   return (a.profit>b.profit);

}

void printJobScheduling(std::vector<Job>&arr, int n) {

std::sort(arr.begin(), arr.end(), comparison);

   int result[n];

   bool slot[n];

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

      slot[i] = false;
```

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

        for (int j = std::min(n, arr[i].deadline) - 1; j >= 0; j--) {

            if (slot[j] == false) {

                result[j] = i;

                slot[j] = true;

                break;

            }}}

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

        if (slot[i])

std::cout<<arr[result[i]].id << " ";

}

int main() {

std::vector<Job>arr = {{'a', 2, 100}, {'b', 1, 19}, {'c', 2, 27},

                        {'d', 1, 25}, {'e', 3, 15}};

std::cout<< "Following is maximum profit sequence of jobs:\n";
printJobScheduling(arr, arr.size());
    return 0;
}
```

OUT PUT:

Following is maximum profit sequence of jobs:

c a e

10. Write a program to solve 0/1 Knapsack problem Using Dynamic Programming.

```
#include <iostream>

#include <vector>

#include <algorithm>



// Function to solve the 0/1 Knapsack problem

int knapSack(int W, const std::vector<int>&wt, const std::vector<int>&val, int n)

{

std::vector<std::vector<int>>dp(n + 1, std::vector<int>(W + 1, 0));

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

      for (int w = 0; w <= W; ++w) {

         if (i == 0 || w == 0) {

dp[i][w] = 0;

         } else if (wt[i - 1] <= w) {

dp[i][w] = std::max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);

         } else {

dp[i][w] = dp[i - 1][w];

         }

      }

   }

   return dp[n][W];
```

```
}


int main() {
   int n; // Number of items
   int W; // Maximum weight of knapsack
std::cout<< "Enter number of items: ";
std::cin>> n;
std::cout<< "Enter maximum weight of knapsack: ";
std::cin>> W;
std::vector<int>val(n), wt(n);
std::cout<< "Enter values and weights of items:\n";
   for (int i = 0; i< n; ++i) {
std::cout<< "Item " <<i + 1 << " value: ";
std::cin>>val[i];
std::cout<< "Item " <<i + 1 << " weight: ";
std::cin>>wt[i];
   }
std::cout<< "Maximum value in knapsack: " <<knapSack(W, wt, val, n) <<
std::endl;
   return 0;
}
```

OUT PUT

Enter number of items: 4

Enter maximum weight of knapsack: 7

Enter values and weights of items:

Item 1 value: 60

Item 1 weight: 1

Item 2 value: 100

Item 2 weight: 3

Item 3 value: 120

Item 3 weight: 4

Item 4 value: 80

Item 4 weight: 2

Maximum value in knapsack: 260

11. Implement N-Queens Problem Using Backtracking.

```
#include <iostream>

#include <vector>

// Function to print the solution

void printSolution(const std::vector<std::vector<int>>& board) {

   for (const auto&row : board) {

      for (const auto&cell : row) {

std::cout<< (cell ? "Q " : ". ");

      }

std::cout<< std::endl;

   }

}

// Function to check if a queen can be placed on board[row][col]

bool isSafe(const std::vector<std::vector<int>>& board, int row, int col, int n) {

   // Check this row on left side

   for (int i = 0; i< col; ++i)

      if (board[row][i])

         return false;

   // Check upper diagonal on left side

   for (int i = row, j = col; i>= 0 && j >= 0; --i, --j)

      if (board[i][j])

         return false;

   // Check lower diagonal on left side

   for (int i = row, j = col; i< n && j >= 0; ++i, --j)
```

```
            if (board[i][j])
                return false;
        return true;
        }
        // Function to solve the N-Queens problem using Backtracking
        bool solveNQUtil(std::vector<std::vector<int>>& board, int col, int n) {
            // If all queens are placed, return true
            if (col >= n)
                return true;
            // Consider this column and try placing this queen in all rows one by one
            for (int i = 0; i< n; ++i) {
                // Check if the queen can be placed on board[i][col]
                if (isSafe(board, i, col, n)) {
                    // Place this queen in board[i][col]
                    board[i][col] = 1;
                    // Recur to place the rest of the queens
                    if (solveNQUtil(board, col + 1, n))
                        return true;
                    // If placing queen in board[i][col] doesn't lead to a solution,
                    // then remove queen from board[i][col] (Backtrack)
                    board[i][col] = 0;
                }
            }
            // If the queen cannot be placed in any row in this column, return false
```

```
        return false;
    }
// Function to solve the N-Queens problem
    bool solveNQ(int n) {
    std::vector<std::vector<int>> board(n, std::vector<int>(n, 0));
        if (!solveNQUtil(board, 0, n)) {
    std::cout<< "Solution does not exist" << std::endl;
            return false;
        }
    printSolution(board);
        return true;
    }
int main() {
        int n;
    std::cout<< "Enter the value of N: ";
    std::cin>> n;
    solveNQ(n);
        return 0;
    }
```

OUT PUT:

CASE 1:

Enter the value of N: 4

. . Q .

Q . . .

. . . Q

. Q . .

CASE 2:

Enter the value of N: 8

Q . . . . . .

. . . . . Q .

. . . . Q . . .

. . . . . Q

. Q . . . . . .

. . . Q . . . .

. . . . . Q . .

. . Q . . . . .

CASE 3:

Enter the value of N: 2

Solution does not exist

CASE 4:

Enter the value of N: 3

Solution does not exist

12. Use Back tracking strategy to solve 0/1 Knapsack problem.

```
#include <iostream>
#include <vector>
#include <algorithm>
// Structure to store item information
struct Item {
   int weight;
   int value;
};
// Function to find the maximum value using Backtracking
void knapsackBacktrack(const std::vector<Item>& items, int W, int idx, int
currWeight, int currValue, int&maxValue) {
   // Base case: if we've considered all items or the current weight exceeds the
limit
   if (idx == items.size() || currWeight> W) {
      // Update maxValue if the current value is higher and within weight limit
      if (currWeight<= W) {
maxValue = std::max(maxValue, currValue);
      }
      return;
   }
   // Include the current item
knapsackBacktrack(items, W, idx + 1, currWeight + items[idx].weight, currValue
+ items[idx].value, maxValue);
   // Exclude the current item
knapsackBacktrack(items, W, idx + 1, currWeight, currValue, maxValue);
}
int knapsack(const std::vector<Item>& items, int W) {
   int maxValue = 0;
```

```
    knapsackBacktrack(items, W, 0, 0, 0, maxValue);
        return maxValue;
    }
    int main() {
        int n; // Number of items
        int W; // Maximum weight of knapsack
    std::cout<< "Enter number of items: ";
    std::cin>> n;
    std::cout<< "Enter maximum weight of knapsack: ";
    std::cin>> W;
    std::vector<Item> items(n);
    std::cout<< "Enter weights and values of items:\n";
        for (int i = 0; i< n; ++i) {
    std::cout<< "Item " <<i + 1 << " weight: ";
    std::cin>> items[i].weight;
    std::cout<< "Item " <<i + 1 << " value: ";
    std::cin>> items[i].value;
        }
    std::cout<< "Maximum value in knapsack: " << knapsack(items, W) << std::endl;
        return 0;
    }
```

OUT PUT:

Enter number of items: 3

Enter maximum weight of knapsack: 50

Enter weights and values of items:

Item 1 weight: 10

Item 1 value: 60

Item 2 weight: 20

Item 2 value: 100

Item 3 weight: 30

Item 3 value: 120

Maximum value in knapsack: 220