//8 rooks

```
#include<iostream>
#include<vector>
using namespace std;
#define N 8
int count=0;
void printSolution(vector<int> rooks)
{
   for(int i=0;i<N;i++)
   {
      for (int j=0;j<N;j++)
      {
         if(rooks[i]==j)
         {
            cout<<"R ";
         }
         else
         {
            cout<<"- ";
         }
      }
      cout<<endl;
   }
   cout<<"\n";
   count++;
}
bool isSafe(vector<int> rooks,int row,int col)
{
   for (int i=0;i<row;i++)
   {
      if(rooks[i]==col)
      {
         return false;
      }
   }
   return true;
}
void solveNRooks(vector<int> rooks,int row)
{
   if(row>=N)
   {
      printSolution(rooks);
      return;
   }
   for (int col=0;col<N;col++)
   {
      if(isSafe(rooks,row,col))
      {
         rooks[row]=col;
         solveNRooks(rooks,row+1);
      }
   }
}
int main()
{
   vector<int> rooks(N, -1);
```

```
    solveNRooks(rooks,0);
    cout<<count;
    return 0;
}
```

//Activity selevtion

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
struct Activity {
    int id;
    int start;
    int finish;
};
void swap(Activity &a, Activity &b)
{
    Activity temp=a;
    a=b;
    b=temp;
}
// Partition function for QuickSort
int partition(vector<Activity> &arr,int low,int high)
{
    Activity pivot=arr[high];
    int i=low - 1;
    for (int j=low;j<high;j++)
    {
        if(arr[j].finish <= pivot.finish) // Sorting based on finish time
        {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1],arr[high]);
    return i+1;
}
// QuickSort to sort activities based on finish time
void quickSort(vector<Activity> &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 main()
{
    int n;
    cout<<"Enter number of activities: ";
    cin>>n;
    vector<Activity> activities(n);
    for (int i=0; i<n;i++)
    {
        activities[i].id=i+1;
```

```
    cout<<"Enter start and finish times for activity "<<activities[i].id <<": ";
    cin>>activities[i].start>>activities[i].finish;
  }
  // Sorting activities by their finish times (Greedy approach)
  quickSort(activities, 0, n - 1);
  int count=0;
  int lastFinish=0;
  cout << "\nSelected activities:\n";
  if (n > 0)
  {
    cout<<"Activity "<<activities[0].id << ": (" <<activities[0].start << ", "<<activities[0].finish<<")\n";
    count = 1;
    lastFinish = activities[0].finish;
  }
  // Selecting activities using a greedy approach
  for (int i = 1;i<n;i++)
  {
    if(activities[i].start >= lastFinish) // Selecting non-overlapping activities
    {
      cout<<"Activity "<<activities[i].id << ": ("<<activities[i].start << ", "<<activities[i].finish<< ")\n";
      count++;
      lastFinish = activities[i].finish;
    }
  }
  cout<<"Total activities selected: "<<count<< "\n";
  return 0;
}
```

//coin selection

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<climits>
using namespace std;

// Function to compute the minimum coins using a greedy approach
int greedyCoinChange(vector<int>& denominations, int target) {
  sort(denominations.rbegin(), denominations.rend()); // Sorting in descending order
  int coinCount = 0;
  for(int coin : denominations)
  {
    while(target>=coin)
    {
      target=target-coin;
      coinCount++;
    }
  }
  return coinCount;
}

// Function to compute the minimum coins using a brute-force dynamic programming approach
int bruteForceCoinChange(vector<int>& denominations, int target)
{
  vector<int> dp(target + 1, INT_MAX);
  dp[0]=0;
```

```
    for(int i = 1; i<=target;i++)
    {
        for(int coin : denominations)
        {
            if(i >=coin && dp[i - coin]!=INT_MAX)
            {
                dp[i]=min(dp[i], dp[i - coin] + 1);
            }
        }
    }
    return (dp[target] == INT_MAX) ? -1 : dp[target]; // If no solution, return -1
}
int main()
{
    int numDenominations, targetValue;

    cout<<"Enter the number of denominations: ";
    cin>>numDenominations;

    vector<int> denominations(numDenominations);
    cout<<"Enter the denominations: ";
    for (int &coin : denominations)
    {
        cin>>coin;
    }
    cout<<"Enter the target value: ";
    cin>>targetValue;

    // Compute results using both approaches
    int greedyResult=greedyCoinChange(denominations, targetValue);
    int bruteForceResult=bruteForceCoinChange(denominations, targetValue);

    cout<<"\nGreedy Strategy Total Coins: "<<greedyResult << endl;
    cout<<"Brute Force Minimum Coins: "<<bruteForceResult << endl;

    // Check if greedy approach gives an optimal solution
    if(greedyResult == bruteForceResult)
    {
        cout<<"Yes, the greedy strategy provides the optimal result for this problem.\n";
    }
    else
    {
        cout<<"No, the greedy strategy does not provide the optimal result for this problem.\n";
    }
    return 0;
}
```

//fractional knapsack

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
struct Item {
    int weight;
```

```
    int value;
    double ratio;
    int index;
};
void swap(Item &a, Item &b)
{
    Item temp=a;
    a=b;
    b=temp;
}
// Partition function for QuickSort
int partition(vector<Item> &arr, int low, int high)
{
    Item pivot=arr[high];
    int i=low - 1;
    for (int j = low;j<high;j++)
    {
        if (arr[j].ratio >= pivot.ratio)
        {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1],arr[high]);
    return i + 1;
}
// QuickSort function to sort items by ratio (Greedy Step)
void quickSort(vector<Item> &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 main()
{
    int n;
    cout<<"Enter number of items: ";
    cin>>n;
    vector<Item> items(n);
    for (int i = 0;i<n;i++)
    {
        cout<<"Enter weight and value for item "<<i + 1<<": ";
        cin>>items[i].weight>>items[i].value;
        items[i].ratio=(double)items[i].value / items[i].weight; // Compute value-to-weight ratio
        items[i].index = i;
    }
    int capacity;
    cout<<"Enter knapsack capacity: ";
    cin>>capacity;
    quickSort(items,0,n - 1); // Sort items by highest value/weight ratio (Greedy Choice)
    double totalValue = 0.0;
    vector<pair<int, double>> fractions;
    // Pick items greedily based on sorted order
    for (int i = 0;i < n && capacity > 0;i++)
```

```
    {
        if (items[i].weight <= capacity)
        {
            capacity=capacity-items[i].weight;
            totalValue=totalValue+items[i].value;
            fractions.push_back({items[i].index, 1.0}); // Fully taken
        }
        else
        {
            double fraction = (double)capacity / items[i].weight;
            totalValue=totalValue+items[i].ratio * capacity;
            fractions.push_back({items[i].index, fraction}); // Partially taken
            capacity = 0;
        }
    }
    cout<<"Maximum Profit in knapsack: "<<totalValue << endl;
    cout<<"Item Fractions:"<<endl;
    for (const auto &f : fractions)
    {
        cout<<"Item "<< f.first<<": "<<f.second << endl;
    }
    return 0;
}
```

//als

```
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

int main()
{
    int stations;
    cout<<"Enter number of stations: ";
    cin>>stations;

    vector<int> line1(stations), line2(stations);
    vector<int> transfer1(stations - 1), transfer2(stations - 1);
    vector<int> entry(2), exit(2);

    cout<<"Enter station processing times for line 1: ";
    for (int i = 0;i < stations; i++)
    {
        cin>>line1[i];
    }

    cout<<"Enter station processing times for line 2: ";
    for (int i = 0; i < stations; i++)
    {
        cin>>line2[i];
    }

    cout<<"Enter transfer times from line 1 to line 2: ";
    for (int i = 0; i < stations - 1; i++)
    {
```

```
        cin>>transfer1[i];
    }

    cout<<"Enter transfer times from line 2 to line 1: ";
    for (int i = 0; i < stations - 1; i++)
    {
        cin>>transfer2[i];
    }

    cout<<"Enter entry times for both lines: ";
    cin>>entry[0] >> entry[1];

    cout<<"Enter exit times for both lines: ";
    cin>>exit[0] >> exit[1];

    vector<int> time1(stations), time2(stations);

    time1[0]=entry[0] + line1[0];
    time2[0]=entry[1] + line2[0];

    for (int i = 1; i < stations; i++)
    {
        time1[i]=min(time1[i - 1] + line1[i], time2[i - 1] + transfer2[i - 1] + line1[i]);
        time2[i]=min(time2[i - 1] + line2[i], time1[i - 1] + transfer1[i - 1] + line2[i]);
    }

    int minTime = min(time1[stations - 1] + exit[0], time2[stations - 1] + exit[1]);
    cout<<"Minimum time to exit: "<<minTime<<endl;

    // Reconstructing the path
    vector<int> path(stations);
    if (time1[stations - 1] + exit[0] < time2[stations - 1] + exit[1]) {
        path[stations - 1]=1;
    }
    else
    {
        path[stations - 1]=2;
    }
    for (int i = stations - 2; i >= 0; i--)
    {
        if (path[i + 1] == 1)
        {
            path[i] = (time1[i] <= time2[i] + transfer2[i]) ? 1 : 2;
        }
        else
        {
            path[i] = (time2[i]<= time1[i] + transfer1[i]) ? 2 : 1;
        }
    }
    cout<<"Optimal path: ";
    for (int i = 0; i < stations; i++)
    {
        cout<<path[i] << " ";
    }
    cout<<endl;
    return 0;
}
```

//floydd warshall

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;
#define INF 1000000  // Large integer value to represent infinity

void floydWarshall(vector<vector<int>> &graph)
{
   int V = graph.size();
   vector<vector<int>> dist = graph;

   // Compute shortest paths
   for (int k = 0; k < V; k++)
   {
      for (int i = 0; i < V; i++)
      {
         for (int j = 0; j < V; j++)
         {
            if (dist[i][k] != INF && dist[k][j] != INF)
               dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
         }
      }
   }

   // Print shortest distance matrix
   cout << "Shortest distance matrix obtained from Floyd-Warshall:\n";
   for (int i = 0; i < V; i++)
   {
      for (int j = 0; j < V; j++)
      {
         if (dist[i][j] == INF)
            cout << "INF ";
         else
            cout << dist[i][j] << " ";
      }
      cout << endl;
   }

   // Print shortest distances between each pair of vertices
   cout << "\nShortest distances between each pair of vertices:\n";
   for (int i = 0; i < V; i++)
   {
      for (int j = 0; j < V; j++)
      {
         if (i != j) {
            if (dist[i][j] == INF)
               cout << "No path from vertex " << i << " to vertex " << j << endl;
            else
               cout << "Shortest distance from vertex " << i << " to vertex " << j << " is " << dist[i][j] <<
endl;
         }
      }
   }
}
```

```
int main() {
   vector<vector<int>> graph = {  //change input
      {0, 3, INF, 7},
      {8, 0, 2, INF},
      {5, INF, 0, 1},
      {2, INF, INF, 0}
   };

   floydWarshall(graph);
   return 0;
}
```

//primss

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

#define V 5

int minKey(vector<int> &key, vector<bool> &mstSet) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
      if (!mstSet[v] && key[v] < min)
         min = key[v], min_index = v;
   return min_index;
}

void primMST(int graph[V][V]) {
   vector<int> parent(V, -1), key(V, INT_MAX);
   vector<bool> mstSet(V, false);
   key[0] = 0;

   for (int count = 0; count < V - 1; count++) {
      int u = minKey(key, mstSet);
      mstSet[u] = true;

      for (int v = 0; v < V; v++)
         if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v])
            parent[v] = u, key[v] = graph[u][v];
   }

   cout << "Edge\tWeight\n";
   for (int i = 1; i < V; i++)
      cout << parent[i] << " - " << i << " \t" << graph[i][parent[i]] << endl;
}

int main() {
   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}
   };
```

```
    primMST(graph);
    return 0;
}
```

//clique

```
#include <iostream>
#include <vector>

#define MAX 100

using namespace std;

bool check_clique(int adj_matrix[MAX][MAX], const vector<int>& nodes, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = i + 1; j < size; j++) {
            if (adj_matrix[nodes[i]][nodes[j]] == 0)
                return false;
        }
    }
    return true;
}

int main() {
    int vertices, edges, node1, node2;
    int adj_matrix[MAX][MAX] = {0};

    cout << "Enter the number of vertices: ";
    cin >> vertices;

    cout << "Enter the number of edges: ";
    cin >> edges;

    cout << "Enter edges (node1 node2):\n";
    for (int i = 0; i < edges; i++) {
        cin >> node1 >> node2;
        adj_matrix[node1][node2] = 1;
        adj_matrix[node2][node1] = 1;
    }

    int subset_size;
    cout << "Enter the size of the candidate solution: ";
    cin >> subset_size;

    vector<int> nodes(subset_size);
    cout << "Enter " << subset_size << " vertices: ";
    for (int i = 0; i < subset_size; i++) {
        cin >> nodes[i];
    }

    if (check_clique(adj_matrix, nodes, subset_size)) {
        cout << "True" << endl;
    } else {
        cout << "The given candidate solution is not actually a solution" << endl;
    }
```

```
    return 0;
}
```

//vertex cover normal

```
#include <iostream>
#include <vector>
using namespace std;

bool isVertexCover(const vector<pair<int, int>>& edges, const vector<int>& candidateSet) {
    for (const auto& edge : edges) {
        bool u_found = false, v_found = false;
        for (int v : candidateSet) {
            if (v == edge.first) u_found = true;
            if (v == edge.second) v_found = true;
        }

        if (!u_found && !v_found) {
            // Neither endpoint is in the candidate set
            return false;
        }
    }
    return true;
}

int main() {
    int v, e;
    cout << "Enter number of vertices and edges: ";
    cin >> v >> e;

    vector<pair<int, int>> edges(e);
    cout << "Enter edges (u v) format:\n";
    for (int i = 0; i < e; ++i) {
        cin >> edges[i].first >> edges[i].second;
    }

    int s;
    cout << "Enter size of candidate vertex cover: ";
    cin >> s;

    vector<int> candidateSet(s);
    cout << "Enter candidate vertex cover: ";
    for (int i = 0; i < s; ++i) {
        cin >> candidateSet[i];
    }

    if (isVertexCover(edges, candidateSet)) {
        cout << "True: The given set is a vertex cover." << endl;
    } else {
        cout << "False: The given set is not a vertex cover." << endl;
    }

    return 0;
}
```

//subset sum with candidate subset

```
#include <iostream>
#include <vector>
using namespace std;

int isSubsetSum(const vector<int> &set, const vector<int> &subset, int target) {
   int sum = 0;
   for (int i = 0; i < subset.size(); i++) {
      bool found = false;
      for (int j = 0; j < set.size(); j++) {
         if (subset[i] == set[j]) {
            found = true;
            break;
         }
      }
      if (!found)
         return -1;
      sum += subset[i];
   }
   return (sum == target) ? 1 : 0;
}

int main() {
   int n, target, k;
   cout << "Enter the number of elements in the set: ";
   cin >> n;

   vector<int> set(n);
   cout << "Enter the set elements:\n";
   for (int i = 0; i < n; i++) {
      cin >> set[i];
   }

   cout << "Enter the target sum: ";
   cin >> target;

   cout << "Enter the size of the candidate subset: ";
   cin >> k;

   vector<int> subset(k);
   cout << "Enter the subset elements:\n";
   for (int i = 0; i < k; i++) {
      cin >> subset[i];
   }

   int result = isSubsetSum(set, subset, target);
   if (result == 1) {
      cout << "True: The given subset sums to the target value.\n";
   } else if (result == 0) {
      cout << "False: The given subset does not sum to the target value.\n";
   } else {
      cout << "Error: The subset contains elements not present in the original set.\n";
   }

   return 0;
```

```
}
```

//vector cover with candidate subset

```
#include <iostream>
#include <vector>
using namespace std;

bool visited[100]; // assuming max 100 vertices, similar to MAX_VERTICES in C

void findVertexCover(const vector<pair<int, int>> &edges, int V) {
   // Initialize visited array
   for (int i = 0; i < V; i++) {
      visited[i] = false;
   }

   cout << "Approximate Vertex Cover: ";
   for (const auto &edge : edges) {
      int u = edge.first;
      int v = edge.second;

      if (!visited[u] && !visited[v]) {
         visited[u] = true;
         visited[v] = true;
         cout << u << " " << v << " ";
      }
   }
   cout << endl;
}

int main() {
   int V, E;
   cout << "Enter number of vertices: ";
   cin >> V;
   cout << "Enter number of edges: ";
   cin >> E;

   vector<pair<int, int>> edges(E);
   cout << "Enter the edges in (u v) format:\n";
   for (int i = 0; i < E; i++) {
      cin >> edges[i].first >> edges[i].second;
   }

   findVertexCover(edges, V);

   return 0;
}
```