Linear Search:

C/C++
```
#include <iostream>

using namespace std;

void linearSearch(int arr[], int a){
	int i;
	int n = sizeof(arr)/sizeof(arr[0]);
	for(i = 0; i<n ; i++){
		if(arr[i] == a ){
			cout<<"element found at " << i+1;
			break;
		}
	}
	return;
}

int main(){
	int arr[5] = {10, 20, 30, 40, 50};
	linearSearch(arr,20);
	return 0;
}
```

Binary Search

C/C++
```
#include <iostream>

using namespace std;

int binary(int arr[],int a, int left, int right){
	if(left<right){
		int mid = (left+right)/2;
		if(arr[mid]==a){
			return mid;
		}
		else if(arr[mid]>a){
			return binary(arr,a,left,mid-1);
		}else{
```

```
                        return binary(arr,a,mid+1,right);
                }
        }
        return -1;

}

int main(){
        int arr[5] = {10, 20, 30, 40, 50};
        int x = binary(arr,90,0,4);
        cout << x;
        return 0;
}
```

Quick Sort

C/C++

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

int partition(int arr[], int low, int high){
        int pivot = arr[low];
        int i = low;
        int j = high;

        while(i<j){
                while(arr[i]<=pivot && i<=high-1){
                        i++;
                }while(arr[j]>pivot && j>=low+1){
                        j--;
                }if(i<j) swap(arr[i],arr[j]);

        }swap(arr[low],arr[j]);
        return j;
}
void quickSort(int arr[], int left, int right){
        if(left<right){
                int pivot = partition(arr,left,right);
                quickSort(arr, left,pivot-1);
                quickSort(arr, pivot+1, right);
        }
}
```

```
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, size - 1);
    printArray(arr, size);
    return 0;
}
```

Merge Sort

C/C++

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

void merge(int arr[], int left, int mid, int right){
	int n1 = mid - left +1;
	int n2 = right - mid;
	vector<int> L(n1), R(n2);
	int i, j;
	for(i = 0;i<n1;i++){
		L[i] = arr[left+i];
	}
	for(j = 0;j<n2;j++){
		R[j] = arr[mid+1+j];
	}
	i = 0, j = 0;
	int k= left;
	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 mergeSort(int arr[], int left, int right){
       if(left<right){
             int mid = (left + right)/2;
             mergeSort(arr,left,mid);
             mergeSort(arr,mid+1,right);
             merge(arr,left,mid,right);
       }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    mergeSort(arr, 0, size - 1);
    printArray(arr, size);
    return 0;
}
```

Dijkstra's algorithm

C/C++

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

#define V 9

int minDistance(const vector<int>& dist, const vector<bool>& visited) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (!visited[v] && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

void printSolution(const vector<int>& dist) {
    cout << "Vertex \t\t Distance from Source\n";
    for (int i = 0; i < V; i++)
        cout << i << " \t\t " << dist[i] << endl;
}

void dijkstra(int graph[V][V], int src) {
    vector<int> dist(V, INT_MAX);
    vector<bool> visited(V, false);

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, visited);

        visited[u] = true;

        for (int v = 0; v < V; v++)
            if (!visited[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    printSolution(dist);
}

int main() {
    int graph[V][V] = {
        { 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 }
    };

    dijkstra(graph, 0);

    return 0;
}
```

N-Queens

C/C++

```
#include <stdbool.h>
#include <stdio.h>
#define N 4

void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i][j])
                printf("Q ");
            else
                printf(". ");
        }
        printf("\n");
    }
}
```

```
bool isSafe(int board[N][N], int row, int col) {
    int i, j;


    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;


    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;


    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}


bool solveNQUtil(int board[N][N], int col) {

    if (col >= N)
        return true;


    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {

            board[i][col] = 1;
            if (solveNQUtil(board, col + 1))
                return true;
            board[i][col] = 0;
        }
    }
    return false;
}

bool solveNQ() {
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist\n");
        return false;
    }

    printSolution(board);
    return true;
}
int main() {
    solveNQ();
    return 0;
}
```

Optimal merge pattern

C/C++

```
#include<bits/stdc++.h>

using namespace std;
#define int long long

signed main(){
       int n;
        cin>>n;
        vector<int> a(n);
        for(int i = 0; i<n; i++){
              cin >> a[i];
        }
        priority_queue<int,vector<int>,greater<int> > minheap;
        for(int i = 0; i<n; i++){
              minheap.push(a[i]);
        }

        int ans = 0;

        while(minheap.size()>1){
              int e1 = minheap.top();
              minheap.pop();
              int e2 = minheap.top();
```

```
             minheap.pop();
             ans += e1+e2;
             minheap.push(e1+e2);
        }
        cout<< ans;
       return 0;
}
```

Flyod warshall

```
C/C++
#include<bits/stdc++.h>

using namespace std;

const int INF = 1e9;

int main() {
    vector<vector<int> > graph = {{0, 5, INF, 10},
                                 {INF, 0, 3, INF},
                                 {INF, INF, 0, 1},
                                 {INF, INF, INF, 0}};
    int n = graph.size();
    vector<vector<int>> dist = graph;
    int i, j, k;


    for (k = 0; k < n; k++) {
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                if (dist[i][j] > dist[i][k] + dist[k][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    // Print the shortest path matrix
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (dist[i][j] == INF) {
                cout << "INF ";
            } else {
                cout << dist[i][j] << " ";
            }
        }
        cout << "\n";
    }

    return 0;
}
```

Fractional knapsack

C/C++

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


struct item {
    double value, weight, valuePerWeight;
};


bool compare(item i1, item i2) {
    return i1.valuePerWeight > i2.valuePerWeight;
}


signed main() {
    int n; cin >> n;


    vector<item> items;
    for(int i=0; i<n; i++) {
        double v,w;
        cin >> v >> w;


        items.push_back({v,w,v/w});
    }
    double W; cin >> W;


    sort(items.begin(), items.end(), compare);


    int ans = 0;
    for(int i=0; i<n; i++) {
        if(W >= items[i].weight) {
            W -= items[i].weight;
            ans += items[i].value;
        }
        else {
            ans += W * items[i].valuePerWeight;
            W = 0;
            break;
        }
    }
    cout << ans << endl;
    return 0;
}
```

0/1 Knapsack

C/C++

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

int knapSack(int W, int wt[], int val[], int n)
{


    if (n == 0 || W == 0)
        return 0;

    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);
    else
        return max(knapSack(W, wt, val, n - 1),
         val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1));
}


int main()
{
    int profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 10 };
    int W = 20;
    int n = sizeof(profit) / sizeof(profit[0]);
    cout << "max weight is"<<knapSack(W, weight, profit, n);
    return 0;
}
```

Prim’s algorithm

C/C++

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

using namespace std;

int minKey(const vector<int> &key, const vector<bool> &mstSet, int V)
{
    int min = INT_MAX, minIndex;

    for (int v = 0; v < V; v++)
    {
        if (!mstSet[v] && key[v] < min)
        {
            min = key[v];
            minIndex = v;
        }
    }
    return minIndex;
}
```

```cpp
void printMST(const vector<int> &parent, const vector<vector<int>> &graph, int
V)
{
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
    {
        cout << parent[i] << " - " << i << "\t" << graph[i][parent[i]] << "\n";
    }
}

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

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

        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];
            }
        }
    }

    printMST(parent, graph, V);
}

int main()
{
    int V = 5;
    vector<vector<int>> graph = {
        {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, V);

    return 0;
}
```

Kruskal's algorithm

C/C++

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

using namespace std;

struct Edge
{
    int u, v, weight;
};

class DisjointSet
{
public:
    vector<int> parent;

    DisjointSet(int n)
    {
        parent.resize(n);
        for (int i = 0; i < n; i++)
        {
            parent[i] = i;
        }
    }

    int find(int x)
    {
        if (parent[x] != x)
        {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void unionSets(int x, int y)
    {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY)
        {
            parent[rootX] = rootY;
        }
    }
};

vector<Edge> kruskal(int n, vector<Edge> &edges)
{
    vector<Edge> result;
    DisjointSet ds(n);

    sort(edges.begin(), edges.end(), [](Edge &a, Edge &b)
         { return a.weight < b.weight; });

    for (Edge &edge : edges)
    {
        int u = edge.u;
        int v = edge.v;

        if (ds.find(u) != ds.find(v))
        {
            ds.unionSets(u, v);
            result.push_back(edge);
        }
    }

    return result;
}

int main()
{
    int n = 6;
    int m = 9;

    // Define the edges (u, v, weight)
    vector<Edge> edges = {
        {0, 1, 4},
        {0, 2, 3},
        {1, 2, 1},
        {1, 3, 2},
        {2, 3, 4},
        {2, 4, 5},
        {3, 4, 6},
        {3, 5, 7},
        {4, 5, 4}};

    vector<Edge> mst = kruskal(n, edges);

    cout << "Minimum Spanning Tree (MST):\n";
    for (Edge &e : mst)
    {
        cout << e.u << " - " << e.v << " : " << e.weight << endl;
    }

    return 0;
}
```