**Doubly linked list**

**Graph->dfs,bfs,adjacency list**

#include<stdio.h>

#include<stdlib.h>

int no\_vertices;

struct node

{

int data;

struct node \*next;

};

void readgraph(struct node \*ad[]);

void printgraph(struct node \*ad[]);

void bfs(struct node \*ad[],int start,int visited[]);

void dfs(struct node \*ad[],int start,int visited[]);

int main()

{

int i,j,k,ch,start;

printf("Enter the total number of vertex :");

scanf("%d",&no\_vertices);

int visited[no\_vertices];

struct node \*adj[no\_vertices];

for(i=0;i<no\_vertices;i++)

{

adj[i] = NULL;

}

readgraph(adj);

do{

printf("\n Enter 1 for BFS\n Enter 2 for DFS\n Enter 3 to Print the adjacency list\nEnter 4 to Exit : ");

scanf("%d",&ch);

switch(ch)

{

case 1:

printf("Enter the vertex from which do you wanted to start :");

scanf("%d",&start);

for(int k=0;k<no\_vertices;k++)

visited[k]=0;

bfs(adj,start,visited); break;

case 2:

printf("Enter the vertex from which do you wanted to start :");

scanf("%d",&start);

for(int k=0;k<no\_vertices;k++)

visited[k]=0;

printf("DFS : ");

dfs(adj,start,visited);

break;

case 3: printgraph(adj);break;

case 4: break;

}

}while(ch!= 4);

return 0;

}

void readgraph(struct node \*ad[])

{

struct node \*newnode;

int i,j,k,data;

for(i=0;i<no\_vertices;i++)

{

struct node \*last =NULL;

printf("\nEnter the Number of neighbours of %d :",i);

scanf("%d",&k);

for(j=0;j<k;j++)

{

printf("Enter the value of %d neighbour of %d : ",j,i);

scanf("%d",&data);

newnode = (struct node\*)malloc(sizeof(struct node\*));

newnode->data = data;

newnode->next = NULL;

if(ad[i]==NULL)

{

ad[i] = newnode;

}

else

last->next = newnode;

last = newnode;

}

}

}

void printgraph(struct node \*ad[])

{

struct node \*ptr = NULL;

int i,j;

for(i=0;i<no\_vertices;i++)

{

ptr = ad[i];

printf("\n The neighbour of %d are :",i);

while(ptr != NULL)

{

printf("%d\t",ptr->data);

ptr = ptr->next;

}

}

}

void bfs(struct node \*ad[],int start,int visited[])

{

int queue[no\_vertices],front=-1,rear=-1;

printf("BFS : ");

queue[++rear] = start;

visited[start]=1;

front++;

while(front<=rear)

{

printf("%d\t",queue[front]);

struct node \*ptr;

ptr = ad[queue[front++]];

while(ptr != NULL)

{

if(visited[ptr->data]!=1)

{

queue[++rear] = ptr->data;

visited[ptr->data] =1;

}

ptr = ptr->next;

}

}

}

void dfs(struct node \*ad[],int start, int visited[])

{

visited[start] = 1;

printf("%d\t",start);

struct node \*ptr;

ptr = ad[start];

while(ptr!=NULL)

{

if(visited[ptr->data]==0)

dfs(ad,ptr->data,visited);

ptr = ptr->next;

}

}

OUTPUT

Enter the total numbe of vertex :5

Enter the Number of neighbours of 0 :3

Enter the value of 0 neighbour of 0 : 1

Enter the value of 1 neighbour of 0 : 2

Enter the value of 2 neighbour of 0 : 4

Enter the Number of neighbours of 1 :2

Enter the value of 0 neighbour of 1 : 0

Enter the value of 1 neighbour of 1 : 2

Enter the Number of neighbours of 2 :3

Enter the value of 0 neighbour of 2 : 0

Enter the value of 1 neighbour of 2 : 1

Enter the value of 2 neighbour of 2 : 3

Enter the Number of neighbours of 3 :1

Enter the value of 0 neighbour of 3 : 2

Enter the Number of neighbours of 4 :1

Enter the value of 0 neighbour of 4 : 0

Enter 1 for BFS

Enter 2 for DFS

Enter 3 to Print the adjacency list

Enter 4 to Exit : 3

The neighbour of 0 are :1 2 4

The neighbour of 1 are :0 2

The neighbour of 2 are :0 1 3

The neighbour of 3 are :2

The neighbour of 4 are :0

Enter 1 for BFS

Enter 2 for DFS

Enter 3 to Print the adjacency list

Enter 4 to Exit : 1

Enter the vertex from which do you wanted to start :0

BFS : 0 1 2 4 3

Enter 1 for BFS

Enter 2 for DFS

Enter 3 to Print the adjacency list

Enter 4 to Exit : 2

Enter the vertex from which do you wanted to start :2

DFS : 2 0 1 4 3

**GCD->CONSECUTIVE NUMBER**

#include<stdio.h>

#include<stdlib.h>

int gcd(num1,num2);

void main()

{

int num1,num2;

printf("enter the two numbers to find the gcd of it\n");

scanf("%d",&num1);

scanf("%d",&num2);

int ans=0;

ans=gcd(num1,num2);

printf("the GCD of the number is %d",ans);

}

int gcd(int n1,int n2)

,

int min;

if(n1<n2)

min=n1;

else

min=n2;

while(min!=0)

{

if(n2%min==0)

{

if(n1%min==0)

return min;

}

else

min=min-1;

    }

}

FIBONACCI->recursively

#include<stdlib.h>

#include<stdlib.h>

int fib(int num);

void main()

{

int num,i;

printf("enter the size of the sequence\n ");

scanf("%d",&num);

printf("the series is ");

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

printf(" %d",fib(i));

}

}

int fib(int n)

{

if(n==0)

return 0;

if(n==1)

return 1;

else

return fib(n-1)+fib(n-2);

}

**ADJACENCY MATRIX OF GRAPH**

#include<stdio.h>

int no\_vertices;

void printgraph(int adj[][no\_vertices])

{

for(int i=0;i<no\_vertices;i++)

{

for (int j=0;j<no\_vertices;j++)

{

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

}

printf("\n");

}

}

int main()

{

int s,d;

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

scanf("%d",&no\_vertices);

int adj[no\_vertices][no\_vertices];

for(int i=0;i<no\_vertices;i++)

for (int j=0;j<no\_vertices;j++)

adj[i][j]=0;

while(s!=-1 &&d!=-1)

{

printf("\n enter an edge from node (0 to %d) to node (0 to %d) :",no\_vertices,no\_vertices);

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

adj[s][d]=1;

adj[d][s]=1;

}

printgraph(adj);

return 0;

}

OUTPUT

enter the number of vertices :5

enter an edge from node (0 to 5) to node (0 to 5) :0 1

enter an edge from node (0 to 5) to node (0 to 5) :1 2

enter an edge from node (0 to 5) to node (0 to 5) :2 3

enter an edge from node (0 to 5) to node (0 to 5) :1 2

enter an edge from node (0 to 5) to node (0 to 5) :2 3

enter an edge from node (0 to 5) to node (0 to 5) :-1 -1

0 1 0 0 0

1 0 1 0 0

0 1 0 1 0

0 0 1 0 0

0 0 0 0 0

**EUCLID ALGORITHM**

#include <stdio.h>

unsigned int EuclidGCD(unsigned int m, unsigned int n) {

unsigned int r;

int opcount = 0;

while (n != 0) {

opcount++; // Increment operation count

r = m % n;

m = n;

n = r;

}

printf("Operation count= %d\n", opcount);

return m; // Return the GCD

}

int main() {

unsigned int m, n;

printf("Enter the two numbers whose GCD has to be calculated: ");

scanf("%d", &m);

scanf("%d", &n);

printf("GCD of two numbers using Euclid's method is %d", EuclidGCD(m, n));

return 0;

}

**No**

**TOWER OF HANOI**

#include <stdio.h>

#include <math.h>

#include <time.h>

int basicOpCount = 0;

void towerOfHanoi(int n, char source, char destination, char auxiliary) {

if (n == 1) {

basicOpCount++;

printf("Move disk 1 from %c to %c\n", source, destination);

return;

}

towerOfHanoi(n - 1, source, auxiliary, destination);

basicOpCount++;

printf("Move disk %d from %c to %c\n", n, source, destination);

towerOfHanoi(n - 1, auxiliary, destination, source);

}

int main() {

int n;

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

scanf("%d", &n);

clock\_t start = clock();

printf("Steps to solve Tower of Hanoi with %d disks:\n", n);

towerOfHanoi(n, 'A', 'C', 'B');

clock\_t end = clock();

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Number of basic operations: %d\n", basicOpCount);

printf("Time taken: %f seconds\n", time\_taken);

return 0;

}

No

Write a program to find GCD using middle school method and analyze its time efficiency.

#include <stdio.h>

#include <math.h>

#include <stdbool.h>

#include <time.h>

// Function to check if a number is prime

bool isPrime(int n) {

if (n <= 1) return false;

if (n <= 3) return true;

if (n % 2 == 0 || n % 3 == 0) return false;

for (int i = 5; i \* i <= n; i += 6) {

if (n % i == 0 || n % (i + 2) == 0) return false;

}

return true;

}

// Function to find prime factors of a number

void primeFactors(int n, int \*factors, int \*count) {

int index = 0;

while (n % 2 == 0) {

factors[index++] = 2;

(\*count)++;

n /= 2;

}

for (int i = 3; i <= sqrt(n); i += 2) {

while (n % i == 0) {

factors[index++] = i;

(\*count)++;

n /= i;

}

}

if (n > 2) {

factors[index++] = n;

(\*count)++;

}

}

// Function to find the GCD using prime factors

unsigned int GCDByPrimeFactors(int m, int n, int \*opcount) {

int factorsM[100], factorsN[100], countM = 0, countN = 0;

primeFactors(m, factorsM, &countM);

primeFactors(n, factorsN, &countN);

int gcd = 1;

int i = 0, j = 0;

while (i < countM && j < countN) {

(\*opcount)++; // Increment operation count

if (factorsM[i] == factorsN[j]) {

gcd \*= factorsM[i];

i++;

j++;

} else if (factorsM[i] < factorsN[j]) {

i++;

} else {

j++;

}

}

return gcd;

}

int main() {

int m, n;

// Input the two numbers

printf("Enter the two numbers whose GCD has to be calculated: ");

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

// Measure start time

clock\_t start = clock();

// Initialize operation count

int opcount = 0;

// Calculate and print the GCD using the prime factors method

printf("GCD of %d and %d using prime factors method: %d\n", m, n, GCDByPrimeFactors(m, n, &opcount));

// Measure end time

clock\_t end = clock();

// Calculate the time taken

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Operation count: %d\n", opcount); // Print operation count

printf("Time taken: %f seconds\n", time\_taken);

return 0;

}

**No**

**Write a program to delete strong numbers from an array using recursion [A strong number is such that the sum of it's factorial is the number itself]**

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

// Function to calculate factorial recursively

int factorial(int n) {

if (n == 0 || n == 1)

return 1;

return n \* factorial(n - 1);

}

// Function to check if a number is strong

bool isStrong(int num) {

int temp = num;

int sum = 0;

while (temp != 0) {

sum += factorial(temp % 10);

temp /= 10;

}

return sum == num;

}

// Function to delete strong numbers from an array recursively

void deleteStrong(int arr[], int \*size, int index, int \*opcount) {

if (index == \*size)

return;

if (isStrong(arr[index])) {

for (int i = index; i < (\*size) - 1; i++) {

arr[i] = arr[i + 1];

(\*opcount)++; // Increment operation count

}

(\*size)--; // Decrement size of the array

deleteStrong(arr, size, index, opcount); // Check the same index again since the element at that index has changed

} else {

deleteStrong(arr, size, index + 1, opcount); // Move to the next index

(\*opcount)++; // Increment operation count

}

}

int main() {

int size;

// Input the size of the array

printf("Enter the size of the array: ");

scanf("%d", &size);

int arr[size];

// Input array elements

printf("Enter %d numbers:\n", size);

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

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

}

// Measure start time

clock\_t start = clock();

// Initialize operation count

int opcount = 0;

// Delete strong numbers recursively

deleteStrong(arr, &size, 0, &opcount);

// Measure end time

clock\_t end = clock();

// Calculate the time taken

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

// Print the array after deleting strong numbers

printf("Array after deleting strong numbers: ");

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

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

}

printf("\n");

// Print operation count and time taken

printf("Operation count: %d\n", opcount);

printf("Time taken: %f seconds\n", time\_taken);

return 0;

}

**Selection sort**

#include <stdio.h>

#include <stdlib.h>

void selectionSort(int \*a, unsigned int n, int \*opcount) {

unsigned int i, j, min;

int temp;

\*opcount = 0; // Reset operation count

for (i = 0; i < n - 1; i++) {

min = i;

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

(\*opcount)++; // Increment opcount for every comparison

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

min = j;

}

temp = a[i];

a[i] = a[min];

a[min] = temp;

}

}

int main() {

int \*a;

int i, n;

printf("Enter the size of the array: ");

scanf("%d", &n);

a = (int \*)malloc(sizeof(int) \* n);

printf("Enter %d elements of the array:\n", n);

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

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

}

// Perform selection sort and count comparisons

int opcount;

selectionSort(a, n, &opcount);

printf("\nElements after selection sort: ");

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

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

printf("\n");

printf("Operation count: %d\n", opcount); // Print operation count

free(a);

return 0;

}

**Write a program to sort set of integers using bubble sort. Analyze its time efficiency.**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Function to perform bubble sort and count the number of comparisons

void bubbleSort(int \*a, int n, int \*opcount) {

\*opcount = 0; // Reset operation count

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

for (int j = 0; j < n - i - 1; j++) {

(\*opcount)++; // Increment opcount for every comparison

if (a[j] > a[j + 1]) {

// Swap if the current element is greater than the next element

int temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

}

}

}

}

int main() {

int \*a;

int n;

// Input the size of the array

printf("Enter the size of the array: ");

scanf("%d", &n);

// Allocate memory for the array

a = (int \*)malloc(sizeof(int) \* n);

if (a == NULL) {

printf("Memory allocation failed.\n");

return 1;

}

// Input array elements from the user

printf("Enter %d elements of the array:\n", n);

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

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

}

// Measure start time

clock\_t start = clock();

// Perform bubble sort and count comparisons

int opcount;

bubbleSort(a, n, &opcount);

// Measure end time

clock\_t end = clock();

// Calculate the time taken

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

// Print the sorted array

printf("\nElements after bubble sort: ");

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

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

printf("\n");

// Print the operation count

printf("Operation count: %d\n", opcount);

// Print the time taken for sorting

printf("Time taken: %f seconds\n", time\_taken);

// Free dynamically allocated memory

free(a);

return 0;

}

**STRING MATCHING**

#include <stdio.h>

#include <string.h>

#include <time.h>

// Function to perform brute-force string matching

int bruteForceStringMatch(char \*text, char \*pattern, int \*opcount) {

int n = strlen(text);

int m = strlen(pattern);

\*opcount = 0; // Reset operation count

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

int j;

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

(\*opcount)++; // Increment opcount for every comparison

if (text[i + j] != pattern[j])

break;

}

if (j == m)

return i; // Pattern found at index i in the text

}

return -1; // Pattern not found in the text

}

int main() {

char text[100], pattern[100];

int opcount;

// Input the text

printf("Enter the text: ");

fgets(text, sizeof(text), stdin);

text[strcspn(text, "\n")] = '\0'; // Remove newline character

// Input the pattern

printf("Enter the pattern: ");

fgets(pattern, sizeof(pattern), stdin);

pattern[strcspn(pattern, "\n")] = '\0'; // Remove newline character

// Measure start time

clock\_t start = clock();

// Perform brute-force string matching and count comparisons

int index = bruteForceStringMatch(text, pattern, &opcount);

// Measure end time

clock\_t end = clock();

// Calculate the time taken

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

// Print the result

if (index != -1)

printf("Pattern found at index %d in the text.\n", index);

else

printf("Pattern not found in the text.\n");

// Print operation count

printf("Operation count: %d\n", opcount);

// Print time taken

printf("Time taken: %f seconds\n", time\_taken);

return 0;

}

**No**

**Write a program to implement matrix multiplication using brute-force technique and analyze its time efficiency.**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Function to perform matrix multiplication

void matrixMultiplication(int \*\*A, int \*\*B, int \*\*C, int n, int \*opcount) {

\*opcount = 0; // Reset operation count

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

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

C[i][j] = 0; // Initialize each element of the resulting matrix C to 0

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

C[i][j] += A[i][k] \* B[k][j]; // Perform dot product

(\*opcount)++; // Increment operation count for each multiplication and addition

}

}

}

}

// Function to print a matrix

void printMatrix(int \*\*matrix, int n) {

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

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

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

}

printf("\n");

}

}

// Function to dynamically allocate memory for a matrix

int\*\* allocateMatrix(int n) {

int \*\*matrix = (int \*\*)malloc(n \* sizeof(int \*));

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

matrix[i] = (int \*)malloc(n \* sizeof(int));

}

return matrix;

}

// Function to deallocate memory for a matrix

void deallocateMatrix(int \*\*matrix, int n) {

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

free(matrix[i]);

}

free(matrix);

}

int main() {

int n;

// Input the size of the square matrices

printf("Enter the size of the square matrices: ");

scanf("%d", &n);

// Allocate memory for matrices A, B, and C

int \*\*A = allocateMatrix(n);

int \*\*B = allocateMatrix(n);

int \*\*C = allocateMatrix(n);

// Input elements of matrix A

printf("Enter elements of matrix A:\n");

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

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

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

}

}

// Input elements of matrix B

printf("Enter elements of matrix B:\n");

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

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

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

}

}

// Perform matrix multiplication and count operations

int opcount;

clock\_t start = clock();

matrixMultiplication(A, B, C, n, &opcount);

clock\_t end = clock();

// Calculate time taken

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

// Print the resulting matrix C

printf("Resulting matrix C:\n");

printMatrix(C, n);

// Print operation count

printf("Operation count: %d\n", opcount);

// Print time taken

printf("Time taken: %f seconds\n", time\_taken);

// Deallocate memory for matrices

deallocateMatrix(A, n);

deallocateMatrix(B, n);

deallocateMatrix(C, n);

return 0;

}

**No**

**Write a program to implement solution to partition problem using recursion.**

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

int basicOpCount = 0; // Global variable to count the number of basic operations

// Function to check if there exists a subset with given sum recursively

bool isSubsetSum(int arr[], int n, int sum) {

// Base cases

if (sum == 0)

return true;

if (n == 0 && sum != 0)

return false;

// Increment the operation count

basicOpCount++;

// If last element is greater than sum, then ignore it

if (arr[n - 1] > sum)

return isSubsetSum(arr, n - 1, sum);

/\* else, check if sum can be obtained by any of the following:

(a) including the last element

(b) excluding the last element \*/

return isSubsetSum(arr, n - 1, sum) || isSubsetSum(arr, n - 1, sum - arr[n - 1]);

}

// Function to check if array can be partitioned into two subsets with equal sum

bool findPartition(int arr[], int n) {

// Calculate sum of all elements

int sum = 0;

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

sum += arr[i];

// If sum is odd, there cannot be two subsets with equal sum

if (sum % 2 != 0)

return false;

// Return true if there exists a subset with sum equal to half of total sum

return isSubsetSum(arr, n, sum / 2);

}

int main() {

int n;

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

scanf("%d", &n);

int arr[n];

printf("Enter %d elements of the array:\n", n);

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

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

}

// Measure start time

clock\_t start = clock();

// Check if the array can be partitioned into two subsets with equal sum

if (findPartition(arr, n))

printf("Array can be partitioned into two subsets with equal sum\n");

else

printf("Array cannot be partitioned into two subsets with equal sum\n");

// Measure end time

clock\_t end = clock();

// Calculate the time taken

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Number of basic operations: %d\n", basicOpCount);

printf("Time taken: %f seconds\n", time\_taken);

return 0;

}

**No**

**Write a program to sort a set of strings using Bubble Sort. Analyze its time efficiency.**

#include <stdio.h>

#include <string.h>

#include <time.h>

void bubbleSort(char \*arr[], int n, int \*opcount) {

int i, j;

char \*temp;

\*opcount = 0; // Reset operation count

for (i = 0; i < n - 1; i++) {

for (j = 0; j < n - i - 1; j++) {

(\*opcount)++; // Increment opcount for every comparison

if (strcmp(arr[j], arr[j + 1]) > 0) {

// Swap arr[j] and arr[j+1]

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

int main() {

int n;

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

scanf("%d", &n);

char \*arr[n];

printf("Enter %d strings:\n", n);

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

arr[i] = (char \*)malloc(sizeof(char) \* 100); // Assuming each string length is at most 100 characters

scanf("%s", arr[i]);

}

// Measure start time

clock\_t start = clock();

// Perform bubble sort and count comparisons

int opcount;

bubbleSort(arr, n, &opcount);

// Measure end time

clock\_t end = clock();

// Calculate the time taken

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Sorted array:\n");

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

printf("%s\n", arr[i]);

}

printf("Operation count: %d\n", opcount); // Print operation count

printf("Time taken: %f seconds\n", time\_taken);

// Free dynamically allocated memory

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

free(arr[i]);

}

return 0;

}

**KNAPSACK**

#include<stdio.h>

#include<math.h>

#include<stdlib.h>

int Knapsack(unsigned int \*w, unsigned int \*v, unsigned int n, unsigned int B) {

unsigned int i, temp;

unsigned int maxVal = 0, maxSequence = 0;

unsigned int totalWeight, totalValue;

int opcount = 0; // Initialize the opcount variable

unsigned int index = 0;

// Generate bit array

for (i = 1; i < pow(2, n); i++) {

totalWeight = totalValue = 0;

index = 0;

temp = i;

while (temp) {

if (temp & 0x1) {

totalWeight += w[index];

totalValue += v[index];

}

index++;

temp >>= 1;

}

if (totalWeight <= B && totalValue > maxVal) {

maxVal = totalValue;

maxSequence = i;

}

opcount++; // Increment opcount for every checked subset

}

printf("\nOperation count = %d\n", opcount);

return maxSequence;

}

int main() {

unsigned int \*v, \*w, i, n, knaps, B;

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

scanf("%u", &n);

v = (unsigned int \*)calloc(n, sizeof(unsigned int));

w = (unsigned int \*)calloc(n, sizeof(unsigned int));

printf("Please enter the weights: ");

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

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

printf("Please enter the values: ");

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

scanf("%u", &v[i]);

printf("Please enter the Knapsack capacity: ");

scanf("%u", &B);

knaps = Knapsack(w, v, n, B);

printf("Knapsack contains the following items:\n");

i = 0;

while (knaps) {

if (knaps & 0x1)

printf("Item %u value %u\n", (i + 1), v[i]);

i++;

knaps >>= 1;

}

free(v);

free(w);

return 0;

}

Write a program for an assignment problem by brute-force technique and analyze its time efficiency. Obtain the experimental result of an order of growth and plot the result.

#include <stdio.h>

#include <limits.h>

#include <time.h>

unsigned long long factorial(int n) {

if (n == 0 || n == 1)

return 1;

return n \* factorial(n - 1);

}

int minCost(int cost[10][10], int n, int assignment[]) {

int minCost = INT\_MAX;

unsigned long long totalPermutations = factorial(n);

int opcount = 0;

for (unsigned long long perm = 0; perm < totalPermutations; perm++) {

int currentCost = 0;

int selected[10];

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

selected[i] = 0;

}

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

int worker = i;

int job = (perm / factorial(n - 1 - i)) % (n - i); // Select job for current worker

opcount++;

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

if (!selected[j]) { // Check if job is unassigned

if (job == 0) {

selected[j] = 1; // Mark job as assigned

assignment[worker] = j; // Update assignment

currentCost += cost[worker][j];

break;

}

job--;

}

}

}

// Update minimum cost

if (currentCost < minCost) {

minCost = currentCost;

}

}

printf("Operation count: %d\n", opcount);

return minCost;

}

int main() {

int n;

printf("Enter the number of workers and jobs: ");

scanf("%d", &n);

if (n > 10) {

printf("Error: Maximum number of workers and jobs exceeded.\n");

return 1;

}

// Declare arrays for cost matrix and assignment

int cost[10][10];6

int assignment[10];

// Input the cost matrix

printf("Enter the cost matrix (%d x %d):\n", n, n);

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

printf("Enter costs for worker %d: ", i + 1);

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

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

}

}

// Measure start time

clock\_t start = clock();

// Find minimum cost using brute force

int min = minCost(cost, n, assignment);

// Measure end time

clock\_t end = clock();

// Calculate the time taken

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

// Print assignment details

printf("Assignment details:\n");

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

printf("Worker %d assigned to Job %d\n", i + 1, assignment[i] + 1);

}

printf("Minimum cost of assignment: %d\n", min);

printf("Time taken: %f seconds\n", time\_taken);

return 0;

}

**2).Write a program to implement the Traveling Salesman Problem using Brute Force Method**

#include

#include

#define N 4 // Number of cities

int minCost = INT\_MAX; // Initialize minimum cost to a very large value

int minPath[N]; // Store the minimum cost path

// Function to swap two cities

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

{

int temp = \*x;

\*x = \*y;

\*y = temp; }

// Function to calculate the total cost of the path

int calculateCost(int graph[N][N], int path[N])

{

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]]; // Add the cost to return to the starting city

return cost;

}

// Function to find the minimum cost path using brute force method

void tspBruteForce(int graph[N][N], int path[N], int start, int end)

{

if (start == end - 1)

{

int currentCost = calculateCost(graph, path);

if (currentCost < minCost)

{

minCost = currentCost;

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

{ minPath[i] = path[i];

}

}

}

else

{

for (int i = start; i < end; i++)

{

swap(&path[start], &path[i]); tspBruteForce(graph, path, start + 1, end); swap(&path[start], &path[i]); // Backtrack

}

}

}

int main()

{

int graph[N][N] = { {0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0} };

int path[N]; // Store the current path

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

path[i] = i;

tspBruteForce(graph, path, 0, N);

printf("Minimum Cost: %d\n", minCost);

printf("Path: ");

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

printf("%d -> ", minPath[i] + 1); // Adding 1 to convert index to city number printf("%d\n", minPath[0] + 1); // Print the starting city again to complete the loop

return 0;

}

**No**

**1). Write a program to check whether a graph is bipartite or not using i. DFS to check for bipartite**

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

#define MAX\_VERTICES 100

struct Graph {

int V; // Number of vertices

int edges[MAX\_VERTICES][MAX\_VERTICES];

};

void initializeGraph(struct Graph\* graph, int V) {

graph->V = V;

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

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

graph->edges[i][j] = 0;

}

}

}

void addEdge(struct Graph\* graph, int u, int v) {

graph->edges[u - 1][v - 1] = 1;

graph->edges[v - 1][u - 1] = 1;

}

bool isBipartiteUtil(struct Graph\* graph, int u, int color[]) {

for (int v = 0; v < graph->V; v++) {

if (graph->edges[u][v]) {

if (color[v] == -1) {

color[v] = 1 - color[u]; // Assign alternate color to adjacent vertex

if (!isBipartiteUtil(graph, v, color)) {

return false;

}

} else if (color[v] == color[u]) {

return false; // If adjacent vertices have the same color, the graph is not bipartite

}

}

}

return true;

}

bool isBipartite(struct Graph\* graph) {

int color[graph->V]; // Array to store colors of vertices

for (int i = 0; i < graph->V; i++) {

color[i] = -1;

}

for (int i = 0; i < graph->V; i++) {

if (color[i] == -1) {

color[i] = 0; // Assign color 0 to the starting vertex

if (!isBipartiteUtil(graph, i, color)) {

return false; // If conflict is found, graph is not bipartite

}

}

}

return true; // If no conflicts are found, the graph is bipartite

}

int main() {

int V, E;

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

scanf("%d", &V);

struct Graph graph;

initializeGraph(&graph, V);

printf("Enter the number of edges in the graph: ");

scanf("%d", &E);

printf("Enter the edges (u v format):\n");

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

int u, v;

scanf("%d %d", &u, &v);

addEdge(&graph, u, v);

}

// Measure start time

clock\_t start = clock();

// Check if the graph is bipartite

if (isBipartite(&graph)) {

printf("The graph is bipartite.\n");

} else {

printf("The graph is not bipartite.\n");

}

// Measure end time

clock\_t end = clock();

// Calculate the time taken

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken: %f seconds\n", time\_taken);

return 0;

}

***No***

***Using bfs***

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

#define MAX\_VERTICES 100

// Structure to represent a graph

struct Graph {

int V; // Number of vertices

int edges[MAX\_VERTICES][MAX\_VERTICES]; // Adjacency matrix

};

// Function to initialize a graph with given number of vertices

void initializeGraph(struct Graph\* graph, int V) {

graph->V = V;

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

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

graph->edges[i][j] = 0;

}

}

}

// Function to add an edge to the graph

void addEdge(struct Graph\* graph, int u, int v) {

graph->edges[u - 1][v - 1] = 1;

graph->edges[v - 1][u - 1] = 1;

}

// Function to check whether the graph is bipartite using BFS

bool isBipartite(struct Graph\* graph) {

int color[graph->V]; // Array to store colors of vertices

for (int i = 0; i < graph->V; i++) {

color[i] = -1; // Initialize colors as uncolored (-1)

}

// Queue for BFS traversal

int queue[MAX\_VERTICES];

int front = -1, rear = -1;

// Start BFS traversal from each vertex

for (int i = 0; i < graph->V; i++) {

if (color[i] == -1) {

color[i] = 0; // Assign color 0 to the starting vertex

front = rear = 0;

queue[rear] = i;

while (front <= rear) {

int u = queue[front++];

for (int v = 0; v < graph->V; v++) {

if (graph->edges[u][v]) {

if (color[v] == -1) {

color[v] = 1 - color[u]; // Assign alternate color to adjacent vertex

queue[++rear] = v;

} else if (color[v] == color[u]) {

return false; // If adjacent vertices have the same color, the graph is not bipartite

}

}

}

}

}

}

return true; // If no conflicts are found, the graph is bipartite

}

int main() {

int V, E;

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

scanf("%d", &V);

struct Graph graph;

initializeGraph(&graph, V);

printf("Enter the number of edges in the graph: ");

scanf("%d", &E);

printf("Enter the edges (u v format):\n");

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

int u, v;

scanf("%d %d", &u, &v);

addEdge(&graph, u, v);

}

// Measure start time

clock\_t start = clock();

// Check if the graph is bipartite

if (isBipartite(&graph)) {

printf("The graph is bipartite.\n");

} else {

printf("The graph is not bipartite.\n");

}

// Measure end time

clock\_t end = clock();

// Calculate the time taken

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken: %f seconds\n", time\_taken);

return 0;

}

**INSERTION SORT**

#include<stdio.h>

#include<stdlib.h>

// Function to perform insertion sort and count basic operations

void insertionSort(int \*a, unsigned int n, int \*opcount) {

\*opcount = 0; // Reset operation count

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

int v = a[i];

int j = i - 1;

// Increment opcount whenever there is an element comparison

while (++(\*opcount) && j >= 0 && a[j] > v) {

a[j + 1] = a[j];

j = j - 1;

}

a[j + 1] = v;

}

}

int main() {

int \*a;

int n;

printf("Enter the size of the array: ");

scanf("%d", &n);

a = (int \*)malloc(sizeof(int) \* n);

printf("Enter %d elements of the array:\n", n);

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

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

}

// Perform insertion sort and count basic operations

int opcount;

insertionSort(a, n, &opcount);

// Print sorted array

printf("\nSorted array:\n");

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

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

}

printf("\n");

// Print operation count

printf("Operation count: %d\n", opcount);

free(a);

return 0;

}

.**Write a program to determine the Topological sort of a given graph using i. Depth-First technique**

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

#define MAX\_VERTICES 100

int adjMatrix[MAX\_VERTICES][MAX\_VERTICES];

int visited[MAX\_VERTICES];

int stack[MAX\_VERTICES];

int top = -1;

void initializeGraph(int vertices) {

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

visited[i] = 0;

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

adjMatrix[i][j] = 0;

}

}

}

void addEdge(int u, int v) {

adjMatrix[u][v] = 1;

}

void dfs(int vertex, int vertices) {

visited[vertex] = 1;

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

if (adjMatrix[vertex][i] && !visited[i]) {

dfs(i, vertices);

}

}

stack[++top] = vertex;

}

void topologicalSort(int vertices) {

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

if (!visited[i]) {

dfs(i, vertices);

}

}

printf("Topological Sorting: ");

while (top >= 0) {

printf("%d ", stack[top--]);

}

printf("\n");

}

int main() {

clock\_t s;

clock\_t e;

int vertices, edges;

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

scanf("%d", &vertices);

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

scanf("%d", &edges);

initializeGraph(vertices);

printf("Enter the edges (u v):\n");

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

int u, v;

scanf("%d %d", &u, &v);

addEdge(u, v);

}

s=clock();

topologicalSort(vertices);

e=clock();

double tot\_time=(double)(e-s)/CLOCKS\_PER\_SEC;

printf("%lf",tot\_time);

return 0;

}😘

). Write a program to find diameter of a binary tree. Diameter of a binary tree is the longest path between any two nodes. For e.g. consider the following two binary trees.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// A binary tree node has data, pointer to left child

// and a pointer to right child

struct node {

int data;

struct node \*left, \*right;

};

// function to create a new node of tree and returns pointer

struct node\* newNode(int data) {

struct node\* node = (struct node\*)malloc(sizeof(struct node));

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

// Function to insert a node into a binary search tree

struct node\* insertNode(struct node\* root, int data) {

if (root == NULL) return newNode(data);

if (data < root->data)

root->left = insertNode(root->left, data);

else if (data > root->data)

root->right = insertNode(root->right, data);

return root;

}

int height(struct node\* node, int\* opcount) {

if (node == NULL)

return 0;

(\*opcount)++;

return 1 + max(height(node->left, opcount), height(node->right, opcount));

}

// Function to get diameter of a binary tree

int diameter(struct node\* tree, int\* opcount) {

if (tree == NULL)

return 0;

int lheight = height(tree->left, opcount);

int rheight = height(tree->right, opcount);

int ldiameter = diameter(tree->left, opcount);

int rdiameter = diameter(tree->right, opcount);

return max(lheight + rheight + 1, max(ldiameter, rdiameter));

}

// returns max of two integers

int max(int a, int b) { return (a > b) ? a : b; }

// Driver Code

int main() {

struct node\* root = NULL;

int val;

char ch;

printf("Enter the value of the root node: ");

scanf("%d", &val);

root = insertNode(root, val);

printf("Do you want to insert child nodes? (y/n): ");

scanf(" %c", &ch);

while (ch == 'y' || ch == 'Y') {

printf("Enter the value of the node: ");

scanf("%d", &val);

insertNode(root, val);

printf("Do you want to insert more nodes? (y/n): ");

scanf(" %c", &ch);

}

// Calculate diameter of the binary tree

clock\_t start = clock();

int opcount = 0;

int diam = diameter(root, &opcount);

clock\_t end = clock();

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

// Output diameter, operation count, and execution time

printf("Diameter of the binary tree: %d\n", diam);

printf("Operation count: %d\n", opcount);

printf("Execution time: %f seconds\n", time\_taken);

return 0;

}

ort given set of integers using Quick sort and analyze its efficiency. Obtain the experimental result of order of growth and plot the result.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int partition(int arr[], int l, int r) {

int pivot = arr[l];

int i = l;

int j = r + 1;

while (1) {

do {

i++;

} while (arr[i] <= pivot && i <= r);

do {

j--;

} while (arr[j] > pivot);

if (i >= j) {

break;

}

swap(&arr[i], &arr[j]);

}

swap(&arr[l], &arr[j]);

return j;

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

int n;

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

scanf("%d", &n);

int arr[n];

printf("Enter %d elements:\n", n);

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

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

}

printf("Unsorted array:\n");

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

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

}

printf("\n");

clock\_t start = clock();

quickSort(arr, 0, n - 1);

clock\_t end = clock();

printf("Sorted array:\n");

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

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

}

printf("\n");

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Execution time: %f seconds\n", time\_taken);

return 0;

}

Sort given set of integers using Merge sort and analyze its efficiency. Obtain the experimental result of order of growth and plot the result.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void merge(int arr[], int low, int mid, int high, int opcount) {

int n1 = mid - low + 1;

int n2 = high - mid;

int L[n1], R[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[low + i];

for (int j = 0; j < n2; j++)

R[j] = arr[mid + 1 + j];

int i = 0, j = 0, k = low;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

opcount += 2;

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

opcount += 2;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

opcount += 2;

}

}

void mergeSort(int arr[], int low, int high, int opcount) {

if (low < high) {

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

mergeSort(arr, low, mid, opcount);

mergeSort(arr, mid + 1, high, opcount);

merge(arr, low, mid, high, opcount);

}

}

int main() {

int n;

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

scanf("%d", &n);

int arr[n];

printf("Enter %d integers:\n", n);

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

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

}

int opcount = 0;

clock\_t start = clock();

mergeSort(arr, 0, n - 1, opcount);

clock\_t end = clock();

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Sorted array:\n");

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

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

}

printf("\n");

printf("Operation count: %d\n", opcount);

printf("Execution time: %f seconds\n", time\_taken);

return 0;

}

**Write a program to determine the height of a binary search tree and analyze its time efficiency.**

#include<stdio.h>

#include<stdlib.h>

#define MAX(a,b) ((a) > (b) ? a : b)

int opcount = 0; // initialize the opcount variable

struct node {

int val;

struct node \*left, \*right;

};

typedef struct node \*NODE;

NODE root = NULL;

NODE insert(int ele, NODE node) {

NODE temp;

if(node == NULL) {

temp = (NODE)malloc(sizeof(struct node));

temp->val = ele;

temp->left = temp->right = NULL;

if(root == NULL)

root = temp;

return temp;

}

else if(ele < node->val) {

node->left = insert(ele, node->left);

return node;

}

else if(ele > node->val) {

node->right = insert(ele, node->right);

return node;

}

else {

printf("Duplicate element found while inserting. Insertion failed\n");

return NULL;

}

}

int height(NODE node) {

opcount++; // increment opcount for the comparison statement

if(node == NULL)

return -1;

else

return MAX(height(node->left), height(node->right)) + 1;

}

int main() {

int choice, ele;

do {

printf("1. Insert an element\n");

printf("2. Find Height of BST\n");

printf("3. Exit\n");

printf("Please enter your choice: ");

scanf("%d", &choice);

switch(choice) {

case 1:

printf("Insertion : Please enter an element: ");

scanf("%d", &ele);

insert(ele, root);

break;

case 2:

printf("Height of BST: %d\n", height(root));

printf("Opcount=%d\n", opcount);

break;

case 3:

break;

default:

printf("Invalid choice. Please enter valid choice\n");

break;

}

} while(choice != 3);

return 0;

}

**No**

**Find total number of nodes in a binary tree and analyze its efficiency. Obtain the experimental result of order of growth and plot the result.**

#include <stdio.h>

#include <stdlib.h>

struct node {

int val;

struct node \*left, \*right;

};

typedef struct node \*NODE;

NODE root = NULL;

int opcount = 0;

NODE insert(int ele, NODE node) {

NODE temp;

if (node == NULL) {

temp = (NODE)malloc(sizeof(struct node));

temp->val = ele;

temp->left = temp->right = NULL;

if (root == NULL)

root = temp;

return temp;

} else if (ele < node->val) {

node->left = insert(ele, node->left);

return node;

} else if (ele > node->val) {

node->right = insert(ele, node->right);

return node;

} else {

printf("Duplicate element found while inserting. Insertion failed\n");

return NULL;

}

}

int countNodes(NODE node) {

if (node == NULL)

return 0;

else

return 1 + countNodes(node->left) + countNodes(node->right);

}

int height(NODE node) {

opcount++; // increment opcount for the comparison statement

if (node == NULL)

return -1;

else

return MAX(height(node->left), height(node->right)) + 1;

}

int main() {

int choice, ele;

do {

printf("1. Insert an element\n");

printf("2. Find Height of BST\n");

printf("3. Count Total Nodes in BST\n");

printf("4. Exit\n");

printf("Please enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Insertion: Please enter an element: ");

scanf("%d", &ele);

insert(ele, root);

break;

case 2:

printf("Height of BST: %d\n", height(root));

printf("Opcount=%d\n", opcount);

break;

case 3:

printf("Total number of nodes in BST: %d\n", countNodes(root));

break;

case 4:

break;

default:

printf("Invalid choice. Please enter valid choice\n");

break;

}

} while (choice != 4);

return 0;

}

**No**

**Given an image, find the defective region in the image using divide and conquer technique. The defective region is denoted by 0 and the non-defective region is denoted by 1**

#include <stdio.h>

#define N 8 // Size of the image

// Function to check if a sub-image contains a defective region

int isDefective(int image[N][N], int rowStart, int rowEnd, int colStart, int colEnd) {

for (int i = rowStart; i <= rowEnd; i++) {

for (int j = colStart; j <= colEnd; j++) {

if (image[i][j] == 0) {

return 1; // Defective region found

}

}

}

return 0; // No defective region found

}

// Function to find defective regions in the image using divide and conquer

void findDefectiveRegions(int image[N][N], int rowStart, int rowEnd, int colStart, int colEnd) {

if (rowStart == rowEnd && colStart == colEnd) {

if (image[rowStart][colStart] == 0) {

printf("Defective region found at (%d, %d)\n", rowStart, colStart);

}

return;

}

int midRow = (rowStart + rowEnd) / 2;

int midCol = (colStart + colEnd) / 2;

if (isDefective(image, rowStart, midRow, colStart, midCol)) {

findDefectiveRegions(image, rowStart, midRow, colStart, midCol);

}

if (isDefective(image, rowStart, midRow, midCol + 1, colEnd)) {

findDefectiveRegions(image, rowStart, midRow, midCol + 1, colEnd);

}

if (isDefective(image, midRow + 1, rowEnd, colStart, midCol)) {

findDefectiveRegions(image, midRow + 1, rowEnd, colStart, midCol);

}

if (isDefective(image, midRow + 1, rowEnd, midCol + 1, colEnd)) {

findDefectiveRegions(image, midRow + 1, rowEnd, midCol + 1, colEnd);

}

}

int main() {

int image[N][N] = {

{1, 1, 1, 1, 1, 1, 1, 1},

{1, 1, 1, 1, 1, 1, 1, 1},

{1, 1, 1, 1, 0, 1, 1, 1},

{1, 1, 1, 1, 0, 1, 1, 1},

{1, 1, 1, 1, 1, 1, 1, 1},

{1, 1, 1, 1, 1, 1, 1, 1},

{1, 1, 1, 1, 1, 1, 1, 1},

{1, 1, 1, 1, 1, 1, 1, 1}

};

printf("Defective regions in the image:\n");

findDefectiveRegions(image, 0, N - 1, 0, N - 1);

return 0;

}

**No**

**Write a program in C to find an where n > 0 using divide and conquer strategy.**

#include <stdio.h>

long long int power(int a, int n) {

if (n == 0)

return 1;

long long int temp = power(a, n / 2);

if (n % 2 == 0)

return temp \* temp;

else

return a \* temp \* temp;

}

int main() {

int a, n;

// Input the base (a) and exponent (n)

printf("Enter the base (a): ");

scanf("%d", &a);

printf("Enter the exponent (n): ");

scanf("%d", &n);

// Compute and print the result

long long int result = power(a, n);

printf("%d raised to the power %d is: %lld\n", a, n, result);

return 0;

}

No

Write a program in C to find gcd of two numbers using Euclid’s algorithm employing the Decrease and Conquer strategy.

#include <stdio.h>

// Function to compute the GCD using Euclid's algorithm

int gcd(int m, int n) {

// Base case: if n is 0, return m

if (n == 0)

return m;

// Reduce the problem size by recursively calling gcd with n and the remainder of m divided by n

return gcd(n, m % n);

}

int main() {

int m, n;

// Input the two numbers

printf("Enter the first number (m): ");

scanf("%d", &m);

printf("Enter the second number (n): ");

scanf("%d", &n);

// Compute the GCD using Euclid's algorithm

int result = gcd(m, n);

// Output the result

printf("The GCD of %d and %d is: %d\n", m, n, result);

return 0;

}

No

Write a program for depth-first search of a graph. Identify the push and pop order of vertices.

#include <stdio.h>

#include <stdbool.h>

#define MAX\_VERTICES 100

// Function to perform depth-first search (DFS) of the graph

void DFS(int start, int graph[MAX\_VERTICES][MAX\_VERTICES], bool visited[MAX\_VERTICES], int numVertices) {

int stack[MAX\_VERTICES];

int top = -1;

stack[++top] = start;

visited[start] = true;

printf("Push: %d\n", start);

while (top != -1) {

int vertex = stack[top--];

// Process the current vertex

printf("Pop: %d\n", vertex);

// Visit adjacent vertices

for (int neighbor = 0; neighbor < numVertices; ++neighbor) {

if (graph[vertex][neighbor] && !visited[neighbor]) {

stack[++top] = neighbor;

visited[neighbor] = true;

printf("Push: %d\n", neighbor);

}

}

}

}

int main() {

int numVertices, numEdges;

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

scanf("%d", &numVertices);

printf("Enter the number of edges in the graph: ");

scanf("%d", &numEdges);

// Create the adjacency matrix for the graph

int graph[MAX\_VERTICES][MAX\_VERTICES] = {0};

printf("Enter the edges (format: source destination):\n");

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

int source, destination;

scanf("%d %d", &source, &destination);

graph[source][destination] = 1;

}

// Visited array to keep track of visited vertices

bool visited[MAX\_VERTICES] = {false};

// Start DFS from vertex 0

printf("Depth-First Search:\n");

DFS(0, graph, visited, numVertices);

return 0;

}

**No**

**Binary search**

#include <stdio.h>

// Binary Search function

int binarySearch(int arr[], int low, int high, int key) {

while (low <= high) {

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

// Check if key is present at mid

if (arr[mid] == key)

return mid;

// If key greater, ignore left half

if (arr[mid] < key)

low = mid + 1;

// If key is smaller, ignore right half

else

high = mid - 1;

}

// If key is not present in array

return -1;

}

int main() {

int arr[] = {2, 3, 4, 10, 40};

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

int key = 10;

int result = binarySearch(arr, 0, n - 1, key);

(result == -1) ? printf("Element is not present in array")

: printf("Element is present at index %d", result);

return 0;

}

No

MAZE

#include <stdio.h>

#include <stdbool.h>

#define ROWS 5

#define COLS 5

// Structure to represent a cell in the maze

struct Cell {

int row;

int col;

};

// Function to check if a cell is valid and can be visited

bool isValidCell(int row, int col, bool visited[ROWS][COLS], char maze[ROWS][COLS]) {

return (row >= 0 && row < ROWS && col >= 0 && col < COLS && maze[row][col] != '#' && !visited[row][col]);

}

// Function to perform depth-first search (DFS)

bool DFS(char maze[ROWS][COLS], bool visited[ROWS][COLS], struct Cell current, struct Cell destination) {

if (current.row == destination.row && current.col == destination.col) {

return true; // Destination reached

}

// Mark the current cell as visited

visited[current.row][current.col] = true;

// Define the possible movements: up, down, left, right

int dr[] = {-1, 1, 0, 0};

int dc[] = {0, 0, -1, 1};

// Try all possible movements

for (int i = 0; i < 4; i++) {

int newRow = current.row + dr[i];

int newCol = current.col + dc[i];

if (isValidCell(newRow, newCol, visited, maze)) {

if (DFS(maze, visited, (struct Cell){newRow, newCol}, destination)) {

return true; // Destination found in this direction

}

}

}

return false; // Destination not found from this cell

}

int main() {

char maze[ROWS][COLS] = {

{'S', '#', '#', '.', '.'},

{'.', '.', '#', '.', '#'},

{'.', '#', '.', '.', '#'},

{'.', '.', '.', '#', '.'},

{'#', '#', '.', 'D', '.'}

};

bool visited[ROWS][COLS] = {false};

// Starting and destination cells

struct Cell start = {0, 0};

struct Cell destination = {ROWS - 1, COLS - 1};

// Find if there exists a path from start to destination using DFS

if (DFS(maze, visited, start, destination)) {

printf("There exists a path from start to destination.\n");

} else {

printf("There is no path from start to destination.\n");

}

return 0;

}

TOPOLO

#include <stdio.h>

#include <stdlib.h>

// Structure for a node of the binary tree

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 to insert a node into the binary tree

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

root->left = insert(root->left, data);

} else if (data > root->data) {

root->right = insert(root->right, data);

}

return root;

}

// Function for inorder traversal of the binary tree

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

// Function for preorder traversal of the binary tree

void preorder(struct Node\* root) {

if (root != NULL) {

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

preorder(root->left);

preorder(root->right);

}

}

// Function for postorder traversal of the binary tree

void postorder(struct Node\* root) {

if (root != NULL) {

postorder(root->left);

postorder(root->right);

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

}

}

int main() {

struct Node\* root = NULL;

int numNodes, data;

printf("Enter the number of nodes in the binary tree: ");

scanf("%d", &numNodes);

printf("Enter the data for each node:\n");

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

scanf("%d", &data);

root = insert(root, data);

}

printf("Inorder traversal: ");

inorder(root);

printf("\n");

printf("Preorder traversal: ");

preorder(root);

printf("\n");

printf("Postorder traversal: ");

postorder(root);

printf("\n");

return 0;

}

start

)Write a program to create a binary search tree and display its elements using all the traversal methods and analyse its time efficiency.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Structure for a node of the binary search tree

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Function to create a new node

struct Node\* create\_node(int data) {

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

new\_node->data = data;

new\_node->left = new\_node->right = NULL;

return new\_node;

}

// Function to insert a new node with given key in BST

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL)

return create\_node(data);

if (data < root->data)

root->left = insert(root->left, data);

else if (data > root->data)

root->right = insert(root->right, data);

return root;

}

// Function for in-order traversal (left-root-right)

void in\_order\_traversal(struct Node\* root) {

if (root != NULL) {

in\_order\_traversal(root->left);

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

in\_order\_traversal(root->right);

}

}

// Function for pre-order traversal (root-left-right)

void pre\_order\_traversal(struct Node\* root) {

if (root != NULL) {

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

pre\_order\_traversal(root->left);

pre\_order\_traversal(root->right);

}

}

// Function for post-order traversal (left-right-root)

void post\_order\_traversal(struct Node\* root) {

if (root != NULL) {

post\_order\_traversal(root->left);

post\_order\_traversal(root->right);

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

}

}

// Function to deallocate memory of the tree

void free\_tree(struct Node\* root) {

if (root != NULL) {

free\_tree(root->left);

free\_tree(root->right);

free(root);

}

}

int main() {

struct Node\* root = NULL;

int data;

int n;

// Taking input for the number of elements

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

scanf("%d", &n);

// Taking input for the elements and inserting into the BST

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

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

scanf("%d", &data);

root = insert(root, data);

}

// Measure time taken for traversals

clock\_t start, end;

double cpu\_time\_used;

// In-order traversal

start = clock();

printf("\nIn-order traversal: ");

in\_order\_traversal(root);

end = clock();

cpu\_time\_used = ((double) (end - start)) / CLOCKS\_PER\_SEC;

printf("\nTime taken for in-order traversal: %lf seconds\n", cpu\_time\_used);

// Pre-order traversal

start = clock();

printf("\nPre-order traversal: ");

pre\_order\_traversal(root);

end = clock();

cpu\_time\_used = ((double) (end - start)) / CLOCKS\_PER\_SEC;

printf("\nTime taken for pre-order traversal: %lf seconds\n", cpu\_time\_used);

// Post-order traversal

start = clock();

printf("\nPost-order traversal: ");

post\_order\_traversal(root);

end = clock();

cpu\_time\_used = ((double) (end - start)) / CLOCKS\_PER\_SEC;

printf("\nTime taken for post-order traversal: %lf seconds\n", cpu\_time\_used);

// Deallocate memory

free\_tree(root);

return 0;

}

Modify the solved exercise to find the balance factor for every node in the binary search tree.

#include <stdio.h>

#include <stdlib.h>

// Definition of a node in the binary search tree

struct TreeNode {

int val;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to create a new node

struct TreeNode\* createNode(int key) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->val = key;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to insert a key into BST

struct TreeNode\* insert(struct TreeNode\* root, int key) {

if (root == NULL) {

return createNode(key);

}

if (key < root->val) {

root->left = insert(root->left, key);

} else if (key > root->val) {

root->right = insert(root->right, key);

}

return root;

}

// Function to calculate the height of a tree

int height(struct TreeNode\* node) {

if (node == NULL) {

return 0;

}

int leftHeight = height(node->left);

int rightHeight = height(node->right);

return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

}

// Function to find the balance factor of a node

int balanceFactor(struct TreeNode\* node) {

if (node == NULL) {

return 0;

}

int leftHeight = height(node->left);

int rightHeight = height(node->right);

return leftHeight - rightHeight;

}

// Function for inorder traversal

void inorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

inorderTraversal(root->left);

printf("%d (Balance Factor: %d) ", root->val, balanceFactor(root));

inorderTraversal(root->right);

}

}

// Main function

int main() {

// Create a binary search tree

struct TreeNode\* root = NULL;

int elements[7];

printf("Enter 7 elements to construct the binary search tree:\n");

for (int i = 0; i < 7; i++) {

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

}

for (int i = 0; i < 7; ++i) {

root = insert(root, elements[i]);

}

// Display elements with balance factors using inorder traversal

printf("Inorder traversal with balance factors:\n");

inorderTraversal(root);

printf("\n");

return 0;

}

Create AVL tree using iterative method

#include <stdio.h>

#include <stdlib.h>

// Definition of a node in the AVL tree

struct AVLNode {

int key;

struct AVLNode \*left;

struct AVLNode \*right;

int height;

};

// Function to create a new node

struct AVLNode\* createNode(int key) {

struct AVLNode\* newNode = (struct AVLNode\*)malloc(sizeof(struct AVLNode));

newNode->key = key;

newNode->left = NULL;

newNode->right = NULL;

newNode->height = 1;

return newNode;

}

// Function to get the height of a node

int getHeight(struct AVLNode\* node) {

if (node == NULL)

return 0;

return node->height;

}

// Function to get the maximum of two integers

int max(int a, int b) {

return (a > b) ? a : b;

}

// Function to rotate the subtree rooted with y to the right

struct AVLNode\* rightRotate(struct AVLNode\* y) {

struct AVLNode\* x = y->left;

struct AVLNode\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

// Return new root

return x;

}

// Function to rotate the subtree rooted with x to the left

struct AVLNode\* leftRotate(struct AVLNode\* x) {

struct AVLNode\* y = x->right;

struct AVLNode\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

// Return new root

return y;

}

// Get the balance factor of a node

int getBalance(struct AVLNode\* node) {

if (node == NULL)

return 0;

return getHeight(node->left) - getHeight(node->right);

}

// Function to insert a key into AVL tree

struct AVLNode\* insert(struct AVLNode\* root, int key) {

if (root == NULL)

return createNode(key);

if (key < root->key)

root->left = insert(root->left, key);

else if (key > root->key)

root->right = insert(root->right, key);

else // Equal keys are not allowed in AVL tree

return root;

// Update height of this ancestor node

root->height = 1 + max(getHeight(root->left), getHeight(root->right));

// Get the balance factor to check whether this node became unbalanced

int balance = getBalance(root);

// If this node becomes unbalanced, there are four cases

// Left Left Case

if (balance > 1 && key < root->left->key)

return rightRotate(root);

// Right Right Case

if (balance < -1 && key > root->right->key)

return leftRotate(root);

// Left Right Case

if (balance > 1 && key > root->left->key) {

root->left = leftRotate(root->left);

return rightRotate(root);

}

// Right Left Case

if (balance < -1 && key < root->right->key) {

root->right = rightRotate(root->right);

return leftRotate(root);

}

// return the (unchanged) node pointer

return root;

}

// Function for inorder traversal

void inorderTraversal(struct AVLNode\* root) {

if (root != NULL) {

inorderTraversal(root->left);

printf("%d ", root->key);

inorderTraversal(root->right);

}

}

// Main function

int main() {

// Create an empty AVL tree

struct AVLNode\* root = NULL;

// Input elements into AVL tree

int numElements;

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

scanf("%d", &numElements);

printf("Enter %d elements to insert into AVL tree:\n", numElements);

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

int element;

scanf("%d", &element);

root = insert(root, element);

}

// Display elements in sorted order (inorder traversal)

printf("Inorder traversal of AVL tree: ");

inorderTraversal(root);

printf("\n");

return 0;

}

Write a program to create a heap for the list of integers using top-down heap construction algorithm and analyze its time efficiency. Obtain the experimental results for order of growth and plot the result.

#include <stdio.h>

#include <stdlib.h>

int op = 0;

void topDown(int arr[], int currIndex)

{

int parent = (currIndex - 1)/2;

while(parent >= 0)

{

op++;

if(arr[parent]<arr[currIndex])

{

int temp = arr[parent];

arr[parent] = arr[currIndex];

arr[currIndex] = temp;

currIndex = parent;

parent = (currIndex - 1)/2;

}

else

return;

}

}

int main()

{

int h[20], n;

printf("Enter no. of elements:\n");

scanf("%d", &n);

printf("Enter Elements:\n");

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

{

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

topDown(h, i);

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

printf("%d ", h[k]);

printf("\n");

}

printf("Heapified array:\n");

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

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

printf("\n");

printf("OP = %d\n", op);

exit(0);

return 0;

}

Write a program to sort the list of integers using heap sort with bottom up max heap construction and analyze its time efficiency. Prove experimentally that the worst case time complexity is O(n log n)

#include <stdio.h>

#include <stdlib.h>

int op = 0;

void heapify(int h[], int l, int n)

{

int i, k, v, heapify, j;

for(i = (n/2); i>=l; i--)

{

k = i; v = h[k]; heapify = 0;

while(heapify == 0 && 2\*k <= n)

{

j = 2\*k;

op++;

if(j<n)

if(h[j]<h[j+1])

j = j+1;

if(v>=h[j])

heapify = 1;

else

{

h[k] = h[j];

k = j;

}

}

h[k] = v;

}

return;

}

void HeapSort(int arr[], int n)

{

int k = 0;

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

{

heapify(arr, 1, n - k);

int temp = arr[1];

arr[1] = arr[n-k];

arr[n-k] = temp;

k++;

}

}

int main()

{

int arr[20], n;

printf("Enter the Number of Elements : \n");

scanf("%d", &n);

printf("Enter the Elements : \n");

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

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

HeapSort(arr, n);

printf("The Sorted List is : \n");

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

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

printf("\n");

printf("Count = %d\n", op);

return 0;

}

: 1) Write a program to sort set of integers using comparison counting algorithm.

#include <stdio.h>

void counting\_sort(int A[], int n) {

int i, j;

int S[15], C[100];

// Initialize count array to 0

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

C[i] = 0;

// Count the occurrences of each element

for (i = 0; i < n - 1; i++) {

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

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

C[j]++;

else

C[i]++;

}

}

// Place elements in sorted order based on counts

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

S[C[i]] = A[i];

// Print the sorted array

printf("The Sorted array is : ");

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

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

}

int main() {

int n, A[15], i;

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

scanf("%d", &n);

printf("\nEnter the integers to be sorted :\n");

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

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

counting\_sort(A, n);

printf("\n");

return 0;

}

Write a program to implement Horspool’s algorithm for String Matching and find the number of key comparisons in successful search and unsuccessful search.

#include <stdio.h>

#include <string.h>

#define MAX\_CHAR 256

// Function to preprocess the pattern and generate the shift table

void preProcessShiftTable(char pattern[], int patternLength, int shiftTable[]) {

for (int i = 0; i < MAX\_CHAR; i++) {

shiftTable[i] = patternLength;

}

for (int i = 0; i < patternLength - 1; i++) {

shiftTable[pattern[i]] = patternLength - 1 - i;

}

}

// Function to perform string matching using Horspool's algorithm

int horspoolSearch(char text[], char pattern[], int \*keyComparisons) {

int textLength = strlen(text);

int patternLength = strlen(pattern);

int shiftTable[MAX\_CHAR];

preProcessShiftTable(pattern, patternLength, shiftTable);

int i = patternLength - 1;

while (i < textLength) {

int k = 0;

while (k < patternLength && pattern[patternLength - 1 - k] == text[i - k]) {

k++;

(\*keyComparisons)++;

}

if (k == patternLength) {

return i - patternLength + 1; // Match found

} else {

i += shiftTable[text[i]]; // Shift the pattern according to the shift table

(\*keyComparisons)++;

}

}

return -1; // Match not found

}

int main() {

char text[100], pattern[100];

int keyComparisons = 0;

printf("Enter the text: ");

fgets(text, sizeof(text), stdin);

printf("Enter the pattern to search for: ");

fgets(pattern, sizeof(pattern), stdin);

// Remove newline characters from inputs

text[strcspn(text, "\n")] = '\0';

pattern[strcspn(pattern, "\n")] = '\0';

int successfulMatchIndex = horspoolSearch(text, pattern, &keyComparisons);

if (successfulMatchIndex != -1) {

printf("Successful search. Match found at index: %d\n", successfulMatchIndex);

} else {

printf("Unsuccessful search. No match found.\n");

}

printf("Number of key comparisons: %d\n", keyComparisons);

return 0;

}

) Write a program to construct the Open hash table. Find the number of key comparisons in successful search and unsuccessful search. This should be done by varying the load factor of the hash table. You may consider varying the number of keys for a fixed value of hash table size say m=10 and n=50, 100, and 200. This should be repeated for at least four different hash table sizes say m= 20, m=50. #include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 200

typedef struct Node {

int key;

struct Node \*next;

} Node;

typedef struct {

Node \*table[MAX\_SIZE];

int size;

} HashTable;

int hash(int key, int size) {

return key % size;

}

///

void insert(HashTable \*ht, int key) {

int index = hash(key, ht->size);

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

newNode->key = key;

newNode->next = NULL;

if (ht->table[index] == NULL) {

ht->table[index] = newNode;

} else {

Node \*temp = ht->table[index];

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

}

bool search(HashTable \*ht, int key, int \*comparisons) {

int index = hash(key, ht->size);

Node \*temp = ht->table[index];

\*comparisons = 0;

while (temp != NULL) {

(\*comparisons)++;

if (temp->key == key) {

return true;

}

temp = temp->next;

}

return false;

}

int main() {

HashTable ht;

int key, comparisons;

// Initialize hash table size

printf("Enter hash table size: ");

scanf("%d", &ht.size);

// Initialize hash table

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

ht.table[i] = NULL;

}

// Insert keys

printf("Enter keys to insert (-1 to stop):\n");

while (true) {

scanf("%d", &key);

if (key == -1) break;

insert(&ht, key);

}

// Search for keys

printf("Enter key to search (-1 to stop):\n");

while (true) {

scanf("%d", &key);

if (key == -1) break;

if (search(&ht, key, &comparisons)) {

printf("Key %d found with %d comparisons.\n", key, comparisons);

} else {

printf("Key %d not found.\n", key);

}

}

return 0;

}

Write a program to find the Binomial Co-efficient using Dynamic Programming.

#include<stdio.h>

#include<stdlib.h>

#define min(a, b) ((a) < (b) ? (a) : (b))

int c[20][20];

void binomial(int n, int k) {

int i, j;

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

for (j = 0; j <= min(i, k); j++) {

if (j == 0 || j == i)

c[i][j] = 1;

else

c[i][j] = c[i - 1][j - 1] + c[i - 1][j];

}

}

}

int main() {

int n, k, i, j;

printf("Enter the value of n: ");

scanf("%d", &n);

printf("Enter the value of k: ");

scanf("%d", &k);

if (n < k)

printf("Invalid input: n cannot be less than k\n");

else if (k < 0)

printf("Invalid input: k cannot be less than 0\n");

else {

binomial(n, k);

printf("Computed matrix is:\n");

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

for (j = 0; j <= min(i, k); j++)

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

printf("\n");

}

printf("Binomial coefficient c[%d,%d] = %d\n", n, k, c[n][k]);

}

return 0;

}

Write a program to compute the transitive closure of a given directed graph using Warshall’s algorithm and analyse its time efficiency. Obtain the experimental results for order of growth and plot the result.

#include <stdio.h>

#include <stdbool.h>

#define MAX\_VERTICES 100

int graph[MAX\_VERTICES][MAX\_VERTICES];

int transitiveClosure[MAX\_VERTICES][MAX\_VERTICES];

int max(int a, int b) {

return (a > b) ? a : b;

}

void warshall(int vertices) {

int i, j, k;

// Initialize transitive closure matrix with the graph matrix

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

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

transitiveClosure[i][j] = graph[i][j];

}

}

// Warshall's algorithm

for (k = 0; k < vertices; k++) {

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

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

transitiveClosure[i][j] = max(transitiveClosure[i][j], transitiveClosure[i][k] && transitiveClosure[k][j]);

}

}

}

}

void displayTransitiveClosure(int vertices) {

printf("Transitive Closure:\n");

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

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

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

}

printf("\n");

}

}

int main() {

int vertices, i, j;

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

scanf("%d", &vertices);

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

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

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

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

}

}

warshall(vertices);

displayTransitiveClosure(vertices);

return 0;

}

Write a program to implement 0/1 Knapsack problem using bottom-up dynamic programming

#include<stdio.h>

#include<time.h>

// Function to find maximum of two integers

int max(int a, int b) {

return (a > b) ? a : b;

}

// Function implementing 0/1 Knapsack problem using bottom-up dynamic programming

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

int i, w;

int K[n + 1][W + 1];

// Build K[][] in bottom-up manner

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

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

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

K[i][w] = 0;

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

K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);

else

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

}

}

return K[n][W];

}

int main() {

int n, W;

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

scanf("%d", &n);

int val[n], wt[n];

printf("Enter the values of items:\n");

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

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

}

printf("Enter the weights of items:\n");

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

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

}

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

scanf("%d", &W);

clock\_t start\_time = clock();

int result = knapSack(W, wt, val, n);

clock\_t end\_time = clock();

double time\_taken = ((double)(end\_time - start\_time)) / CLOCKS\_PER\_SEC;

printf("Maximum value that can be obtained is %d\n", result);

printf("Time taken: %f seconds\n", time\_taken);

return 0;

}

Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Prim’s algorithm.

#include<stdio.h>

#include<time.h>

int a[50][50], t[50][50], root[50], parent[50], n, i, j, value, e = 0, k = 0;

int ivalue, jvalue, cost = 0, mincost = 0, TV[50], count = 0, present = 0;

void read\_cost() {

printf("\nEnter the number of vertices: ");

scanf("%d", &n);

printf("\nEnter cost adjacency matrix:\n");

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

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

printf("(%d,%d): ", i, j);

scanf("%d", &value);

a[i][j] = value;

if (value != 0)

e++;

a[j][i] = value; // Since the graph is undirected

}

}

}

int check\_reach(int vertex) {

int v, queue[50], front = 0, rear = -1, reach[50], visited[50];

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

visited[v] = 0;

reach[v] = 0;

}

queue[++rear] = vertex;

visited[vertex] = 1;

while (front <= rear) {

v = queue[front++];

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

if (a[v][i] != 0 && !visited[i]) {

reach[i] = 1;

queue[++rear] = i;

visited[i] = 1;

}

}

}

return reach[n];

}

void prims() {

while (e && k < n - 1) {

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

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

if (a[i][j] != 0) {

int x = check\_reach(i);

int y = check\_reach(j);

if (x == 1 && y == 0) {

present = 1;

if (a[i][j] < cost || cost == 0) {

cost = a[i][j];

ivalue = i;

jvalue = j;

}

}

}

}

}

if (present) {

TV[++count] = jvalue;

parent[jvalue] = ivalue;

cost += a[ivalue][jvalue];

k++;

e--;

present = 0;

a[ivalue][jvalue] = a[jvalue][ivalue] = 0;

}

}

}

void display() {

printf("\nMinimum cost spanning tree is:\n");

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

printf("%d <-> %d\n", parent[i], i);

}

printf("\nCost of minimum cost spanning tree = %d\n", cost);

}

int main() {

printf("\n\t\t\tPRIMS ALGORITHM\n");

TV[++count] = 1;

read\_cost();

clock\_t start\_time = clock();

prims();

clock\_t end\_time = clock();

display();

double time\_taken = ((double)(end\_time - start\_time)) / CLOCKS\_PER\_SEC;

printf("\nTime taken by algorithm: %f seconds\n", time\_taken);

] return 0;

}

Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm and analyse its time efficiency

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Structure to represent an edge in the graph

struct Edge {

int src, dest, weight;

};

// Structure to represent a subset for union-find

struct Subset {

int parent;

int rank;

};

// Function prototypes

int find(struct Subset subsets[], int i);

void Union(struct Subset subsets[], int x, int y);

int compare(const void\* a, const void\* b);

void KruskalMST(struct Edge\* edges, int V, int E);

void printMST(struct Edge\* result, int e);

// Find set of an element i

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

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

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

return subsets[i].parent;

}

// Perform union of two sets

void Union(struct 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++;

}

}

// Compare function used by qsort to sort edges based on their weights

int compare(const void\* a, const void\* b) {

struct Edge\* edge1 = (struct Edge\*)a;

struct Edge\* edge2 = (struct Edge\*)b;

return edge1->weight - edge2->weight;

}

// Function to find Minimum Spanning Tree using Kruskal's algorithm

void KruskalMST(struct Edge\* edges, int V, int E) {

struct Edge result[V];

int e = 0;

int i = 0;

// Sort all the edges in non-decreasing order of their weight

qsort(edges, E, sizeof(edges[0]), compare);

// Allocate memory for creating V subsets

struct Subset\* subsets = (struct Subset\*)malloc(V \* sizeof(struct Subset));

// Create V subsets with single elements

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

subsets[v].parent = v;

subsets[v].rank = 0;

}

// Keep adding edges until V-1 edges are added or there are no more edges

while (e < V - 1 && i < E) {

// Pick the smallest edge

struct Edge next\_edge = edges[i++];

int x = find(subsets, next\_edge.src);

int y = find(subsets, next\_edge.dest);

// If including this edge doesn't cause a cycle, add it to the result

if (x != y) {

result[e++] = next\_edge;

Union(subsets, x, y);

}

}

// Print the constructed MST

printMST(result, e);

free(subsets);

}

// Print the MST

void printMST(struct Edge\* result, int e) {

printf("Following are the edges in the constructed MST:\n");

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

printf("%d -- %d == %d\n", result[i].src + 1, result[i].dest + 1, result[i].weight);

}

int main() {

int V, E;

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

scanf("%d %d", &V, &E);

// Array of edges with their source, destination, and weight

struct Edge\* edges = (struct Edge\*)malloc(E \* sizeof(struct Edge));

printf("Enter the details of each edge (source, destination, weight):\n");

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

scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);

// Adjust indices to match array indexing (starting from 0)

edges[i].src--;

edges[i].dest--;

}

clock\_t start\_time = clock(); // Measure start time

KruskalMST(edges, V, E);

clock\_t end\_time = clock(); // Measure end time

double execution\_time = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC; // Calculate execution time

printf("Execution time: %.6f seconds\n", execution\_time);

free(edges);

return 0;

}

Write a program to find shortest path from a given vertex to other vertices in a given weighted connected graph, Using Dijkstra's algorithm and analyse its time efficiency.

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define V 6 // Number of vertices in the graph

// Function prototypes

void dijkstra(int graph[V][V], int src);

int minDistance(int dist[], int visited[]);

void dijkstra(int graph[V][V], int src) {

int dist[V]; // Array to store the shortest distance from src to i

int visited[V]; // Array to keep track of visited vertices

// Initialize distances as INFINITE and visited array as 0

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

dist[i] = INT\_MAX;

visited[i] = 0;

}

// Distance of source vertex from itself is always 0

dist[src] = 0;

// Find shortest path for all vertices

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

// Pick the minimum distance vertex from the set of vertices not yet processed

int u = minDistance(dist, visited);

// Mark the picked vertex as visited

visited[u] = 1;

// Update dist value of the adjacent vertices of the picked vertex

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

// Update dist[v] only if it's not in visited[], there is an edge from u to v, and total weight of path from src to v through u is smaller than current value of dist[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];

}

}

// Print the shortest distances

printf("Shortest distances from vertex %d to all other vertices:\n", src);

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

printf("Vertex %d -> Distance = %d\n", i, dist[i]);

}

}

// Utility function to find the vertex with minimum distance value from the set of vertices not yet included in shortest path tree

int minDistance(int dist[], int visited[]) {

int min = INT\_MAX, min\_index;

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

if (visited[v] == 0 && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

int main() {

int graph[V][V]; // Weighted graph representation

int source\_vertex; // Source vertex for shortest path calculation

// Read the weighted graph from user input

printf("Enter the weighted graph (%d x %d matrix):\n", V, V);

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

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

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

}

}

// Read the source vertex from user input

printf("Enter the source vertex (0 to %d): ", V - 1);

scanf("%d", &source\_vertex);

// Find and print shortest path from source vertex to all other vertices using Dijkstra's algorithm

dijkstra(graph, source\_vertex);

return 0;

}

4) Write a program to sort the elements using distribution counting method.

#include <stdio.h>

#include <stdlib.h>

// Function to perform distribution counting sort

void distributionCountingSort(int arr[], int n) {

// Find the maximum element in the array

int max = arr[0];

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

if (arr[i] > max)

max = arr[i];

}

// Create a count array to store the count of each element

int\* count = (int\*)calloc(max + 1, sizeof(int));

// Store the count of each element in the count array

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

count[arr[i]]++;

// Modify the count array to store the position of each element in the sorted array

for (int i = 1; i <= max; ++i)

count[i] += count[i - 1];

// Create a temporary array to store the sorted elements

int\* output = (int\*)malloc(n \* sizeof(int));

// Fill the output array using the count array

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

output[count[arr[i]] - 1] = arr[i];

count[arr[i]]--;

}

// Copy the sorted elements back to the original array

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

arr[i] = output[i];

// Free dynamically allocated memory

free(count);

free(output);

}

// Function to print the sorted array

void printArray(int arr[], int n) {

printf("Sorted array: ");

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

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

printf("\n");

}

// Main function

int main() {

int n;

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

scanf("%d", &n);

// Input elements of the array

int\* arr = (int\*)malloc(n \* sizeof(int));

printf("Enter %d elements:\n", n);

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

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

// Perform distribution counting sort

distributionCountingSort(arr, n);

// Print the sorted array

printArray(arr, n);

// Free dynamically allocated memory

free(arr);

return 0;

}

Write a program to implement Floyd’s algorithm for the All-Pairs- Shortest

Paths problem for any given graph and analyse its time efficiency.

// Floyd-Warshall Algorithm in C

#include <stdio.h>

// defining the number of vertices

#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

// Implementing floyd warshall algorithm

void floydWarshall(int graph[][nV]) {

int matrix[nV][nV], i, j, k;

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

for (j = 0; j < nV; j++)

matrix[i][j] = graph[i][j];

// Adding vertices individually

for (k = 0; k < nV; k++) {

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

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

if (matrix[i][k] + matrix[k][j] < matrix[i][j])

matrix[i][j] = matrix[i][k] + matrix[k][j];

}

}

}

printMatrix(matrix);

}

void printMatrix(int matrix[][nV]) {

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

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

if (matrix[i][j] == INF)

printf("%4s", "INF");

else

printf("%4d", matrix[i][j]);

}

printf("\n");

}

}

int main() {

int graph[nV][nV] = {{0, 3, INF, 5},

{2, 0, INF, 4},

{INF, 1, 0, INF},

{INF, INF, 2, 0}};

floydWarshall(graph);

}

HUFFMAN CODING

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Structure for a node in Huffman tree

struct Node {

char data;

unsigned freq;

struct Node \*left, \*right;

};

// Function to create a new node

struct Node\* newNode(char data, unsigned freq) {

struct Node\* temp = (struct Node\*)malloc(sizeof(struct Node));

temp->left = temp->right = NULL;

temp->data = data;

temp->freq = freq;

return temp;

}

// Structure for min heap node

struct MinHeap {

unsigned size;

unsigned capacity;

struct Node\*\* array;

};

// Function to create a min heap

struct MinHeap\* createMinHeap(unsigned capacity) {

struct MinHeap\* minHeap = (struct MinHeap\*)malloc(sizeof(struct MinHeap));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (struct Node\*\*)malloc(minHeap->capacity \* sizeof(struct Node\*));

return minHeap;

}

// Function to swap two nodes of min heap

void swapNode(struct Node\*\* a, struct Node\*\* b) {

struct Node\* t = \*a;

\*a = \*b;

\*b = t;

}

// Function to heapify at a given index

void minHeapify(struct MinHeap\* minHeap, int idx) {

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)

smallest = right;

if (smallest != idx) {

swapNode(&minHeap->array[smallest], &minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

// Function to check if size of heap is 1

int isSizeOne(struct MinHeap\* minHeap) {

return (minHeap->size == 1);

}

// Function to extract minimum value node from heap

struct Node\* extractMin(struct MinHeap\* minHeap) {

struct Node\* temp = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

// Function to insert new node to heap

void insertMinHeap(struct MinHeap\* minHeap, struct Node\* minHeapNode) {

++minHeap->size;

int i = minHeap->size - 1;

while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {

minHeap->array[i] = minHeap->array[(i - 1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;

}

// Function to build Huffman tree

struct Node\* buildHuffmanTree(char data[], int freq[], int size) {

struct Node \*left, \*right, \*top;

struct MinHeap\* minHeap = createMinHeap(size);

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

insertMinHeap(minHeap, newNode(data[i], freq[i]));

while (!isSizeOne(minHeap)) {

left = extractMin(minHeap);

right = extractMin(minHeap);

top = newNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

return extractMin(minHeap);

}

// Function to print Huffman codes from the tree

void printCodes(struct Node\* root, int arr[], int top) {

if (root->left) {

arr[top] = 0;

printCodes(root->left, arr, top + 1);

}

if (root->right) {

arr[top] = 1;

printCodes(root->right, arr, top + 1);

}

if (!(root->left) && !(root->right)) {

printf("%c: ", root->data);

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

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

printf("\n");

}

}

// Main function

int main() {

int n;

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

scanf("%d", &n);

char data[n];

int freq[n];

// Input characters and their frequencies

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

printf("Enter character %d: ", i + 1);

scanf(" %c", &data[i]);

printf("Enter frequency for character %c: ", data[i]);

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

}

clock\_t start\_time = clock(); // Start timer

// Build Huffman tree

struct Node\* root = buildHuffmanTree(data, freq, n);

clock\_t end\_time = clock(); // Stop timer

int arr[100], top = 0;

printf("Huffman Codes:\n");

printCodes(root, arr, top);

double time\_taken = ((double)(end\_time - start\_time)) / CLOCKS\_PER\_SEC;

printf("Time taken by algorithm: %f seconds\n", time\_taken);

return 0;

}