

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.



LAB REPORT

on

Analysis and design of Algorithms

Submitted by

SARIM ALI (1BM23CS304)



B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019



CERTIFICATE

This is to certify that the Lab work entitled “Analysis and design of algorithm– 23CS4PCADA” carried out by SARIM ALI (1BM23CS304), who is bonafide student of B. M. S. College of Engineering. It is in partial fulfillment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year 2024-2025. The Lab report has been approved as it satisfies the academic requirements in respect of Analysis and design of Algorithms - (23CS4PCADA) work prescribed for the said degree.

Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr.Kavitha Sooda
Professor and HOD
Department of CSE
BMSCE, Bengaluru

Index Sheet

Sl. No.	Experiment Title	Page No.
1.	Write program to obtain the Topological ordering of vertices in a given digraph.	4-6
2.	Sort a given set of N integer elements using Merge Sort technique and compute its time taken.	7-9
3.	Sort a given set of N integer elements using Quick Sort technique and compute its time taken.	10-12
4.	a. Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm. b. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.	13-20
5.	From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.	21-23
6.	Implement Johnson Trotter algorithm to generate permutations.	24-27
7.	Implement fractional Knapsack problem using Greedy technique.	28-30
8.	Implement 0/1 Knapsack problem using dynamic programming.	31-33
9.	Sort a given set of N integer elements using Heap Sort technique and compute its time taken.	34-36
10.	Implement All Pair Shortest paths problem using Floyd's algorithm.	37-40
11.	Implement "N-Queens Problem" using Backtracking.	41-43

1. Write program to obtain the Topological ordering of vertices in a given digraph.

```
#include<stdio.h>

int a[10][10],n,t[10],indegree[10];

int stack[10],top=-1;

void computeIndegree(int,int[][10]);

void tps_SourceRemoval(int,int[][10]);

int main(){

printf("Enter the no. of nodes: ");

scanf("%d",&n);

int i,j;

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

for(j=0;j<n;j++){

scanf("%d",&a[i][j]);

}

}

computeIndegree(n,a);

tps_SourceRemoval(n,a);

printf("Solution:");

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

printf("%d ",t[i]);

}

return 0;

}

void computeIndegree(int n,int a[][10]){

int i,j,sum=0;

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

```

sum=0;
for(j=0;j<n;j++){
    sum=sum+a[j][i];
}
indegree[i]=sum;
}
}

```

```

void tps_SourceRemoval(int n,int a[][10]){
    int i,j,v;
    for(i=0;i<n;i++){
        if(indegree[i]==0){
            stack[++top]=i;
        }
    }
    int k=0;
    while(top!=-1){
        v=stack[top--];
        t[k++]=v;
        for(i=0;i<n;i++){
            if(a[v][i]!=0){
                indegree[i]=indegree[i]-1;
                if(indegree[i]==0){
                    stack[++top]=i;
                }
            }
        }
    }
}

```

```
    }  
}  
}
```

Output –

```
Enter the no. of nodes: 5  
0 0 1 0 0  
1 0 0 1 0  
0 0 0 0 1  
0 0 1 0 1  
0 0 0 0 0  
Solution:1 3 0 2 4  
Process returned 0 (0x0)   execution time : 18.138 s  
Press any key to continue.  
|
```

2. Sort a given set of N integer elements using Merge Sort technique and compute its time taken.

Code-

```
#include <stdio.h>

#include<time.h>

int a[20],n;

void simple_sort(int [],int,int,int);

void merge_sort(int[],int,int);

int main() {

int i;

clock_t start, end;

double time_taken;

printf("Enter the no. of elements:");

scanf("%d", &n);

printf("Enter the array elements:");

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

scanf("%d", &a[i]);

}

start = clock();

merge_sort(a, 0, n - 1);

end = clock();

time_taken = (double)(end - start) / CLOCKS_PER_SEC;

printf("Sorted array:");

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

printf("%d ", a[i]);

}

}
```

```
printf("\n");  
printf("Time taken to sort: %f seconds\n", time_taken);  
return 0;  
}  
  
void merge_sort(int a[],int low, int high){  
    if(low<high){  
        int mid=(low+high)/2;  
        merge_sort(a,low,mid);  
        merge_sort(a,mid+1,high);  
        simple_sort(a,low,mid,high);  
    }  
}  
  
void simple_sort(int a[],int low, int mid, int high){  
    int i=low,j=mid+1,k=low;  
    int c[n];  
    while(i<=mid && j<=high){  
        if(a[i]<a[j]){  
            c[k++]=a[i];  
            i++;  
        }else{  
            c[k++]=a[j];  
            j++;  
        }  
    }  
    while(i<=mid){  
        c[k++]=a[i];
```



```
i++;  
}  
while(j<=high){  
c[k++]=a[j];  
j++;  
}  
for(i=low;j<=high;i++){  
a[i]=c[i];  
}  
}
```

Code-

```
Enter the no. of elements:8  
Enter the array elements:1 4 2 3 8 7 6 5  
Sorted array:1 2 3 4 5 6 7 8  
Time taken to sort: 0.000000 seconds  
  
Process returned 0 (0x0)   execution time : 35.522 s  
Press any key to continue.  
|
```

3. Sort a given set of N integer elements using Quick Sort technique and compute its time taken.

Code-

```
#include <stdio.h>

#include<time.h>

    int a[20],n;

int partition(int [],int, int);

void quick_sort(int [],int,int);

void swap(int*,int*);

int main() {

    int i;

    clock_t start, end;

    double time_taken;

    printf("Enter the no. of elements:");

    scanf("%d", &n);

    printf("Enter the array elements:");

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

        scanf("%d", &a[i]);

    }

    start = clock();

    quick_sort(a, 0, n - 1);

    end = clock();

    time_taken = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Sorted array:");

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

        printf("%d ", a[i]);

    }
```

```
printf("\n");  
printf("Time taken to sort: %f seconds\n", time_taken);  
return 0;  
}  
  
void swap(int *a,int *b){  
    int temp=*a;  
    *a=*b;  
    *b=temp;  
}  
  
void quick_sort(int a[],int low,int high){  
    if(low<high){  
        int mid=partition(a,low,high);  
        quick_sort(a,low,mid-1);  
        quick_sort(a,mid+1,high);  
    }  
}  
  
int partition(int a[],int low,int high){  
    int pivot=a[low];  
    int i=low;  
    int j=high+1;  
    while(i<=j){  
        do{  
            i=i+1;  
        }while(a[i]<pivot && i<=high);  
        do{  
            j=j-1;  
        }while(a[j]>pivot && j>low);  
        if(i<j){  
            swap(&a[i],&a[j]);  
        }  
    }  
    swap(&a[low],&a[i]);  
    return i;  
}
```

```
    } while(a[j]>pivot && j>=low);  
    if(i<j){  
        swap(&a[i],&a[j]);  
    }  
}  
swap(&a[j], &a[low]);  
return j;  
}
```

Output -

```
Enter the no. of elements:5  
Enter the array elements:3 2 1 5 4  
Sorted array:1 2 3 4 5  
Time taken to sort: 0.000000 seconds  
  
Process returned 0 (0x0)   execution time : 12.259 s  
Press any key to continue.
```

4 a. Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.

Code -

```
#include <stdio.h>

int cost[10][10], n, t[10][2], sum;

void prims(int cost[10][10], int n);

int main() {
    int i, j;

    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    printf("Enter the cost adjacency matrix:\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &cost[i][j]);
        }
    }

    prims(cost, n);

    printf("Edges of the minimal spanning tree:\n");
    for (i = 0; i < n - 1; i++) {
        printf("(%d, %d) ", t[i][0], t[i][1]);
    }

    printf("\nSum of minimal spanning tree: %d\n", sum);

    return 0;
}
```

```

}

void prims(int cost[10][10], int n) {
    int i, j, u, v;
    int min, source;
    int p[10], d[10], s[10];
    min = 999;
    source = 0;
    // Initialize arrays
    for (i = 0; i < n; i++) {
        d[i] = cost[source][i];
        s[i] = 0;
        p[i] = source;
    }

    s[source] = 1;
    sum = 0;
    int k = 0;
    // Find MST
    for (i = 0; i < n - 1; i++) {
        min = 999;
        u = -1;
        // Find the vertex with minimum distance to the MST
        for (j = 0; j < n; j++) {
            if (s[j] == 0 && d[j] < min) {
                min = d[j];
                u = j;
            }
        }
    }
}

```

```

    }
}
if (u != -1) {
    // Add edge to MST
    t[k][0] = u;
    t[k][1] = p[u];
    k++;
    sum += cost[u][p[u]];
    s[u] = 1;
    // Update distances
    for (v = 0; v < n; v++) {
        if (s[v] == 0 && cost[u][v] < d[v]) {
            d[v] = cost[u][v];
            p[v] = u;
        }
    }
}
}
}
}

```

Output-

```
Enter the number of vertices: 4
Enter the cost adjacency matrix:
0 1 5 2
1 0 99 99
5 99 0 3
2 99 3 0
Edges of the minimal spanning tree:
(1, 0) (3, 0) (2, 3)
Sum of minimal spanning tree: 6

Process returned 0 (0x0)   execution time : 38.867 s
Press any key to continue.
```


4 b. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.

Code-

```
#include <stdio.h>

int cost[10][10], n, t[10][2], sum;

void kruskal(int cost[10][10], int n);

int find(int parent[10], int i);

int main() {
    int i, j;

    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    printf("Enter the cost adjacency matrix:\n");

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &cost[i][j]);
        }
    }

    kruskal(cost, n);

    printf("Edges of the minimal spanning tree:\n");

    for (i = 0; i < n - 1; i++) {
        printf("(%d, %d) ", t[i][0], t[i][1]);
    }

    printf("\nSum of minimal spanning tree: %d\n", sum);

    return 0;
}
```

```

void kruskal(int cost[10][10], int n) {
    int min, u, v, count, k;
    int parent[10];
    k = 0;
    sum = 0;
    // Initialize parent array for Union-Find
    for (int i = 0; i < n; i++) {
        parent[i] = i;
    }
    count = 0;
    while (count < n - 1) {
        min = 999;
        u = -1;
        v = -1;
        // Find the minimum edge
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (find(parent, i) != find(parent, j) && cost[i][j] < min) {
                    min = cost[i][j];
                    u = i;
                    v = j;
                }
            }
        }
        // Perform Union operation
        int root_u = find(parent, u);

```

```
int root_v = find(parent, v);  
if (root_u != root_v) {  
    parent[root_u] = root_v;  
    t[k][0] = u;  
    t[k][1] = v;  
    sum += min;  
    k++;  
    count++;  
}  
}  
}  
  
int find(int parent[10], int i) {  
    while (parent[i] != i) {  
        i = parent[i];  
    }  
    return i;  
}
```

Output-

```
Enter the number of vertices: 4
Enter the cost adjacency matrix:
0 1 5 2
1 0 99 99
5 99 0 3
2 99 3 0
Edges of the minimal spanning tree:
(0, 1) (0, 3) (2, 3)
Sum of minimal spanning tree: 6

Process returned 0 (0x0)   execution time : 28.836 s
Press any key to continue.
```

5.From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

// C program to implement Dijkstra's algorithm

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

```
int cost[10][10], n, result[10][2], weight[10];
```

```
void dijkstras(int[][10], int );
```

```
void main() {
```

```
    int i, j, s;
```

```
    printf("Enter the number of vertices: ");
```

```
    scanf("%d", &n);
```

```
    printf("Enter the cost adjacency matrix:\n");
```

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

```
        for (j = 0; j < n; j++) {
```

```
            scanf("%d", &cost[i][j]);
```

```
        }
```

```
    }
```

```
    printf("Enter the source vertex: ");
```

```
    scanf("%d", &s);
```

```
    dijkstras(cost, s);
```

```
    printf("Path:\n");
```

```
    for (i = 1; i < n; i++) {
```

```
        printf("(%d, %d) with weight %d ", result[i][0], result[i][1], weight[result[i][1]]);
```

```
    }
```

```
}
```

```

void dijkstras(int cost[][10], int s){
    int d[10], p[10], visited[10];

    int i, j, min, u, v, k;

    for(i = 0; i < 10; i++){
        d[i] = 999;
        visited[i] = 0;
        p[i] = s;
    }
    d[s] = 0;
    visited[s] = 1;
    for(i = 0; i < n; i++){
        min = 999;
        u = 0;
        for(j = 0; j < n; j++){
            if(visited[j] == 0){
                if(d[j] < min){
                    min = d[j];
                    u = j;
                }
            }
        }
        visited[u] = 1;
        for(v = 0; v < n; v++){
            if(visited[v] == 0 && (d[u] + cost[u][v] < d[v])){
                d[v] = d[u] + cost[u][v];
                p[v] = u;
            }
        }
    }
}

```

```

        }
    }
}

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

    result[i][0] = p[i];

    result[i][1] = i;

    weight[i] = d[i];

}

}

```

Output –

```

Enter the number of vertices: 4
Enter the cost adjacency matrix:
0 1 7 8
99 0 2 5
99 99 0 4
99 99 99 0
Enter the source vertex: 0
Path:
(0, 1) with weight 1 (1, 2) with weight 3 (1, 3) with weight 6
Process returned 4 (0x4)   execution time : 85.301 s
Press any key to continue.

```

6. Implement Johnson Trotter algorithm to generate permutations.

```
#include <stdio.h>

#include <stdlib.h>

#define LEFT -1
#define RIGHT 1

int getLargestMobile(int* arr, int* dir, int n) {
    int mobile = 0;

    for (int i = 0; i < n; i++) {
        int neighbor = i + dir[i];

        if (neighbor >= 0 && neighbor < n && arr[i] > arr[neighbor]) {
            if (arr[i] > mobile) {
                mobile = arr[i];
            }
        }
    }

    return mobile;
}

int findIndex(int* arr, int n, int val) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == val)
            return i;
    }

    return -1;
}
```



```

void printPermutation(int* arr, int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int factorial(int n) {
    int f = 1;
    for (int i = 2; i <= n; i++)
        f *= i;
    return f;
}

void johnsonTrotter(int* arr, int n) {
    int dir[n];
    for (int i = 0; i < n; i++)
        dir[i] = LEFT;
    printPermutation(arr, n); // print first permutation
    int total = factorial(n);
    for (int step = 1; step < total; step++) {
        int mobile = getLargestMobile(arr, dir, n);
        if (mobile == 0) break; // no more mobile integers
        int pos = findIndex(arr, n, mobile);
        int moveTo = pos + dir[pos];
        if (moveTo < 0 || moveTo >= n) {
            break;
        }
        int temp = arr[pos];

```

```

    arr[pos] = arr[moveTo];
    arr[moveTo] = temp;
    int dtemp = dir[pos];
    dir[pos] = dir[moveTo];
    dir[moveTo] = dtemp;
    pos = moveTo;
    for (int i = 0; i < n; i++) {
        if (arr[i] > mobile) {
            dir[i] = -dir[i];
        }
    }
    printPermutation(arr, n);
}

int main() {
    int n;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("Invalid input.\n");
        return 1;
    }

    int arr[n];

    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

```

```
printf("Permutations using Johnson-Trotter algorithm:\n");  
johnsonTrotter(arr, n);  
}
```

Output-

```
Enter the number of elements: 3  
Enter 3 elements: 1 3 5  
Permutations using Johnson-Trotter algorithm:  
1 3 5  
1 5 3  
5 1 3  
5 3 1  
3 5 1  
3 1 5  
  
Process returned 0 (0x0)   execution time : 7.134 s  
Press any key to continue.
```

9. Implement fractional Knapsack problem using Greedy technique.

Code-

```
#include <stdio.h>

void knapsack(int n, int p[], int w[], int W) {
    int used[n];

    for (int i = 0; i < n; ++i)
        used[i] = 0;

    int cur_w = W;
    float tot_v = 0.0;
    int i, maxi;

    while (cur_w > 0) {
        maxi = -1;

        for (i = 0; i < n; ++i)
            if ((used[i] == 0) && ((maxi == -1) || ((float)w[i]/p[i] > (float)w[maxi]/p[maxi])))
                maxi = i;

        used[maxi] = 1;

        if (w[maxi] <= cur_w) {
            cur_w -= w[maxi];
            tot_v += p[maxi];

            printf("Added object %d (%d, %d) completely in the bag. Space left: %d.\n", maxi + 1,
                w[maxi], p[maxi], cur_w);
        } else {
            int taken = cur_w;
            cur_w = 0;
```

```

        tot_v += (float)taken/p[maxi] * p[maxi];

        printf("Added %d%% (%d, %d) of object %d in the bag.\n", (int)((float)taken/w[maxi] *
100), w[maxi], p[maxi], maxi + 1);

    }

}

printf("Filled the bag with objects worth %.2f.\n", tot_v);
}

int main(){

    int n, W;

    printf("Enter the number of objects: ");

    scanf("%d", &n);

    int p[n], w[n];

    printf("Enter the profits of the objects: ");

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

        scanf("%d", &p[i]);

    }

    printf("Enter the weights of the objects: ");

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

        scanf("%d", &w[i]);

    }

    printf("Enter the maximum weight of the bag: ");

    scanf("%d", &W);

    knapsack(n, p, w, W);

    return 0;

}

```

Output –

```
Enter the number of objects: 7
Enter the profits of the objects: 5 10 15 7 8 9 4
Enter the weights of the objects: 1 3 5 4 1 3 2
Enter the maximum weight of the bag: 15
Added object 4 (4, 7) completely in the bag. Space left: 11.
Added object 7 (2, 4) completely in the bag. Space left: 9.
Added object 3 (5, 15) completely in the bag. Space left: 4.
Added object 6 (3, 9) completely in the bag. Space left: 1.
Added 33% (3, 10) of object 2 in the bag.
Filled the bag with objects worth 36.00.

Process returned 0 (0x0)   execution time : 95.273 s
Press any key to continue.
|
```

8. Implement 0/1 Knapsack problem using dynamic programming.

Code-

```
#include <stdio.h>

int n,m,w[10],p[10],v[10][10];

void knapsack(int,int,int[],int[]);

int max(int,int);

int main()

{
    int i,j;

    printf("Enter the no. of items:");

    scanf("%d",&n);

    printf("Enter the capacity of knapsack:");

    scanf("%d",&m);

    printf("Enter weights:");

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

        scanf("%d",&w[i]);

    }

    printf("Enter profits:");

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

        scanf("%d",&p[i]);

    }

    knapsack(n,m,w,p);

    printf("Optimal Solution:\n");
```

```

for(i=0;i<n;i++){
for(j=0;j<n;j++){
printf("%d ",v[i][j]);
}
printf("\n");
}
return 0;
}

void knapsack(int n, int m, int w[],int p[]){
int i,j;
for(i=0;i<n;i++){
for(j=0;j<m;j++){
if(i==0 || j==0){
v[i][j]=0;
}else if(w[i]>j){
v[i][j]=v[i-1][j];
}else{
v[i][j]=max(v[i-1][j],((v[i-1][j-w[i]])+p[i]));
}
}
}
}

int max(int a,int b){
if(a>b){
return a;
}else{

```



```
return b;
```

```
}
```

```
}
```

Code-

```
Enter the no. of items:5
Enter the capacity of knapsack:5
Enter weights:1 2 3 4 5
Enter profits:20 30 40 25 45
Optimal Solution:
0 0 0 0 0
0 0 30 30 30
0 0 30 40 40
0 0 30 40 40
0 0 30 40 40

Process returned 0 (0x0)   execution time : 67.405 s
Press any key to continue.
|
```

9. Implement All Pair Shortest paths problem using Floyd's algorithm.

Code-

```
#include <stdio.h>

int a[10][10],D[10][10],n;

void floyd(int[][10],int);

int min(int,int);

int main(){

printf("Enter the no. of vertices:");

scanf("%d",&n);

printf("Enter the cost adjacency matrix:\n");

int i,j;

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

for(j=0;j<n;j++){

scanf("%d",&a[i][j]);

}

}

floyd(a,n);

printf("Distance Matrix:\n");

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

for(j=0;j<n;j++){

printf("%d ",D[i][j]);

}

printf("\n");
```

```

}
return 0;
}
void floyd(int a[][10],int n){
    int i,j,k;
    for(i=0;i<n;i++){
        for(j=0;j<n;j++){
            D[i][j]=a[i][j];
        }
    }
    for(k=0;k<n;k++){
        for(i=0;i<n;i++){
            for(j=0;j<n;j++){
                D[i][j]=min(D[i][j],(D[i][k]+D[k][j]));
            }
        }
    }
}
int min(int a,int b){
    if(a<b){
        return a;
    }else{
        return b;
    }
}

```

Output -

```
"C:\Users\AYUSH ADITYA\Doc x + v
Enter the no. of vertices:4
Enter the cost adjacency matrix:
0 5 3 4
5 0 999 7
3 999 0 9
4 7 9 0
Distance Matrix:
0 5 3 4
5 0 8 7
3 8 0 7
4 7 7 0
Process returned 0 (0x0) execution time : 63.214 s
Press any key to continue.
```

10. Sort a given set of N integer elements using Heap Sort technique and compute its time taken.

Code-

```
#include <stdio.h>

#include<time.h>

#define MAX 100

int a[MAX], n;

void heapify(int a[], int n, int i);

void heapSort(int a[], int n);

void swap(int *x, int *y);

int main() {

    printf("Enter the number of array elements: ");

    scanf("%d", &n);

    if (n > MAX) {

        printf("Array size exceeds maximum limit.\n");

        return 1;

    }

    printf("Enter array elements:\n");

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

        scanf("%d", &a[i]);

    }

    clock_t start, end;

    start = clock();

    heapSort(a, n);
```

```

    end = clock();
    float total = 0;
    printf("Sorted array using Heap Sort:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
    total = (float)(end-start)/CLOCKS_PER_SEC;
    return 0;
}

// Function to perform heap sort
void heapSort(int a[], int n) {
    // Build max heap (heapify non-leaf nodes from bottom up)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(a, n, i);
    }

    // One by one extract elements from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root (max) to end
        swap(&a[0], &a[i]);

        // Heapify reduced heap
        heapify(a, i, 0);
    }
}

// To heapify a subtree rooted at index i in array of size n

```

```

void heapify(int a[], int n, int i) {
    int largest = i;    // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child
    // If left child exists and is greater than root
    if (left < n && a[left] > a[largest])
        largest = left;
    // If right child exists and is greater than current largest
    if (right < n && a[right] > a[largest])
        largest = right;
    // If largest is not root, swap and continue heapifying
    if (largest != i) {
        swap(&a[i], &a[largest]);
        heapify(a, n, largest);
    }
}

// Utility function to swap two elements
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

```

Output –

```
"C:\Users\AYUSH ADITYA\Doc" x + v
Enter the number of array elements: 7
Enter array elements:
6 3 2 8 9 1 4
Sorted array using Heap Sort:
1 2 3 4 6 8 9
Process returned 0 (0x0)   execution time : 21.105 s
Press any key to continue.
|
```


11. Implement “N-Queens Problem” using Backtracking.

Code-

```
#include <stdio.h>

#include <stdbool.h>

bool place(int[], int);

void printSolution(int[], int);

void nQueens(int);

void main() {

    int n;

    printf("Enter the number of queens: ");

    scanf("%d",&n);

    nQueens(n);

}

void nQueens(int n){

    int x[10];

    int count=0;

    int k=1;

    while(k!=0){

        x[k]=x[k]+1;

        while(x[k]<=n && !place(x,k)){

            x[k]=x[k]+1;

        }

        if(x[k]<=n){

            if(k==n){
```

```

        printSolution(x, n);

        printf("Solution found\n");

        count++;

    }else{

        k++;

        x[k]=0;

    }

}

}

printf("Total solutions: %d\n", count);

}

bool place(int x[10], int k){

    int i;

    for(i=1;i<k;i++){

        if((x[i]==x[k]) || (i-x[i]==k-x[k]) || (i+x[i]==k+x[k])){

            return false;

        }

    }

    return true;

}

void printSolution(int x[10], int n){

    int i;

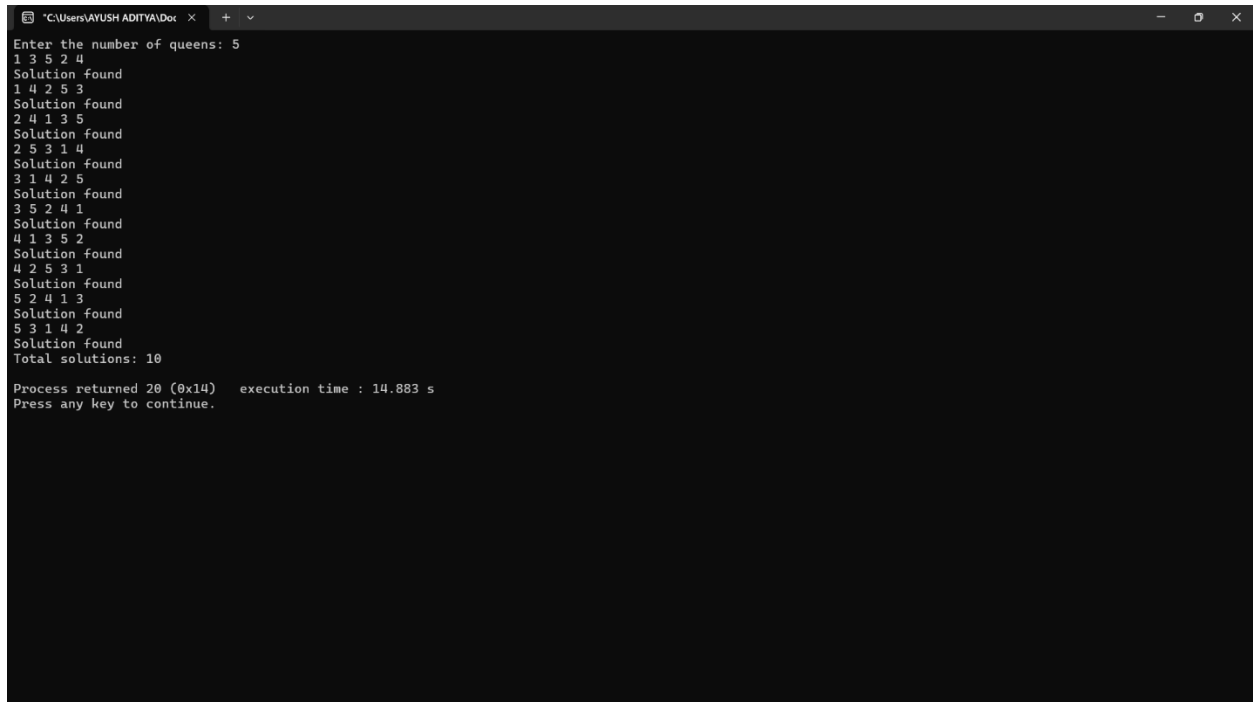
    for(i=1;i<=n;i++){

        printf("%d ", x[i]);

```

```
}  
  
printf("\n");  
  
}
```

Output-



```
*C:\Users\AYUSH ADITYA\Doc  x  +  v  
Enter the number of queens: 5  
1 3 5 2 4  
Solution found  
1 4 2 5 3  
Solution found  
2 4 1 3 5  
Solution found  
2 5 3 1 4  
Solution found  
3 1 4 2 5  
Solution found  
3 5 2 4 1  
Solution found  
4 1 3 5 2  
Solution found  
4 2 5 3 1  
Solution found  
5 2 4 1 3  
Solution found  
5 3 1 4 2  
Solution found  
Total solutions: 10  
  
Process returned 20 (0x14)   execution time : 14.883 s  
Press any key to continue.
```

LeetCode Problems

207. Course Schedule

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

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

```
#include <stdbool.h>
```

```
typedef struct Edge {
```

```
    int dest;
```

```
    struct Edge* next;
```

```
} Edge;
```

```
void addEdge(Edge** graph, int src, int dest) {
```

```
    Edge* newEdge = (Edge*)malloc(sizeof(Edge));
```

```
    newEdge->dest = dest;
```

```
    newEdge->next = graph[src];
```

```
    graph[src] = newEdge;
```

```
}
```

```
void createGraph(Edge** graph, int** prerequisites, int prerequisitesSize) {
```

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

```
        int src = prerequisites[i][0];
```

```
        int dest = prerequisites[i][1];
```

```
        addEdge(graph, src, dest);
```

```
    }
```

```
}
```

```

bool hasCycleUtil(Edge** graph, int curr, bool* visited, bool* stack) {
    visited[curr] = true;
    stack[curr] = true;

    Edge* temp = graph[curr];
    while (temp != NULL) {
        int neighbor = temp->dest;
        if (stack[neighbor]) {
            return true;
        } else if (!visited[neighbor]) {
            if (hasCycleUtil(graph, neighbor, visited, stack)) {
                return true;
            }
        }
        temp = temp->next;
    }

    stack[curr] = false;
    return false;
}

```

```

bool hasCycle(Edge** graph, int numCourses) {
    bool* visited = (bool*)calloc(numCourses, sizeof(bool));
    bool* stack = (bool*)calloc(numCourses, sizeof(bool));

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

```

```

        if (!visited[i]) {
            if (hasCycleUtil(graph, i, visited, stack)) {
                free(visited);
                free(stack);
                return true;
            }
        }
    }

    free(visited);
    free(stack);
    return false;
}

bool canFinish(int numCourses, int** prerequisites, int prerequisitesSize) {
    Edge** graph = (Edge**)calloc(numCourses, sizeof(Edge*));
    createGraph(graph, prerequisites, prerequisitesSize);

    bool result = !hasCycle(graph, numCourses);

    // Free graph memory
    for (int i = 0; i < numCourses; i++) {
        Edge* curr = graph[i];
        while (curr) {
            Edge* temp = curr;
            curr = curr->next;
        }
    }
}

```

```
        free(temp);  
    }  
}  
free(graph);  
  
return result;  
}
```


15. 3Sum

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

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

```
#include <stdbool.h>
```

```
// Comparator for qsort
```

```
int cmp(const void *a, const void *b) {
```

```
    return (*(int*)a - *(int*)b);
```

```
}
```

```
// Structure to store result triplets
```

```
typedef struct {
```

```
    int** triplets;
```

```
    int returnSize;
```

```
    int* returnColumnSizes;
```

```
    int capacity;
```

```
} Result;
```

```
// Helper to add a triplet to the result
```

```
void addTriplet(Result* res, int a, int b, int c) {
```

```
    if (res->returnSize == res->capacity) {
```

```
        res->capacity *= 2;
```

```
        res->triplets = realloc(res->triplets, res->capacity * sizeof(int*));
```

```
        res->returnColumnSizes = realloc(res->returnColumnSizes, res->capacity * sizeof(int));
```

```
}
```

```

    res->triplets[res->returnSize] = malloc(3 * sizeof(int));
    res->triplets[res->returnSize][0] = a;
    res->triplets[res->returnSize][1] = b;
    res->triplets[res->returnSize][2] = c;
    res->returnColumnSizes[res->returnSize] = 3;
    res->returnSize++;
}

```

// Main function

```

int** threeSum(int* nums, int numsSize, int* returnSize, int** returnColumnSizes) {
    Result res;

    res.triplets = malloc(1000 * sizeof(int*)); // Initial allocation
    res.returnColumnSizes = malloc(1000 * sizeof(int));
    res.returnSize = 0;
    res.capacity = 1000;

    qsort(nums, numsSize, sizeof(int), cmp);

    for (int i = 0; i < numsSize - 2; i++) {
        if (i > 0 && nums[i] == nums[i - 1]) continue; // Skip duplicate i

        int left = i + 1;
        int right = numsSize - 1;

        while (left < right) {
            int sum = nums[i] + nums[left] + nums[right];

```

```
    if (sum < 0) {  
        left++;  
    } else if (sum > 0) {  
        right--;  
    } else {  
        addTriplet(&res, nums[i], nums[left], nums[right]);  
        left++;  
        right--;  
  
        // Skip duplicates for left and right  
        while (left < right && nums[left] == nums[left - 1]) left++;  
        while (left < right && nums[right] == nums[right + 1]) right--;  
    }  
}  
}  
}  
  
*returnSize = res.returnSize;  
*returnColumnSizes = res.returnColumnSizes;  
return res.triplets;  
}
```

912. Sort an Array

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

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

```
// Merge two sorted subarrays: nums[left..mid] and nums[mid+1..right]
```

```
void merge(int* nums, int left, int mid, int right) {
```

```
    int size = right - left + 1;
```

```
    int* temp = (int*)malloc(size * sizeof(int));
```

```
    int i = left, j = mid + 1, k = 0;
```

```
    // Merge both halves into temp[]
```

```
    while (i <= mid && j <= right) {
```

```
        if (nums[i] < nums[j]) {
```

```
            temp[k++] = nums[i++];
```

```
        } else {
```

```
            temp[k++] = nums[j++];
```

```
        }
```

```
    }
```

```
    // Copy remaining elements from left half
```

```
    while (i <= mid) {
```

```
        temp[k++] = nums[i++];
```

```
    }
```

```
    // Copy remaining elements from right half
```

```

while (j <= right) {
    temp[k++] = nums[j++];
}

// Copy sorted data back to original array
for (int l = 0; l < size; l++) {
    nums[left + l] = temp[l];
}

free(temp);
}

// Recursive merge sort function
void mergeSort(int* nums, int left, int right) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    mergeSort(nums, left, mid);
    mergeSort(nums, mid + 1, right);
    merge(nums, left, mid, right);
}

// Entry function for sorting
int* sortArray(int* nums, int numsSize, int* returnSize) {
    mergeSort(nums, 0, numsSize - 1);
    *returnSize = numsSize;
}

```

```
    return nums;
}

int main() {
    int nums[] = {5, 2, 3, 1};
    int size = sizeof(nums) / sizeof(nums[0]);
    int returnSize;

    int* sorted = sortArray(nums, size, &returnSize);

    for (int i = 0; i < returnSize; i++) {
        printf("%d ", sorted[i]);
    }

    return 0;
}
```

1903. Largest Odd Number in String

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

```
#include <string.h>
```

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

```
#include <stdbool.h>
```

```
char* largestOddNumber(char* num) {
```

```
    int n = strlen(num);
```

```
    int idx = -1;
```

```
    // Traverse from right to left to find the last odd digit
```

```
    for (int i = n - 1; i >= 0; i--) {
```

```
        if ((num[i] - '0') % 2 != 0) {
```

```
            idx = i;
```

```
            break;
```

```
        }
```

```
    }
```

```
    if (idx == -1) {
```

```
        // No odd digit found
```

```
        char* empty = (char*)malloc(1);
```

```
        empty[0] = '\0';
```

```
        return empty;
```

```
    }
```

```
// Allocate memory for the result substring
char* result = (char*)malloc((idx + 2) * sizeof(char)); // +1 for null-terminator
strncpy(result, num, idx + 1);
result[idx + 1] = '\0';
return result;
}

int main() {
    char num[] = "35420";
    char* result = largestOddNumber(num);
    printf("Largest odd number: %s\n", result);
    free(result); // Always free dynamically allocated memory
    return 0;
}
```


509. Fibonacci Number

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

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

```
// Function to compute Fibonacci number using DP
```

```
int fib(int n) {
```

```
    if (n <= 1) return n;
```

```
    int* dp = (int*)malloc((n + 1) * sizeof(int));
```

```
    dp[0] = 0;
```

```
    dp[1] = 1;
```

```
    for (int i = 2; i <= n; i++) {
```

```
        dp[i] = dp[i - 1] + dp[i - 2];
```

```
    }
```

```
    int result = dp[n];
```

```
    free(dp); // Free allocated memory
```

```
    return result;
```

```
}
```

```
int main() {
```

```
    int n;
```

```
    printf("Enter a number n to compute the nth Fibonacci number: ");
```

```
    scanf("%d", &n);
```

```
int result = fib(n);  
printf("Fibonacci number at position %d is %d\n", n, result);  
  
return 0;  
}
```

821. Shortest Distance to a Character

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

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

```
#include <string.h>
```

```
int* shortestToChar(char* s, char c, int* returnSize) {
```

```
    int n = strlen(s);
```

```
    int* answer = (int*)malloc(n * sizeof(int));
```

```
    *returnSize = n;
```

```
    int pos = -n; // Initialize to a large negative to simulate "not seen yet"
```

```
    // First pass: left to right
```

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

```
        if (s[i] == c) {
```

```
            pos = i;
```

```
        }
```

```
        answer[i] = i - pos;
```

```
    }
```

```
    // Second pass: right to left
```

```
    pos = 2 * n; // Initialize to a large positive
```

```
    for (int i = n - 1; i >= 0; i--) {
```

```
        if (s[i] == c) {
```

```
            pos = i;
```

```
    }  
    if (pos - i < answer[i]) {  
        answer[i] = pos - i;  
    }  
}  
  
return answer;  
}  
  
int main() {  
    char s[] = "loveleetcode";  
    char c = 'e';  
    int returnSize;  
  
    int* result = shortestToChar(s, c, &returnSize);  
  
    printf("Shortest distances to character '%c':\n", c);  
    for (int i = 0; i < returnSize; i++) {  
        printf("%d ", result[i]);  
    }  
    printf("\n");  
  
    free(result);  
    return 0;  
}
```