1.POST ORDER WITHOUT RECURSION

```c
#include <stdio.h>

#include <stdlib.h>


// Definition of a binary tree node
struct Node {

    int data;

    struct Node* left;

    struct Node* right;

};


// Function to create a new node
struct Node* createNode(int data) {

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}


// Function for iterative postorder traversal
void iterativePostorder(struct Node* root) {

    if (root == NULL)

        return;


    struct Node* stack1[100];

    struct Node* stack2[100];

    int top1 = -1;

    int top2 = -1;
```

```c
    stack1[++top1] = root;

    while (top1 >= 0) {
        struct Node* curr = stack1[top1--];
        stack2[++top2] = curr;

        if (curr->left != NULL)
            stack1[++top1] = curr->left;
        if (curr->right != NULL)
            stack1[++top1] = curr->right;
    }

    while (top2 >= 0) {
        struct Node* curr = stack2[top2--];
        printf("%d ", curr->data);
    }
}

int main() {
    // Create the binary tree
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Postorder traversal: ");
    iterativePostorder(root);
```

```c
    return 0;
}
```

2. PREORDER WITHOUT RECURSION

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

// Definition of a binary tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function for iterative preorder traversal
void iterativePreorder(struct Node* root) {
    if (root == NULL)
        return;

    struct Node* stack[100];
```

```c
    int top = -1;

    stack[++top] = root;


    while (top >= 0) {

        struct Node* curr = stack[top--];

        printf("%d ", curr->data);


        if (curr->right != NULL)

            stack[++top] = curr->right;

        if (curr->left != NULL)

            stack[++top] = curr->left;

    }

}


int main() {

    // Create the binary tree

    struct Node* root = createNode(1);

    root->left = createNode(2);

    root->right = createNode(3);

    root->left->left = createNode(4);

    root->left->right = createNode(5);


    printf("Preorder traversal: ");

    iterativePreorder(root);


    return 0;

}
```

3. COINS PROBLEM USING GREEDY METHOD

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

```c
void countNotes(int amount) {

    int denominations[] = {2000, 500, 200, 100, 50, 20, 10, 5, 2, 1};

    int count[10] = {0};

    int i;


    for (i = 0; i < 10; i++) {

        if (amount >= denominations[i]) {

            count[i] = amount / denominations[i];

            amount = amount - (count[i] * denominations[i]);

        }

    }


    printf("Denomination\tNumber of Notes\n");

    for (i = 0; i < 10; i++) {

        if (count[i] != 0) {

            printf("%d\t\t%d\n", denominations[i], count[i]);

        }

    }

}


int main() {

    int amount;


    printf("Enter the amount to withdraw: ");

    scanf("%d", &amount);


    printf("Minimum number of coins/notes:\n");

    countNotes(amount);
```

```c
    return 0;

}

4. RANDOMIZED QUICK SORT

#include <stdio.h>

#include<stdlib.h>

#include<math.h>

#include<time.h>

void quicksort(int x[20],int p,int q)

{

    int temp,i,j,pivot;

    if(p<q)

    {

        i=p;

        j=q;

        pivot=p+rand()%(q-p);

        temp=x[pivot];

        x[pivot]=x[p];

        x[p]=temp;

        while(i<j)

        {

            while(x[i]<=x[pivot] && i<q)

            {

                i++;

            }

            while(x[j]>x[pivot])

            {

                j--;

            }
```

```c
        if(i<j)

        {

            temp=x[i];

            x[i]=x[j];

            x[j]=temp;

        }}

    temp=x[pivot];

    x[pivot]=x[j];

    x[j]=temp;

    quicksort(x,p,j-1);

    quicksort(x,j+1,q);

    }

}


int main()

{

    int n;

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

    scanf("%d",&n);

    int a[n];

    int i;

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

    {

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

    }

    quicksort(a,0,n-1);

    printf("Sorted:");

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

    {
```

```c
        printf("%d\t",a[i]);
    }


    return 0;
}
```

5. DOUBLE HASHING

```c
#include <stdio.h>
#include<stdlib.h>
#define TABLE_SIZE 10


int h[TABLE_SIZE]={NULL};


void insert()
{


 int key,index,i,flag=0,hkey,hash2;
 printf("\nenter a value to insert into hash table\n");
 scanf("%d",&key);
 hkey=key%TABLE_SIZE;
 hash2 = 7-(key %7);
 for(i=0;i<TABLE_SIZE;i++)
 {
   index=(hkey+i*hash2)%TABLE_SIZE;
   if(h[index] == NULL)
   {
      h[index]=key;
      break;
   }
 }
```

```c
  if(i == TABLE_SIZE)
     printf("\nelement cannot be inserted\n");
}
void search()
{

 int key,index,i,flag=0,hash2,hkey;
 printf("\nenter search element\n");
 scanf("%d",&key);
  hkey=key%TABLE_SIZE;
 hash2 = 7-(key %7);
 for(i=0;i<TABLE_SIZE; i++)
 {
   index=(hkey+i*hash2)%TABLE_SIZE;
   if(h[index]==key)
   {
     printf("value is found at index %d",index);
     break;
   }
 }
 if(i == TABLE_SIZE)
   printf("\n value is not found\n");
}
void display()
{

 int i;
 printf("\nelements in the hash table are \n");
 for(i=0;i< TABLE_SIZE; i++)
```

```c
        printf("\nat index %d \t value =  %d",i,h[i]);


}
main()
{
    int opt,i;
    while(1)
    {
        printf("\nPress 1. Insert\t 2. Display \t3. Search \t4.Exit \n");
        scanf("%d",&opt);
        switch(opt)
        {
            case 1:
                insert();
                break;
            case 2:
                display();
                break;
            case 3:
                search();
                break;
            case 4:exit(0);
        }
    }
}
```

6. BINARY SEARCH USING DIVIDE AND CONQUER

```c
#include<stdio.h>


int binary_search(int A[], int key, int len) {
```

```c
    int low = 0;
    int high = len -1;

    while (low <= high) {
     int mid = low + ((high - low) / 2);

      if (A[mid] == key) {
        return mid;
      }

      if (key < A[mid]) {
        high = mid - 1;
      }
      else {
        low = mid + 1;
      }
     }
    return -1;
   }

int main() {
  int a[10]={1,3,5,7,9,11,13,15,17,21};
  int key = 3;

  int position = binary_search(a, key, 10);
  if (position == -1){
    printf("Not found");
    return 0;
```

```c
  }

  printf("Found it at %d", position);

  return 0;

}
```

7. INORDER TRAVERSAL WITHOUT RECURSION

```c
#include<stdio.h>

#include<stdlib.h>

struct node // node defining for tree

{

        struct node* left;

        struct node* right;

        int data;

};

struct stack // node defining for stack

{

        struct node* data;

        struct stack* next;

};


void push(struct stack** top,struct node* n); //function declation

struct node* pop(struct stack** top);

int isEmpty(struct stack* top);


int Inorder(struct node* root) //Inorder Traversing function

{

        struct node* temp = root;

        struct stack* s_temp = NULL;

        int flag = 1;

        while(flag) //Loop run untill temp is null and stack is empty
```

```c
        {
                if(temp){

                        push(&s_temp,temp);

                        temp = temp->left;

                }

                else{

                        if(!isEmpty(s_temp)){

                                temp = pop(&s_temp);

                                printf("%d ",temp->data);

                                temp = temp->right;

                        }

                        else

                                flag = 0;

                }

        }

}

void push(struct stack** top,struct node* n) //push node in stack

{

        struct stack* new_n = (struct stack*)malloc(sizeof(struct stack));

        new_n->data = n;

        new_n->next = (*top);

        (*top) = new_n;

}

int isEmpty(struct stack* top) // check if stack is empty

{

        if(top==NULL)

                return 1;

        else

                return 0;
```

```c
}
struct node* pop(struct stack** top_n) // pop the node from stack
{
        struct node* item;

        struct stack* top;

        top = *top_n;

        item = top->data;

        *top_n = top->next;

        free(top);

        return item;
}
struct node* create_node(int data) // create a node for tree
{
        struct node* new_n = (struct node*)malloc(sizeof(struct node));

        new_n->data = data;

        new_n->left = NULL;

        new_n->right = NULL;

        return (new_n);
}

int main()
{
        struct node* root;

        root = create_node(8);

        root->left = create_node(5);

        root->right = create_node(4);

        root->left->left = create_node(7);

        root->left->right = create_node(6);

        Inorder(root);
```

```c
        return 0;

}
```

8. JOB SEQUENCING USING GREEDY ALGORITHM

```c
#include <stdio.h>

#define MAX 100

typedef struct Job {

  char id[5];

  int deadline;

  int profit;

} Job;

void jobSequencingWithDeadline(Job jobs[], int n);

int minValue(int x, int y) {

  if(x < y) return x;

  return y;

}

int main(void) {

  //variables

  int i, j;

  //jobs with deadline and profit

  Job jobs[5] = {

    {"j1", 2,  60},

    {"j2", 1, 100},

    {"j3", 3,  20},
```

```c
    {"j4", 2,  40},

    {"j5", 1,  20},

};


//temp
Job temp;


//number of jobs
int n = 5;


//sort the jobs profit wise in descending order
for(i = 1; i < n; i++) {

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

    if(jobs[j+1].profit > jobs[j].profit) {

      temp = jobs[j+1];

      jobs[j+1] = jobs[j];

      jobs[j] = temp;

    }

  }

}


printf("%10s %10s %10s\n", "Job", "Deadline", "Profit");

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

  printf("%10s %10i %10i\n", jobs[i].id, jobs[i].deadline, jobs[i].profit);

}


jobSequencingWithDeadline(jobs, n);


return 0;
```

```c
}

void jobSequencingWithDeadline(Job jobs[], int n) {
  //variables
  int i, j, k, maxprofit;

  //free time slots
  int timeslot[MAX];

  //filled time slots
  int filledTimeSlot = 0;

  //find max deadline value
  int dmax = 0;
  for(i = 0; i < n; i++) {
    if(jobs[i].deadline > dmax) {
      dmax = jobs[i].deadline;
    }
  }

  //free time slots initially set to -1 [-1 denotes EMPTY]
  for(i = 1; i <= dmax; i++) {
    timeslot[i] = -1;
  }

  printf("dmax: %d\n", dmax);

  for(i = 1; i <= n; i++) {
    k = minValue(dmax, jobs[i - 1].deadline);
```

```c
    while(k >= 1) {

      if(timeslot[k] == -1) {

        timeslot[k] = i-1;

        filledTimeSlot++;

        break;

      }

      k--;

    }


    //if all time slots are filled then stop
    if(filledTimeSlot == dmax) {

      break;

    }

  }


  //required jobs
  printf("\nRequired Jobs: ");

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

    printf("%s", jobs[timeslot[i]].id);


    if(i < dmax) {

      printf(" --> ");

    }

  }


  //required profit
  maxprofit = 0;

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

    maxprofit += jobs[timeslot[i]].profit;
```

```c
  }
  printf("\nMax Profit: %d\n", maxprofit);

}
```

9. MAXMIN USING DIVIDE AND CONQUER

```c
#include<stdio.h>

#include<stdio.h>

int max, min;

int a[100];

void maxmin(int i, int j)

{

 int max1, min1, mid;

 if(i==j)

 {

  max = min = a[i];

 }

 else

 {

  if(i == j-1)

  {

   if(a[i] <a[j])

   {

    max = a[j];

    min = a[i];

   }

   else

   {

    max = a[i];

    min = a[j];

   }
```

```c
    }
    else
    {
     mid = (i+j)/2;
     maxmin(i, mid);
     max1 = max; min1 = min;
     maxmin(mid+1, j);
     if(max <max1)
      max = max1;
     if(min > min1)
      min = min1;
    }
   }
  }
int main ()
{
 int i, num;
 printf ("\nEnter the total number of numbers : ");
 scanf ("%d",&num);
 printf ("Enter the numbers : \n");
 for (i=1;i<=num;i++)
  scanf ("%d",&a[i]);

 max = a[0];
 min = a[0];
 maxmin(1, num);
 printf ("Minimum element in an array : %d\n", min);
 printf ("Maximum element in an array : %d\n", max);
 return 0;
```

}

10. OPTIMAL STORAGE ON TAPES

```c
#include <stdio.h>


// Function to calculate the optimal storage on tapes
void optimalTapeStorage(int tapes[], int numTapes, int files[], int numFiles) {
    int tapeIndex = 0;  // Current tape index


    // Loop through each file
    for (int i = 0; i < numFiles; i++) {
        // Check if the file can fit in the current tape
        if (files[i] <= tapes[tapeIndex]) {
            // Store the file in the current tape
            tapes[tapeIndex] -= files[i];
            printf("File %d stored in Tape %d\n", i+1, tapeIndex+1);
        } else {
            // Move to the next tape
            tapeIndex++;
            // Check if all tapes have been used
            if (tapeIndex >= numTapes) {
                printf("Error: Not enough tapes to store all files\n");
                return;
            }
            // Store the file in the next tape
            tapes[tapeIndex] -= files[i];
            printf("File %d stored in Tape %d\n", i+1, tapeIndex+1);
        }
    }
}
```

```c
int main() {

    // Number of tapes and files

    int numTapes = 3;

    int numFiles = 5;


    // Sizes of tapes and files

    int tapes[] = {100, 200, 150};

    int files[] = {80, 50, 30, 70, 120};


    // Calculate the optimal storage on tapes

    optimalTapeStorage(tapes, numTapes, files, numFiles);


    return 0;
}
```

11. LINEAR PROBING HASHING

```c
#include <stdio.h>

#include<stdlib.h>

#define TABLE_SIZE 10


int h[TABLE_SIZE]={NULL};


void insert()

{


 int key,index,i,flag=0,hkey;

 printf("\nenter a value to insert into hash table\n");

 scanf("%d",&key);

 hkey=key%TABLE_SIZE;
```

```c
    for(i=0;i<TABLE_SIZE;i++)

    {

    index=(hkey+i)%TABLE_SIZE;

    if(h[index] == NULL)

    {

        h[index]=key;

        break;

    }

    }

    if(i == TABLE_SIZE)

    printf("\nelement cannot be inserted\n");
}
void search()
{

    int key,index,i,flag=0,hkey;
    printf("\nenter search element\n");
    scanf("%d",&key);
    hkey=key%TABLE_SIZE;
    for(i=0;i<TABLE_SIZE; i++)
    {
        index=(hkey+i)%TABLE_SIZE;
        if(h[index]==key)
        {
```

```c
            printf("value is found at index %d",index);

            break;

        }

    }

    if(i == TABLE_SIZE)

        printf("\n value is not found\n");

}

void display()

{


    int i;


    printf("\nelements in the hash table are \n");


    for(i=0;i< TABLE_SIZE; i++)


    printf("\nat index %d \t value =  %d",i,h[i]);


}

int main()

{

    int opt,i;

    while(1)

    {

        printf("\nPress 1. Insert\t 2. Display \t3. Search \t4.Exit \n");

        scanf("%d",&opt);

        switch(opt)

        {

            case 1:
```

```c
                insert();
                break;
            case 2:
                display();
                break;
            case 3:
                search();
                break;
            case 4:exit(0);
        }
    }
    return 0;
}
```

12. QUICK SORT

```c
#include<stdio.h>
void quicksort(int number[25],int first,int last){
    int i, j, pivot, temp;
    if(first<last){
        pivot=first;
        i=first;
        j=last;
        while(i<j){
            while(number[i]<=number[pivot]&&i<last)
            i++;
            while(number[j]>number[pivot])
            j--;
            if(i<j){
                temp=number[i];
                number[i]=number[j];
```

```c
            number[j]=temp;

        }

    }

    temp=number[pivot];

    number[pivot]=number[j];

    number[j]=temp;

    quicksort(number,first,j-1);

    quicksort(number,j+1,last);

    }

}

int main(){

    int i, count, number[25];

    printf("How many elements are u going to enter?: ");

    scanf("%d",&count);

    printf("Enter %d elements: ", count);

    for(i=0;i<count;i++)

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

    quicksort(number,0,count-1);

    printf("Order of Sorted elements: ");

    for(i=0;i<count;i++)

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

    return 0;

}
```

13. TOPOLOGICAL SORT

```c
#include<stdio.h>

int vis[10]={0};

int a[10][10]={0};

void fun1(int n)

{
```

```c
    int i,j,count;
    for(i=0;i<n;i++)
    {
       count=0;
       if(vis[i]==0)
       {
          for(j=0;j<n;j++)
          {
             if((a[j][i]==1)&&(vis[j]!=1))
             {
                count+=1;
             }
          }
          if(count==0)
          {
             printf("%d",i+1);
             vis[i]=1;
          }
       }
    }
    return;
}


int main()
{
   int n,i,j;
   printf("Enter no.of vertices:");
   scanf("%d",&n);
   printf("Enter the adjacency matrix:");
```

```c
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    fun1(n);
    for(i=0;i<n;i++)
    {
        if(vis[i]==0)
        {
            fun1(n);
        }
    }
    return 0;
}
```

14. FRACTIONAL KNAPSACK

```c
#include <stdio.h>


int n = 5; /* The number of objects */
int c[10] = {12, 1, 2, 1, 4}; /* c[i] is the *COST* of the ith object; i.e. what
        YOU PAY to take the object */
int v[10] = {4, 2, 2, 1, 10}; /* v[i] is the *VALUE* of the ith object; i.e.
        what YOU GET for taking the object */
int W = 15; /* The maximum weight you can take */


void simple_fill() {
    int cur_w;
```

```c
    float tot_v;

    int i, maxi;

    int used[10];


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

        used[i] = 0; /* I have not used the ith object yet */


    cur_w = W;

    while (cur_w > 0) { /* while there's still room*/

        /* Find the best object */

        maxi = -1;

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

            if ((used[i] == 0) &&

                ((maxi == -1) || ((float)v[i]/c[i] > (float)v[maxi]/c[maxi])))

                maxi = i;


        used[maxi] = 1; /* mark the maxi-th object as used */

        cur_w -= c[maxi]; /* with the object in the bag, I can carry less */

        tot_v += v[maxi];

        if (cur_w >= 0)

            printf("Added object %d (%d$, %dKg) completely in the bag. Space left: %d.\n", maxi + 1, v[maxi], c[maxi], cur_w);

        else {

            printf("Added %d%% (%d$, %dKg) of object %d in the bag.\n", (int)((1 + (float)cur_w/c[maxi]) * 100), v[maxi], c[maxi], maxi + 1);

            tot_v -= v[maxi];

            tot_v += (1 + (float)cur_w/c[maxi]) * v[maxi];

        }

    }
```

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

}


int main(int argc, char *argv[]) {

    simple_fill();


    return 0;

}
```

15. KRUSKALS ALGORITHM

```c
#include <stdio.h>

#include <stdlib.h>


#define MAX_EDGES 100

#define MAX_VERTICES 100


typedef struct {

    int u, v, weight;

} Edge;


typedef struct {

    int parent, rank;

} Subset;


int find(Subset subsets[], int i) {

    if (subsets[i].parent != i)

        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;

}
```

```c
void union_set(Subset subsets[], int x, int y) {

    int xroot = find(subsets, x);

    int yroot = find(subsets, y);


    if (subsets[xroot].rank < subsets[yroot].rank)

        subsets[xroot].parent = yroot;

    else if (subsets[xroot].rank > subsets[yroot].rank)

        subsets[yroot].parent = xroot;

    else {

        subsets[yroot].parent = xroot;

        subsets[xroot].rank++;

    }

}


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

    Edge* a1 = (Edge*)a;

    Edge* b1 = (Edge*)b;

    return a1->weight - b1->weight;

}


void kruskal(int n, int m, Edge edges[]) {

    Subset subsets[MAX_VERTICES];

    int i, j, k;

    Edge result[MAX_EDGES];


    qsort(edges, m, sizeof(Edge), cmp);


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

        subsets[i].parent = i;
```

```c
        subsets[i].rank = 0;
    }


    j = 0;
    k = 0;
    while (j < n - 1 && k < m) {
        Edge next_edge = edges[k++];


        int x = find(subsets, next_edge.u);
        int y = find(subsets, next_edge.v);


        if (x != y) {
            result[j++] = next_edge;
            union_set(subsets, x, y);
        }
    }


    printf("Minimum spanning tree:\n");
    for (i = 0; i < j; i++)
        printf("(%d, %d) -> %d\n", result[i].u, result[i].v, result[i].weight);
}


int main() {
    int n, m, i;
    Edge edges[MAX_EDGES];


    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &n, &m);
```

```c
    printf("Enter edges in the format (u, v, weight):\n");

    for (i = 0; i < m; i++)

        scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].weight);


    kruskal(n, m, edges);


    return 0;
}
```

16. PRIMS ALGORITHM

```c
#include <stdio.h>

#include <limits.h>


#define MAX 100


int cost[MAX][MAX], near[MAX], min_cost = 0;


void prim(int n) {
    int i, j, k, u, v, min;


    // initialize near array
    for (i = 2; i <= n; i++) {
        near[i] = 1;
    }


    // iterate n-1 times to construct tree
    for (i = 1; i < n; i++) {
        min = INT_MAX;


        // find edge with minimum cost
```

```c
        for (j = 2; j <= n; j++) {

            if (near[j] != 0 && cost[j][near[j]] < min) {

                min = cost[j][near[j]];

                u = j;

                v = near[j];

            }

        }


        // add edge to tree

        printf("(%d, %d) cost: %d\n", u, v, min);

        min_cost += min;

        near[u] = 0;


        // update near array

        for (k = 2; k <= n; k++) {

            if (near[k] != 0 && cost[k][near[k]] > cost[k][u]) {

                near[k] = u;

            }

        }

    }


    printf("Minimum cost = %d\n", min_cost);

}


int main() {

    int n, i, j;


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

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

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

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

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

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

        }

    }


    printf("Minimum cost spanning tree edges:\n");

    prim(n);


    return 0;

}
```

17. HEAP SORT

```c
#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the

index of root node in array a[], and 'n' is the size of heap. */

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 is larger than root

    if (left < n && a[left] > a[largest])

        largest = left;

    // If right child is larger than root

    if (right < n && a[right] > a[largest])

        largest = right;

    // If root is not largest
```

```c
    if (largest != i) {

        // swap a[i] with a[largest]

        int temp = a[i];

        a[i] = a[largest];

        a[largest] = temp;


        heapify(a, n, largest);

    }

}
/*Function to implement the heap sort*/

void heapSort(int a[], int n)

{

    for (int i = n / 2 - 1; i >= 0; i--)

        heapify(a, n, i);

    // One by one extract an element from heap

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

        /* Move current root element to end*/

        // swap a[0] with a[i]

        int temp = a[0];

        a[0] = a[i];

        a[i] = temp;


        heapify(a, i, 0);

    }

}
/* function to print the array elements */

void printArr(int arr[], int n)

{

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

```c
    {
        printf("%d", arr[i]);

        printf(" ");

    }


}
int main()
{
    int a[] = {48, 10, 23, 43, 28, 26, 1};

    int n = sizeof(a) / sizeof(a[0]);

    printf("Before sorting array elements are - \n");

    printArr(a, n);

    heapSort(a, n);

    printf("\nAfter sorting array elements are - \n");

    printArr(a, n);

    return 0;
}
```

18. MERGE SORT

```c
#include <stdio.h>


#define max 10


int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];


void merging(int low, int mid, int high) {
    int l1, l2, i;


    for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
```

```c
        if(a[l1] <= a[l2])
            b[i] = a[l1++];
        else
            b[i] = a[l2++];
    }

    while(l1 <= mid)
        b[i++] = a[l1++];

    while(l2 <= high)
        b[i++] = a[l2++];

    for(i = low; i <= high; i++)
        a[i] = b[i];
}

void sort(int low, int high) {
    int mid;

    if(low < high) {
        mid = (low + high) / 2;
        sort(low, mid);
        sort(mid+1, high);
        merging(low, mid, high);
    } else {
        return;
    }
}
```

```c
int main() {
    int i;

    printf("List before sorting\n");

    for(i = 0; i <= max; i++)
        printf("%d ", a[i]);

    sort(0, max);

    printf("\nList after sorting\n");

    for(i = 0; i <= max; i++)
        printf("%d ", a[i]);
    return 0;
}
```

19. BFS

```c
#include <stdio.h>
#define MAX_SIZE 100

// Queue implementation
int queue[MAX_SIZE];
int front = -1, rear = -1;

void enqueue(int value) {
    if (rear == MAX_SIZE - 1) {
        printf("Queue is full.\n");
        return;
    }
```

```c
    if (front == -1)

        front = 0;

    rear++;

    queue[rear] = value;

}


int dequeue() {

    if (front == -1 || front > rear) {

        printf("Queue is empty.\n");

        return -1;

    }

    int value = queue[front];

    front++;

    return value;

}


int isQueueEmpty() {

    if (front == -1 || front > rear)

        return 1;

    else

        return 0;

}


// BFS implementation

void BFS(int adjacencyMatrix[][MAX_SIZE], int vertices, int source) {

    int visited[MAX_SIZE] = {0};

    enqueue(source);

    visited[source] = 1;
```

```c
    printf("BFS traversal: ");

    while (!isQueueEmpty()) {

        int currentVertex = dequeue();

        printf("%d ", currentVertex);


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

            if (adjacencyMatrix[currentVertex][i] && !visited[i]) {

                enqueue(i);

                visited[i] = 1;

            }

        }

    }

}


// Test the BFS function

int main() {

    int vertices, source;

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

    scanf("%d", &vertices);


    int adjacencyMatrix[MAX_SIZE][MAX_SIZE];

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

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

        for (int j = 0; j < vertices; j++) {

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

        }

    }


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

```c
    scanf("%d", &source);

    BFS(adjacencyMatrix, vertices, source);

    return 0;
}
```

20. DFS

```c
#include <stdio.h>

#include <stdlib.h>


#define MAX_SIZE 100


// Data structure to represent a graph node
typedef struct Node {

    int data;

    struct Node* next;
} Node;


// Function to create a new node
Node* createNode(int data) {

    Node* newNode = (Node*)malloc(sizeof(Node));

    newNode->data = data;

    newNode->next = NULL;

    return newNode;
}


// Function to add an edge between two nodes in the graph
void addEdge(Node* graph[], int src, int dest) {

    Node* newNode = createNode(dest);
```

```c
    newNode->next = graph[src];

    graph[src] = newNode;


    newNode = createNode(src);

    newNode->next = graph[dest];

    graph[dest] = newNode;

}


// Function to perform Depth-First Search traversal

void DFS(Node* graph[], int vertex, int visited[]) {

    visited[vertex] = 1;

    printf("%d ", vertex);


    Node* adjNode = graph[vertex];

    while (adjNode != NULL) {

        int adjVertex = adjNode->data;

        if (visited[adjVertex] == 0) {

            DFS(graph, adjVertex, visited);

        }

        adjNode = adjNode->next;

    }

}


// Function to initialize visited array with all zeros

void initializeVisited(int visited[], int size) {

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

        visited[i] = 0;

    }

}
```

```c
// Driver code
int main() {

    int numVertices = 6;

    Node* graph[MAX_SIZE];

    int visited[MAX_SIZE];


    // Initialize graph and visited array

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

        graph[i] = NULL;

        visited[i] = 0;

    }


    // Add edges to the graph

    addEdge(graph, 0, 1);

    addEdge(graph, 0, 2);

    addEdge(graph, 1, 3);

    addEdge(graph, 2, 4);

    addEdge(graph, 3, 4);

    addEdge(graph, 3, 5);


    printf("Depth-First Search Traversal: ");

    initializeVisited(visited, numVertices);

    DFS(graph, 0, visited);


    return 0;

}
```

1. Implement 0/1 Knapsack algorithm.

```c
#include <stdio.h>
int max(int a, int b) {
    return (a > b) ? a : b;
}
int knapsack(int max_weight, int weights[], int values[], int num_items) {
    int dp[num_items + 1][max_weight + 1];
    for (int i = 0; i<= num_items; i++) {
        for (int w = 0; w <= max_weight; w++) {
            if (i == 0 || w == 0)
dp[i][w] = 0;
            else if (weights[i - 1] <= w)
dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
            else
dp[i][w] = dp[i - 1][w];
        }
    }
    return dp[num_items][max_weight];
}
int main() {
    int num_items;
printf("Enter the number of items: ");
scanf("%d", &num_items);
    int weights[num_items];
    int values[num_items];
printf("Enter the weight and value of each item:\n");
    for (int i = 0; i<num_items; i++) {
printf("Item %d:\n", i + 1);
scanf("%d", &weights[i]);
scanf("%d", &values[i]);
    }
    int max_weight;
printf("Enter the maximum weight: ");
scanf("%d", &max_weight);
    int max_value = knapsack(max_weight, weights, values, num_items);
printf("Maximum value: %d\n", max_value);
    return 0;
}
```

2. Alice and Bob:

```c
#include <stdio.h>
#include <stdbool.h>
bool canWin(int n) {
    if (n <= 1) {
        return false;
    }
    for (int x = 1; x < n; x++) {
        if (n % x == 0) {
            if (!canWin(n - x)) {
                return true;
            }
        }
    }
    return false;
}
int main() {
    int n;
printf("Enter the initial number: ");
scanf("%d", &n);
    bool aliceWins = canWin(n);
    if (aliceWins) {
printf("true");
    } else {
printf("false");
    }
    return 0;
}
```

3. Implement Matrix Chain multiplication algorithm with top-down approach.

```c
#include <stdio.h>
#include <limits.h>
#define MAX_SIZE 100
int matrixChainMultiplication(int dimensions[], int i, int j, int dp[][MAX_SIZE]) {
    if (i == j) {
        return 0;
    }
    if (dp[i][j] != -1) {
        return dp[i][j];
    }
dp[i][j] = INT_MAX;
    for (int k = i; k < j; k++) {
        int cost = matrixChainMultiplication(dimensions, i, k, dp) +
matrixChainMultiplication(dimensions, k + 1, j, dp) +
                dimensions[i - 1] * dimensions[k] * dimensions[j];
        if (cost <dp[i][j]) {
dp[i][j] = cost;
        }
    }
    return dp[i][j];
}
int main() {
    int numMatrices;
printf("Enter the number of matrices: ");
scanf("%d", &numMatrices);
    int dimensions[numMatrices + 1];
printf("Enter the dimensions of the matrices:\n");
    for (int i = 0; i<= numMatrices; i++) {
scanf("%d", &dimensions[i]);
    }
    int dp[MAX_SIZE][MAX_SIZE];
    for (int i = 0; i< MAX_SIZE; i++) {
        for (int j = 0; j < MAX_SIZE; j++) {
dp[i][j] = -1;
        }
    }
    int minimumCost = matrixChainMultiplication(dimensions, 1, numMatrices, dp);
printf("Minimum number of multiplications: %d\n", minimumCost);
    return 0;
}
```

4. Write a program to find minimum change to return when unlimited number of denominations are available using Dynamic programming.

```c
#include <stdio.h>
#include <limits.h>
int minCoins(int coins[], int numCoins, int amount) {
    int dp[amount + 1];
dp[0] = 0;
    for (int i = 1; i<= amount; i++) {
dp[i] = INT_MAX;
    }
    for (int i = 1; i<= amount; i++) {
        for (int j = 0; j <numCoins; j++) {
            if (coins[j] <= i) {
                int subproblem = dp[i - coins[j]];
                if (subproblem != INT_MAX && subproblem + 1 <dp[i]) {
dp[i] = subproblem + 1;
                }
            }
        }
    }
    return dp[amount];
}
int main() {
    int numCoins;
printf("Enter the number of coin denominations: ");
scanf("%d", &numCoins);
    int coins[numCoins];
printf("Enter the coin denominations:\n");
    for (int i = 0; i<numCoins; i++) {
scanf("%d", &coins[i]);
    }
    int amount;
printf("Enter the amount for which to make change: ");
scanf("%d", &amount);
    int minNumCoins = minCoins(coins, numCoins, amount);
printf("Minimum number of coins needed: %d\n", minNumCoins);
    return 0;
}
```

5. Implement the LCS problem using dynamic programming

```c
#include <stdio.h>
#include <string.h>
#define MAX_SIZE 100
int max(int a, int b) {
    return (a > b) ? a : b;
}
int lcs(char* str1, char* str2, int len1, int len2) {
    int dp[MAX_SIZE + 1][MAX_SIZE + 1];
    for (int i = 0; i<= len1; i++) {
        for (int j = 0; j <= len2; j++) {
            if (i == 0 || j == 0)
dp[i][j] = 0;
            else if (str1[i - 1] == str2[j - 1])
dp[i][j] = dp[i - 1][j - 1] + 1;
            else
dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
    return dp[len1][len2];
}
int main() {
    char str1[MAX_SIZE];
    char str2[MAX_SIZE];
printf("Enter the first string: ");
scanf("%s", str1);
printf("Enter the second string: ");
scanf("%s", str2);
    int len1 = strlen(str1);
    int len2 = strlen(str2);
    int lcsLength = lcs(str1, str2, len1, len2);
printf("Length of Longest Common Subsequence: %d\n", lcsLength);
    return 0;
}
```

6. Implement a program to find longest increasing subsequence.

```c
#include <stdio.h>
#include <stdlib.h>
int lis(int arr[], int n) {
    int* dp = (int*)malloc(sizeof(int) * n);
    int maxLen = 0;
    for (int i = 0; i< n; i++) {
dp[i] = 1;
    }
    for (int i = 1; i< n; i++) {
        for (int j = 0; j <i; j++) {
            if (arr[i] >arr[j] &&dp[i] <dp[j] + 1) {
dp[i] = dp[j] + 1;
            }
        }
    }
    for (int i = 0; i< n; i++) {
        if (dp[i] >maxLen) {
maxLen = dp[i];
        }
    }
    free(dp);
    return maxLen;
}
int main() {
    int n;
printf("Enter the number of elements in the array: ");
scanf("%d", &n);
    int arr[n];
printf("Enter the elements of the array:\n");
    for (int i = 0; i< n; i++) {
scanf("%d", &arr[i]);
    }
    int length = lis(arr, n);
printf("Length of Longest Increasing Subsequence: %d\n", length);
    return 0;
}
```

7. Implement Matrix Chain multiplication algorithm with bottom-up approach.

```c
#include <stdio.h>
#include <limits.h>
#define MAX_SIZE 100
int min(int a, int b) {
   return (a < b) ? a : b;
}
int matrixChainMultiplication(int dimensions[], int numMatrices) {
   int dp[MAX_SIZE][MAX_SIZE];
   for (int i = 0; i<= numMatrices; i++) {
dp[i][i] = 0;
   }
   for (int length = 2; length <= numMatrices; length++) {
      for (int i = 1; i<= numMatrices - length + 1; i++) {
         int j = i + length - 1;
dp[i][j] = INT_MAX;
         for (int k = i; k < j; k++) {
            int cost = dp[i][k] + dp[k + 1][j] +
                   dimensions[i - 1] * dimensions[k] * dimensions[j];
dp[i][j] = min(dp[i][j], cost);
         }
      }
   }
   return dp[1][numMatrices];
}
int main() {
   int numMatrices;
printf("Enter the number of matrices: ");
scanf("%d", &numMatrices);
   int dimensions[numMatrices + 1];
printf("Enter the dimensions of the matrices:\n");
   for (int i = 0; i<= numMatrices; i++) {
scanf("%d", &dimensions[i]);
   }
   int minimumCost = matrixChainMultiplication(dimensions, numMatrices);
printf("Minimum number of multiplications: %d\n", minimumCost);
   return 0;
}
```

8. Write a program to find subset sum by using Dynamic programming

```c
#include <stdio.h>
#include <stdbool.h>
#define MAX_SIZE 100
bool subsetSum(int set[], int n, int sum) {
    bool dp[MAX_SIZE + 1][MAX_SIZE + 1];
    for (int i = 0; i<= n; i++) {
dp[i][0] = true;
    }
    for (int j = 1; j <= sum; j++) {
dp[0][j] = false;
    }
    for (int i = 1; i<= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (set[i - 1] > j) {
dp[i][j] = dp[i - 1][j];
            }
            else {
dp[i][j] = dp[i - 1][j] || dp[i - 1][j - set[i - 1]];
            }
        }
    }
    return dp[n][sum];
}
int main() {
    int n;
printf("Enter the number of elements in the set: ");
scanf("%d", &n);
    int set[n];
printf("Enter the elements of the set:\n");
    for (int i = 0; i< n; i++) {
scanf("%d", &set[i]);
    }
    int sum;
printf("Enter the target sum: ");
scanf("%d", &sum);
    bool exists = subsetSum(set, n, sum);
    if (exists) {
printf("Subset with the given sum exists.\n");
    } else {
printf("No subset with the given sum exists.\n");
    }
    return 0;
}
```

9. Implement of N-queens problem with Back tracking.

```c
#include <stdio.h>
#include <stdbool.h>
#define N 8
void printSolution(int board[N][N]) {
    for (int i = 0; i< N; i++) {
        for (int j = 0; j < N; j++) {
printf("%c ", board[i][j] ? 'Q' : '.');
        }
printf("\n");
    }
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; i< N && j >= 0; i++, j--) {
        if (board[i][j])
            return false;
    }
    return true;
}
bool solveNQueensUtil(int board[N][N], int col) {
    if (col == N) {
printSolution(board);
        return true;
    }
    bool res = false;
    for (int i = 0; i< N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            res = solveNQueensUtil(board, col + 1) || res;
            board[i][col] = 0;
        }
    }
    return res;
}
void solveNQueens() {
    int board[N][N] = {0};
    if (!solveNQueensUtil(board, 0)) {
printf("No solution found.\n");
    }
}
int main() {
solveNQueens();
    return 0;
}
```

10.Implement Sum of subsets problem by using Backtracking

```c
#include <stdio.h>
#include <stdbool.h>
void printSubset(int set[], int subset[], int n) {
printf("Subset: ");
    for (int i = 0; i< n; i++) {
        if (subset[i])
printf("%d ", set[i]);
    }
printf("\n");
}
void subsetSumUtil(int set[], int subset[], int n, int sum, int currentSum, int index) {
    if (currentSum == sum) {
printSubset(set, subset, n);
        return;
    }
    if (index == n)
        return;
    if (currentSum + set[index] <= sum) {
        subset[index] = 1;
subsetSumUtil(set, subset, n, sum, currentSum + set[index], index + 1);
        subset[index] = 0;
    }
subsetSumUtil(set, subset, n, sum, currentSum, index + 1);
}
void subsetSum(int set[], int n, int sum) {
    int subset[n];
subsetSumUtil(set, subset, n, sum, 0, 0);
}
int main() {
    int n;
printf("Enter the number of elements in the set: ");
scanf("%d", &n);
    int set[n];
printf("Enter the elements of the set:\n");
    for (int i = 0; i< n; i++) {
scanf("%d", &set[i]);
    }
    int sum;
printf("Enter the target sum: ");
scanf("%d", &sum);
subsetSum(set, n, sum);
    return 0;
}
```

11. Implement Graph coloring problem with back tracking.

```c
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 20
bool isSafe(int v, int graph[MAX_VERTICES][MAX_VERTICES], int vertices[], int color, int n) {
    for (int i = 0; i< n; i++) {
        if (graph[v][i] && color == vertices[i])
            return false;
    }
    return true;
}
bool graphColoringUtil(int graph[MAX_VERTICES][MAX_VERTICES], int m, int vertices[], int v, int n) {
    if (v == n)
        return true;
    for (int color = 1; color <= m; color++) {
        if (isSafe(v, graph, vertices, color, n)) {
            vertices[v] = color;
            if (graphColoringUtil(graph, m, vertices, v + 1, n))
                return true;
            vertices[v] = 0;
        }
    }
    return false;
}
void graphColoring(int graph[MAX_VERTICES][MAX_VERTICES], int m, int n) {
    int vertices[MAX_VERTICES] = {0};
    if (graphColoringUtil(graph, m, vertices, 0, n)) {
printf("Graph can be colored using at most %d colors.\n", m);
printf("Coloring: ");
        for (int i = 0; i< n; i++) {
printf("%d ", vertices[i]);
        }
printf("\n");
    } else {
printf("Graph cannot be colored using %d colors.\n", m);
    }
}
int main() {
    int n, m;
printf("Enter the number of vertices in the graph: ");
scanf("%d", &n);
printf("Enter the adjacency matrix of the graph:\n");
    int graph[MAX_VERTICES][MAX_VERTICES];
    for (int i = 0; i< n; i++) {
        for (int j = 0; j < n; j++) {
scanf("%d", &graph[i][j]);
        }
    }
printf("Enter the number of colors available: ");
scanf("%d", &m);
graphColoring(graph, m, n);
    return 0;
}
```

12. Implement a program to find Hamiltonian cycle from a given graph

```c
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 20
void printSolution(int path[], int n) {
printf("Hamiltonian Cycle: ");
   for (int i = 0; i< n; i++) {
printf("%d ", path[i]);
   }
printf("%d\n", path[0]);
}
bool isSafe(int v, int graph[MAX_VERTICES][MAX_VERTICES], int path[], int pos, int n) {
   if (graph[path[pos - 1]][v] == 0)
      return false;
   for (int i = 0; i< pos; i++) {
      if (path[i] == v)
         return false;
   }
   return true;
}
bool hamiltonianCycleUtil(int graph[MAX_VERTICES][MAX_VERTICES], int path[], int pos, int n) {
   if (pos == n) {
      if (graph[path[pos - 1]][path[0]] == 1)
         return true;
      else
         return false;
   }
   for (int v = 1; v < n; v++) {
      if (isSafe(v, graph, path, pos, n)) {
         path[pos] = v;
         if (hamiltonianCycleUtil(graph, path, pos + 1, n))
            return true;
         path[pos] = -1;  // Backtrack
      }
   }
   return false;
}
void hamiltonianCycle(int graph[MAX_VERTICES][MAX_VERTICES], int n) {
   int path[MAX_VERTICES];
   for (int i = 0; i< n; i++) {
      path[i] = -1;
   }
   path[0] = 0;
   if (hamiltonianCycleUtil(graph, path, 1, n)) {
printSolution(path, n);
   } else {
printf("No Hamiltonian Cycle found.\n");
   }
}
int main() {
   int n;
printf("Enter the number of vertices in the graph: ");
scanf("%d", &n);
printf("Enter the adjacency matrix of the graph:\n");
   int graph[MAX_VERTICES][MAX_VERTICES];
   for (int i = 0; i< n; i++) {
      for (int j = 0; j < n; j++) {
scanf("%d", &graph[i][j]);
      }
   } hamiltonianCycle(graph, n); return 0;}
```

13.Implement TCS by branch and bound:

```c
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#define MAX_N 10
int n;
int graph[MAX_N][MAX_N];
int minCost = INT_MAX;
int bestPath[MAX_N];
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int calculatePathCost(int path[]) {
    int cost = 0;
    for (int i = 0; i< n - 1; i++) {
        cost += graph[path[i]][path[i + 1]];
    }
    cost += graph[path[n - 1]][path[0]];
    return cost;
}
void TSPBranchAndBoundUtil(int path[], bool visited[], int level, int cost) {
    if (level == n) {
        int currentCost = cost + graph[path[level - 1]][path[0]];
        if (currentCost<minCost) {
minCost = currentCost;
            for (int i = 0; i< n; i++) {
bestPath[i] = path[i];
            }
        }
        return;
    }
    for (int i = 0; i< n; i++) {
        if (!visited[i]) {
            path[level] = i;
            visited[i] = true;
            int newCost = cost + graph[path[level - 1]][i];
            int lowerBound = 0;
            for (int j = 0; j < n; j++) {
                if (!visited[j]) {
                    int minEdgeCost = INT_MAX;
                    for (int k = 0; k < n; k++) {
                        if (graph[j][k] <minEdgeCost&& j != k) {
minEdgeCost = graph[j][k];
                        }
                    }
lowerBound += minEdgeCost;
                }
            }
            if (newCost + lowerBound<minCost) {
TSPBranchAndBoundUtil(path, visited, level + 1, newCost);
            }
            visited[i] = false;
        }
    }
}
void TSPBranchAndBound(int startingCity) {
    int path[MAX_N];
    bool visited[MAX_N];
```

```c
    for (int i = 0; i< n; i++) {
        visited[i] = false;
    }
    path[0] = startingCity;
    visited[startingCity] = true;
TSPBranchAndBoundUtil(path, visited, 1, 0);
printf("Optimal TSP Path: ");
    for (int i = 0; i< n; i++) {
printf("%d ", bestPath[i]);
    }
printf("%d\n", bestPath[0]);
printf("Optimal TSP Cost: %d\n", minCost);
}
int main() {
printf("Enter the number of cities: ");
scanf("%d", &n);
printf("Enter the adjacency matrix of distances between cities:\n");
    for (int i = 0; i< n; i++) {
        for (int j = 0; j < n; j++) {
scanf("%d", &graph[i][j]);
        }
    }
    int startingCity;
printf("Enter the starting city (0-%d): ", n - 1);
scanf("%d", &startingCity);
TSPBranchAndBound(startingCity);
    return 0;
}
```

14.Implement the Dijkstra's single source shortest paths algorithm.

```c
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#define MAX_VERTICES 20
int graph[MAX_VERTICES][MAX_VERTICES];
int n;
void dijkstra(int source) {
    int dist[MAX_VERTICES];
    bool visited[MAX_VERTICES];
    int parent[MAX_VERTICES];

    for (int i = 0; i< n; i++) {
dist[i] = INT_MAX;
        visited[i] = false;
        parent[i] = -1;
    }
dist[source] = 0;
    for (int count = 0; count < n - 1; count++) {
        int minDist = INT_MAX, minDistVertex;
        for (int v = 0; v < n; v++) {
            if (!visited[v] &&dist[v] <minDist) {
minDist = dist[v];
minDistVertex = v;
            }
        }
        visited[minDistVertex] = true;
        for (int v = 0; v < n; v++) {
            if (!visited[v] && graph[minDistVertex][v] != 0 &&dist[minDistVertex] != INT_MAX &&
dist[minDistVertex] + graph[minDistVertex][v] <dist[v]) {
dist[v] = dist[minDistVertex] + graph[minDistVertex][v];
                parent[v] = minDistVertex;
            }
        }
    }
printf("Vertex\tDistance\tPath\n");
    for (int v = 0; v < n; v++) {
printf("%d\t%d\t\t%d", v, dist[v], v);
        int p = parent[v];
        while (p != -1) {
printf(" <- %d", p);
            p = parent[p];
        }
printf("\n");
    }
}
int main() {
printf("Enter the number of vertices in the graph: ");
scanf("%d", &n);
printf("Enter the adjacency matrix of the graph (0 for no edge, positive weight for edge):\n");
    for (int i = 0; i< n; i++) {
        for (int j = 0; j < n; j++) {
scanf("%d", &graph[i][j]);
        }
```

```c
    }
    int source;
printf("Enter the source vertex: ");
scanf("%d", &source);
dijkstra(source);
    return 0;
}
```

15. Implement 0/1 knapsack by branch and bound.

```c
#include <stdio.h>
#include <stdbool.h>
#define MAX_ITEMS 100
typedef struct {
    int weight;
    int value;
} Item;
int maxProfit = 0;
int bestItems[MAX_ITEMS];
int numItems;
int capacity;
void branchAndBoundKnapsack(Item items[], int level, int currentWeight, int currentValue, bool selected[]) {
    if (currentWeight> capacity) {
        return;
    }
    if (currentValue>maxProfit) {
maxProfit = currentValue;
        for (int i = 0; i<numItems; i++) {
bestItems[i] = selected[i];
        }
    }
    if (level == numItems) {
        return;
    }
    double bound = currentValue;
    int remainingWeight = capacity - currentWeight;
    int i = level;
    while (i<numItems&&remainingWeight> 0) {
        if (items[i].weight <= remainingWeight) {
            bound += items[i].value;
remainingWeight -= items[i].weight;
        } else {
            bound += (double)items[i].value / items[i].weight * remainingWeight;
remainingWeight = 0;
        }
i++;
    }
    if (bound <= maxProfit) {
        return;
    }
    selected[level] = true;
branchAndBoundKnapsack(items, level + 1, currentWeight + items[level].weight, currentValue +
items[level].value, selected);
    selected[level] = false;
branchAndBoundKnapsack(items, level + 1, currentWeight, currentValue, selected);
}
void knapsack(Item items[], int num, int cap) {
numItems = num;
    capacity = cap;
    bool selected[MAX_ITEMS] = { false };
branchAndBoundKnapsack(items, 0, 0, 0, selected);
printf("Optimal Items: ");
    for (int i = 0; i<numItems; i++) {
        if (bestItems[i]) {
printf("%d ", i);
        }
    }
printf("\n");
printf("Optimal Profit: %d\n", maxProfit);
```

```c
}
int main() {
    int num, cap;
printf("Enter the number of items: ");
scanf("%d", &num);
printf("Enter the capacity of the knapsack: ");
scanf("%d", &cap);
    Item items[MAX_ITEMS];
printf("Enter the weight and value of each item:\n");
    for (int i = 0; i<num; i++) {
scanf("%d %d", &items[i].weight, &items[i].value);
    }
    knapsack(items, num, cap);
    return 0;
}
```

16. Implement Bellman ford single source shortest paths algorithm

```c
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#define MAX_VERTICES 20
#define MAX_EDGES 100
typedef struct {
    int source;
    int destination;
    int weight;
} Edge;
int numVertices, numEdges;
Edge edges[MAX_EDGES];
int distances[MAX_VERTICES];
void bellmanFord(int source) {
    for (int i = 0; i<numVertices; i++) {
        if (i == source) {
            distances[i] = 0;
        } else {
            distances[i] = INT_MAX;
        }
    }
    for (int i = 0; i<numVertices - 1; i++) {
        for (int j = 0; j <numEdges; j++) {
            int u = edges[j].source;
            int v = edges[j].destination;
            int weight = edges[j].weight;
            if (distances[u] != INT_MAX && distances[u] + weight < distances[v]) {
                distances[v] = distances[u] + weight;
            }
        }
    }
    for (int i = 0; i<numEdges; i++) {
        int u = edges[i].source;
        int v = edges[i].destination;
        int weight = edges[i].weight;
        if (distances[u] != INT_MAX && distances[u] + weight < distances[v]) {
printf("Negative-weight cycle detected. The graph contains a negative-weight cycle.\n");
            return;
        }
    }
printf("Vertex\tDistance from Source\n");
    for (int i = 0; i<numVertices; i++) {
printf("%d\t%d\n", i, distances[i]);
    }
}
int main() {
printf("Enter the number of vertices: ");
scanf("%d", &numVertices);
printf("Enter the number of edges: ");
scanf("%d", &numEdges);
printf("Enter the edges (source, destination, weight):\n");
    for (int i = 0; i<numEdges; i++) {
scanf("%d %d %d", &edges[i].source, &edges[i].destination, &edges[i].weight);
    }
    int source;
printf("Enter the source vertex: ");
scanf("%d", &source);
bellmanFord(source); return 0;}
```