# **Experiment No – 1**

**AIM:** Finding Prime Numbers

**PROBLEM STATEMENT**: Write a program to display Prime Numbers between 1-100, and check whether the given number is Prime or Not.

**COURSE OBJECTIVE:** To understand the importance of number theory in CP

**COURSE OUTCOMES:** To understand the importance of number theory in CP

**THEORY:**

### The theory behind writing a program to display prime numbers between 1 and 100 and checking the primarily of a given number involves the following concepts:

### Prime Numbers: A prime number is a natural number greater than 1 that has no positive divisors other than and itself. For example, 2, 3, 5, 7, 11, etc., are prime numbers.

### Algorithm for Checking Prime Numbers: One of the simplest algorithms to check whether a number is prime or not is the Trial Division method. In this method, we iterate through all the numbers from 2 to the square root of the given number and check if any of them divides the number evenly. If such a divisor is found, the number is not prime; otherwise, it is prime.

### Program Structure: The program can be structured as follows:

### Write a function isPrime() that takes a number as input and returns 1 if it is prime, and 0 otherwise, based on the trial division algorithm.

### Implement a loop to iterate through the numbers from 1 to 100 and call the isPrime() function for each number.

### Inside the loop, if a number is determined to be prime, print it.

### Finally, ask the user to input a number and call the isPrime() function to check whether the given number is prime or not.By implementing this program, you can display all prime numbers between 1 and 100 and also check whether a given number is prime or not using the trial division algorithm.

### CODE:

### #include <stdio.h>

### int isPrime(int num) {

### if (num <= 1)

### return 0; // not a prime number

### for (int i = 2; i \* i <= num; i++) {

### if (num % i == 0)

### return 0; // not a prime number

### }

### return 1; // prime number

### }

### void displayPrimeNumbers(int start, int end) {

### printf("Prime numbers between %d and %d are:\n", start, end);

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

### if (isPrime(i))

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

### }

### printf("\n");

### }

### int main() {

### int number;

### printf("Enter a number: ");

### scanf("%d", &number);

### if (isPrime(number))

### printf("%d is a prime number.\n", number);

### else

### printf("%d is not a prime number.\n", number);

### displayPrimeNumbers(1, 100);

### return 0;

### }

### OUTCOMES:

### Enter a number: 5

### 5 is a prime number.

### Prime numbers between 1 and 100 are:

### 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

**Questions:**

### What is prime number?

1. Why is 2 a prime number?

### How do you know what a prime number is?

1. Write a program to find out prime no’s from 1-100 numbers in Java/c++/python.

### Attach code and Output for same problem Statement

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Marks Obtained** | | | | **Dated Signature of Teacher** |
| **Attendance**  **(2)** | **Learning Approach (4)** | **Implementation**  **(4)** | **Total (10)** |  |
|  |  |  |  |

# **Experiment No – 2**

**AIM:** String and Array Manipulation: Reverse, Concatenation Operations.

**PROBLEM STATEMENT**: Write a Program to perform Following operation: a) Reverse of String b) Reverse of Array c) Concatenation of Two String

**COURSE OBJECTIVE:** To understand the importance of number theory in CP

**COURSE OUTCOMES:** To understand the importance of number theory in CP

**THEORY:**

### Reverse of String:

### The reverseString() function takes a string as input and reverses it.

### It uses the two-pointer approach, where one pointer starts from the beginning of the string and another from the end.

### The characters at these positions are swapped, and the pointers move towards each other until they meet in the middle.

### This process continues until the entire string is reversed.

### Reverse of Array:

### The reverseArray() function takes an array and its size as input and reverses the elements.

### It also uses the two-pointer approach, similar to reversing a string.

### The function swaps the elements at corresponding positions from the start and end of the array, gradually moving towards the middle.

### This process continues until the entire array is reversed.

### Concatenation of Two Strings:

### The concatenateStrings() function takes two strings as input and concatenates them.

### It uses the strcat() function from the string.h library to append the contents of the second string to the end of the first string.

### The result is stored in the first string, which is modified in place.

### The concatenated string is then displayed as the output.

### Program Structure:

### The main() function demonstrates the usage of the reverseString(), reverseArray(), and concatenateStrings() functions.

### It initializes a string and an array with initial values.

### It calls the respective functions to perform the operations.

### The modified string and array are displayed as output.

### By implementing these operations, the program allows you to reverse a string, reverse an array, and concatenate two strings using appropriate functions and algorithms.

**CODE:**

#include <stdio.h>

#include <string.h>

// Function to reverse a string

void reverseString(char str[]) {

int length = strlen(str);

int start = 0;

int end = length - 1;

while (start < end) {

char temp = str[start];

str[start] = str[end];

str[end] = temp;

start++;

end--;

}

}

// Function to reverse an array

void reverseArray(int arr[], int size) {

int start = 0;

int end = size - 1;

while (start < end) {

int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

// Function to concatenate two strings

void concatenateStrings(char str1[], char str2[]) {

strcat(str1, str2);

}

int main() {

// Reverse a string

char string[] = "Hello, World!";

printf("Original string: %s\n", string);

reverseString(string);

printf("Reversed string: %s\n", string);

// Reverse an array

int array[] = {1, 2, 3, 4, 5};

int size = sizeof(array) / sizeof(array[0]);

printf("\nOriginal array: ");

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

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

}

reverseArray(array, size);

printf("\nReversed array: ");

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

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

}

// Concatenate two strings

char str1[100] = "Hello";

char str2[] = " World!";

printf("\n\nString 1: %s\n", str1);

printf("String 2: %s\n", str2);

concatenateStrings(str1, str2);

printf("Concatenated string: %s\n", str1);

return 0;

}

### OUTCOMES:

### Original string: Hello, World!

### Reversed string: !dlroW ,olleH

### Original array: 1 2 3 4 5

### Reversed array: 5 4 3 2 1

### String 1: Hello

### String 2: World!

### Concatenated string: Hello World!

**Questions:**

1. How does the reverseString() function in the program handle string reversal for strings of different lengths?
2. What is the time complexity of the reverseArray() function in terms of the array size
3. What is the purpose of using the strcpy() function in the concatenateStrings() function
4. How would you modify the program to handle multiple strings concatenation, instead of concatenating just two strings?
5. Attach code and Output for same problem Statement

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Marks Obtained** | | | | **Dated Signature of Teacher** |
| **Attendance**  **(2)** | **Learning Approach (4)** | **Implementation**  **(4)** | **Total (10)** |  |
|  |  |  |  |

# **Experiment No – 03**

**TITLE:** Program for Heap Sort Algorithm Implementation

**PROBLEM STATEMENT:** Write a program to Implement Heap sort Algorithm.

**COURSE OBJECTIVE:** Solve N-Queen Problem by using backtracking

**COURSE OUTCOMES:** CO1: Solve N-Queen Problem by using backtracking

### THEORY:

### Heap Data Structure:

### A heap is a complete binary tree that satisfies the heap property.

### In a max heap, for any node i, the value of arr[i] is greater than or equal to the values of its children.

### In a min heap, for any node i, the value of arr[i] is less than or equal to the values of its children.

### Heaps can be represented using arrays, where the parent-child relationship can be determined based on the indices of the array elements.

### Heapify Operation:

### Heapify is an operation that maintains the heap property of a binary tree.

### Given an array arr and an index i, heapify ensures that the subtree rooted at i satisfies the heap property.

### To heapify, the algorithm compares the root element with its children and swaps the root with the larger (or smaller) child if necessary.

### The heapify operation is applied recursively to the affected subtree until the heap property is satisfied.

### Building a Heap:

### Building a heap involves transforming an array into a heap data structure.

### The bottom-up approach is commonly used to build a heap. Starting from the last internal node (parent of the last leaf node), the heapify operation is performed on each internal node in reverse order.

### This process ensures that every subtree rooted at an internal node satisfies the heap property.

### Heap Sort Algorithm Steps:

### The Heap Sort algorithm utilizes the heap data structure to sort an array in ascending or descending order.

### First, the algorithm builds a max heap from the input array using the building heap process described above.

### Once the heap is built, the algorithm extracts the maximum element (root) from the heap and places it at the end of the array.

### The heapify operation is then performed on the remaining elements to restore the heap property.

### This process is repeated until all elements are extracted and the array is sorted.

### Time Complexity:

### Building a heap takes O(n) time, where n is the number of elements in the array.

### Extracting elements and restoring the heap property take O(log n) time for each element.

### Therefore, the overall time complexity of Heap Sort is O(n log n).

### The space complexity is O(1) as the sorting is done in-place.

### By utilizing the heap data structure, the heapify operation, and the building process, the Heap Sort algorithm efficiently sorts an array in ascending or descending order.

**CODE:**

#include <studio.h>

// Function to swap two elements

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to perform heapify on a subtree rooted at index i

void heapify(int arr[], 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 && arr[left] > arr[largest])

largest = left;

// If right child is larger than current largest

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

largest = right;

// If largest is not root

if (largest != i) {

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

// Recursively heapify the affected subtree

heapify(arr, n, largest);

}

}

// Heap Sort function

void heapSort(int arr[], int n) {

// Build heap (rearrange array)

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

heapify(arr, n, i);

// Extract elements from the heap one by one

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

// Move current root to the end

swap(&arr[0], &arr[i]);

// Heapify the reduced heap

heapify(arr, i, 0);

}

}

// Function to print an array

void printArray(int arr[], int n) {

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

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

printf("\n");

}

// Driver program

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

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

printf("Original array: ");

printArray(arr, n);

heapSort(arr, n);

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

### OUTCOMES:

### Original array: 12 11 13 5 6 7

### Sorted array: 5 6 7 11 12 13

### Questions

### Explain the Heap Sort algorithm and its steps in detail. How does it utilize the heap data structure to sort an array efficiently?

### Compare and contrast Heap Sort with other sorting algorithms, such as Quick Sort and Merge Sort. Discuss their time complexities, space complexities, and stability.

### Describe the process of building a max heap from an array. How does the bottom-up approach work, and what is its time complexity?

### Discuss the role of the heapify operation in the Heap Sort algorithm. How does it maintain the heap property and enable the sorting process? Provide an example illustrating the heapify operation.

5. Attach code and Output for same problem Statement

# 

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Marks Obtained** | | | | **Dated Signature of Teacher** |
| **Attendance**  **(2)** | **Learning Approach (4)** | **Implementation**  **(4)** | **Total (10)** |  |
|  |  |  |  |

# **Experiment No – 4**

### AIM: Solve N-Queen Problem.

**PROBLEM STATEMENT**: Write a program to Solve N-Queen Problem (4 Queen & 8 Queen).

**COURSE OBJECTIVE:** Solve N-Queen Problem by using backtracking

**COURSE OUTCOMES:** Solve N-Queen Problem by using backtracking

**THEORY:**

Backtracking:-Backtracking can be defined as a general algorithmic technique that considers searching every possible Combination in order to solve a computational problem. There are three types of problems in backtracking – Decision Problem – In this; we search for a feasible solution.

Backtracking is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that don’t give rise to the solution of the problem based on the constraints given to solve the problem.

Backtracking is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that don’t give rise to the solution of the problem based on the constraints given to solve the problem. Backtracking algorithm is applied to some specific types of problems,

• Decision problem used to find a feasible solution of the problem.

• Optimisation problem used to find the best solution that can be applied.

• Enumeration problem used to find the set of all feasible solutions of the problem.

In backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.

N-Queen Problem- In N-Queen problem, we are given an NxN chessboard and we have to place n queens on the board in such a way that no two queens attack each other. A queen will attack another queen if it is placed in horizontal, vertical or diagonal points in its way. Here, we will do 4-Queen problem.

### CODE: #include <stdio.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("%2d ", board[i][j]);

### }

### printf("\n");

### }

### printf("\n");

### }

### int isSafe(int board[N][N], int row, int col) {

### int i, j;

### // Check if there is a queen in the same row

### for (i = 0; i < col; i++) {

### if (board[row][i])

### return 0;

### }

### // Check if there is a queen in the upper left diagonal

### for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {

### if (board[i][j])

### return 0;

### }

### // Check if there is a queen in the lower left diagonal

### for (i = row, j = col; j >= 0 && i < N; i++, j--) {

### if (board[i][j])

### return 0;

### }

### return 1;

### }

### int solveNQueenUtil(int board[N][N], int col) {

### if (col >= N) {

### printSolution(board);

### return 1;

### }

### int res = 0;

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

### if (isSafe(board, i, col)) {

### board[i][col] = 1;

### res += solveNQueenUtil(board, col + 1);

### board[i][col] = 0;

### }

### }

### return res;

### }

### void solveNQueen(int n) {

### int board[N][N] = {0};

### int count = solveNQueenUtil(board, 0);

### printf("Total solutions for %d-Queen: %d\n", n, count);

### }

### int main() {

### printf("Solutions for 4-Queen Problem:\n");

### solveNQueen(4);

### printf("Solutions for 8-Queen Problem:\n");

### solveNQueen(8);

### return 0;

### }

### OUTCOME:

### 0 0 1 0 0 0 0 0 | 0 1 0 0

### 0 0 0 0 0 1 0 0 | 0 0 0 1 🡨 4 Queen

### 0 0 0 1 0 0 0 0 | 1 0 0 0

### 0 1 0 0 0 0 0 0 | 0 0 1 0

### 0 0 0 0 0 0 0 1

### 0 0 0 0 1 0 0 0

### 0 0 0 0 0 0 1 0 🡨 8 Queen

### 1 0 0 0 0 0 0 0

### Algorithm

Step 1 − Start from 1st position in the array.

Step 2 − Place queens in the board and check. Do,

Step 2.1 − After placing the queen, mark the position as a part of the solution and then recursively check if this will

lead to a solution.

Step 2.2 − Now, if placing the queen doesn’t lead to a solution and trackback and go to step (a) and place queens to

other rows.

Step 2.3 − If placing queen returns a lead to solution return **TRUE.**

Step 3 − If all queens are placed return TRUE.

Step 4 − If all rows are tried and no solution is found, return FALSE.

**Questions:**

1. What do you mean by backtracking?
2. Which data structure is used for backtracking?
3. What is the complexity of n queen problem?
4. How do you solve a n queen problem?
5. Attach code and Output for same problem Statement

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Marks Obtained** | | | | **Dated Signature of Teacher** |
| **Attendance**  **(2)** | **Learning Approach (4)** | **Implementation**  **(4)** | **Total (10)** |  |
|  |  |  |  |

# **Experiment No – 5**

**AIM:** Finding Divisibility Greatest Common Divisor

**PROBLEM STATEMENT**: Finding Divisibility Greatest Common Divisor

**COURSE OBJECTIVE:** To understand importance of number theory in CP

**COURSE OUTCOMES:**

**THEORY:**

GCD of two or more integers is the largest positive integer that divides the given integer without any remainder For example, Java GCD of 8 and 12 is 4 because both 8 and 12 are divisible by 1, 2, and 4. The largest positive integer among the factors 1, 2, and 4 is 4.

In mathematics, the greatest common divisor (GCD) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers. For two integers x, y, the greatest common divisor of x and y is denoted

The greatest common divisor (GCD) of two nonzero integers a and b is the greatest positive integer d such that d is a divisor of both a and b; that is, there are integers e and f such that a = de and b = df, and d is the largest such integer. The GCD of a and b is generally denoted gcd(a, b).

This definition also applies when one of a and b is zero. In this case, the GCD is the absolute value of the non zero integer: gcd(a, 0) = gcd(0, a) = |a|. This case is important as the terminating step of the Euclidean algorithm.

The above definition cannot be used for defining gcd(0, 0), since 0 × n = 0, and zero thus has no greatest divisor. However, zero is its own greatest divisor if greatest is understood in the context of the divisibility relation, so gcd(0, 0) is commonly defined as 0. This preserves the usual identities for GCD, and in particular Bézout's identity, namely that gcd(a, b) generates the same ideal as {a, b}.[10][11][12] This convention is followed by many computer algebra systems.[13] Nonetheless, some authors leave gcd(0, 0) undefined.

The GCD of a and b is their greatest positive common divisor in the partial order relation of divisibility. This means that the common divisors of a and b are exactly the divisors of their GCD. This is commonly proved by using Euclid’s lemma, the fundamental theorem of arithmetic, or the Euclidean algorithm. This is the meaning of "greatest" that is used for the generalizations of the concept of GCD.

### Code:

### #include <stdio.h>

### // Function to calculate the GCD of two numbers

### int gcd(int a, int b) {

### // Base case: If b is 0, the GCD is a

### if (b == 0) {

### return a;

### }

### 

### // Recursive case: Calculate GCD using Euclidean algorithm

### return gcd(b, a % b);

### }

### int main() {

### int num1, num2;

### printf("Enter two numbers: ");

### scanf("%d %d", &num1, &num2);

### // Calculate the GCD

### int result = gcd(num1, num2);

### printf("The GCD of %d and %d is: %d\n", num1, num2, result);

### return 0;

### }

### Outcome:

### Enter two numbers: 8 10

### The GCD of 8 and 10 is: 2

**Questions:**

1. How do you find the greatest common divisor?

2. How do I find the greatest common divisor in Python?

3. What are common Factors?

4. What are the methods to find GCD?

5. Attach code and Output for same problem Statement

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Marks Obtained** | | | | **Dated Signature of Teacher** |
| **Attendance**  **(2)** | **Learning Approach (4)** | **Implementation**  **(4)** | **Total (10)** |  |
|  |  |  |  |

# **Experiment No – 6**

**AIM:** Study DFS algorithm

**PROBLEM STATEMENT**: Write a program for DFS Searching algorithm.

**COURSE OBJECTIVE:** Study DFS algorithm

**COURSE OUTCOMES:** Student will able to search the element using DFS algorithm

**THEORY:**

**Depth First Search (DFS) Algorithm**

It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children. Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

1. The step by step process to implement the DFS traversal is given as follows - First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

**Applications of DFS algorithm:**

The applications of using the DFS algorithm are given as follows -

1. DFS algorithm can be used to implement the topological sorting.
2. It can be used to find the paths between two vertices.
3. It can also be used to detect cycles in the graph.
4. DFS algorithm is also used for one solution puzzles.
5. DFS is used to determine if a graph is bipartite or not.

**Algorithm:**

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

**CODE:**

/\*A sample java program to implement the DFS algorithm\*/

import java.util.\*;

class DFSTraversal {

private LinkedList<Integer> adj[]; /\*adjacency list representation\*/

private boolean visited[];

/\* Creation of the graph \*/

DFSTraversal(int V) /\*'V' is the number of vertices in the graph\*/

{

adj = new LinkedList[V];

visited = new boolean[V];

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

adj[i] = new LinkedList<Integer>();

}

/\* Adding an edge to the graph \*/

void insertEdge(int src, int dest) {

adj[src].add(dest);

}

void DFS(int vertex) {

visited[vertex] = true; /\*Mark the current node as visited\*/

System.out.print(vertex + " ");

Iterator<Integer> it = adj[vertex].listIterator();

while (it.hasNext()) {

int n = it.next();

if (!visited[n])

DFS(n);

}

}

public static void main(String args[]) {

DFSTraversal graph = new DFSTraversal(8);

graph.insertEdge(0, 1);

graph.insertEdge(0, 2);

graph.insertEdge(0, 3);

graph.insertEdge(1, 3);

graph.insertEdge(2, 4);

graph.insertEdge(3, 5);

graph.insertEdge(3, 6);

graph.insertEdge(4, 7);

graph.insertEdge(4, 5);

graph.insertEdge(5, 2);

System.out.println("Depth First Traversal for the graph is:");

graph.DFS(0);

}

}

QUESTION:

1. What is DFS algorithm?

2. How does DFS algorithm work? What is DFS used for?

3. What is difference Between DFS and BFS?

4. What is time complexity of DFS?

5. Attach code and Output for same problem Statement

ANS:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Marks Obtained** | | | | **Dated Signature of Teacher** |
| **Attendance**  **(2)** | **Learning Approach (4)** | **Implementation**  **(4)** | **Total (10)** |  |
|  |  |  |  |

# **Experiment No – 7**

**AIM:** Study BFS algorithm

**PROBLEM STATEMENT**: Write a program for BFS Searching algorithm.

**COURSE OBJECTIVE:** Study BFS algorithm

**COURSE OUTCOMES:** Student will able to search the element using BFS algorithm

**THEORY:**

**BFS algorithm**

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

**Applications of BFS algorithm**

The applications of breadth-first-algorithm are given as follows –

1. BFS can be used to find the neighboring locations from a given source location.
2. In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.
3. BFS can be used in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages. It starts traversing from the source page and follows the links associated with the page. Here, every web page is considered as a node in the graph.
4. BFS is used to determine the shortest path and minimum spanning tree.
5. BFS is also used in Cheney's technique to duplicate the garbage collection.
6. It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

**Algorithm:**

The steps involved in the BFS algorithm to explore a graph are given as follows -

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2

(waiting state)

[END OF LOOP]

Step 6: EXIT

**CODE:**

import java.io.\*;

import java.util.\*;

public class BFSTraversal

{

private int vertex; /\* total number number of vertices in the graph \*/

private LinkedList<Integer> adj[]; /\* adjacency list \*/

private Queue<Integer> que; /\* maintaining a queue \*/

BFSTraversal(int v)

{

vertex = v;

adj = new LinkedList[vertex];

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

{

adj[i] = new LinkedList<>();

}

que = new LinkedList<Integer>();

}

void insertEdge(int v,int w)

{

adj[v].add(w); /\* adding an edge to the adjacency list (edges are bidirectional in this example) \*/

}

void BFS(int n)

{

boolean nodes[] = new boolean[vertex]; /\* initialize boolean array for holding the data \*/

int a = 0;

nodes[n]=true;

que.add(n); /\* root node is added to the top of the queue \*/

while (que.size() != 0)

{

n = que.poll(); /\* remove the top element of the queue \*/

System.out.print(n+" "); /\* print the top element of the queue \*/

for (int i = 0; i < adj[n].size(); i++) /\* iterate through the linked list and push all neighbors into queue \*/

{

a = adj[n].get(i);

if (!nodes[a]) /\* only insert nodes into queue if they have not been explored already \*/

{

nodes[a] = true;

que.add(a);

}

}

}

}

public static void main(String args[])

{

BFSTraversal graph = new BFSTraversal(10);

graph.insertEdge(0, 1);

graph.insertEdge(0, 2);

graph.insertEdge(0, 3);

graph.insertEdge(1, 3);

graph.insertEdge(2, 4);

graph.insertEdge(3, 5);

graph.insertEdge(3, 6);

graph.insertEdge(4, 7);

graph.insertEdge(4, 5);

graph.insertEdge(5, 2);

graph.insertEdge(6, 5);

graph.insertEdge(7, 5);

graph.insertEdge(7, 8);

System.out.println("Breadth First Traversal for the graph is:");

graph.BFS(2);

}

}

QUESTION:

1. What is BFS algorithm ?

2. What are the disadvantages of BFS algorithm ?

3. Why BFS takes more memory than DFS?

4. What is the time complexity of BFS?

5. Attach code and Output for same problem Statement

ANS:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Marks Obtained** | | | | **Dated Signature of Teacher** |
| **Attendance**  **(2)** | **Learning Approach (4)** | **Implementation**  **(4)** | **Total (10)** |  |
|  |  |  |  |

**AIM:** Study calculate minimum distance using MST algorithm

# **Experiment No – 8**

**PROBLEM STATEMENT**: Write a program to Display minimum spanning tree using Prims Algorithm.

**COURSE OBJECTIVE:** To understand importance MST

**COURSE OUTCOMES:** Student will able to span tree using prims algorithm **THEORY:**

A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected. spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

**Applications of the spanning tree:**

Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -

1. Cluster Analysis
2. Civil network planning
3. Computer network routing protocol

**CODE:**

#include <iostream>

#include<bits/stdc++.h>

#include <cstring>

using namespace std;

// number of vertices in graph

#define V 7

// create a 2d array of size 7x7

//for adjacency matrix to represent graph

int main () {

// create a 2d array of size 7x7

//for adjacency matrix to represent graph

int G[V][V] = {

{0,28,0,0,0,10,0},

{28,0,16,0,0,0,14},

{0,16,0,12,0,0,0},

{0,0,12,22,0,18},

{0,0,0,22,0,25,24},

{10,0,0,0,25,0,0},

{0,14,0,18,24,0,0}

};

int edge; // number of edge

// create an array to check visited vertex

int visit[V];

//initialise the visit array to false

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

visit[i]=false;

}

// set number of edge to 0

edge = 0;

// the number of edges in minimum spanning tree will be

// always less than (V -1), where V is the number of vertices in

//graph

// choose 0th vertex and make it true

visit[0] = true;

int x; // row number

int y; // col number

// print for edge and weight

cout << "Edge" << " : " << "Weight";

cout << endl;

while (edge < V - 1) {//in spanning tree consist the V-1 number of edges

//For every vertex in the set S, find the all adjacent vertices

// , calculate the distance from the vertex selected.

// if the vertex is already visited, discard it otherwise

//choose another vertex nearest to selected vertex.

int min = INT\_MAX;

x = 0;

y = 0;

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

if (visit[i]) {

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

if (!visit[j] && G[i][j]) { // not in selected and there is an edge

if (min > G[i][j]) {

min = G[i][j];

x = i;

y = j;

}

}

}

}

}

cout << x << " ---> " << y << " : " << G[x][y];

cout << endl;

visit[y] = true;

edge++;

}

return 0;

}

QUESTION:

1) What is minimum spanning tree?

2) Which is better Kruskal or Prims?

3) What is the best-case time complexity of Prims?

4) What is complexity of Prim's algorithm?

5) Attach Code and output for same Problem Statement

ANS:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Marks Obtained** | | | | **Dated Signature of Teacher** |
| **Attendance**  **(2)** | **Learning Approach (4)** | **Implementation**  **(4)** | **Total (10)** |  |
|  |  |  |  |

# **Experiment No – 9**

**AIM:** To find Shortest path

**PROBLEM STATEMENT**: Write a Program for Travelling Sales Person Problem

**COURSE OBJECTIVE:** To study dynamic programming approach of Traveling Sales Person

**COURSE OUTCOMES:** Students will able to traverse the graph using Traveling Sales Person

**THEORY:**

The traveling salesman problems abide by a salesman and a set of cities. The salesman has to visit every one of the cities starting from a certain one (e.g., the hometown) and to return to the same city. The challenge of the problem is that the traveling salesman needs to minimize the total length of the trip. Suppose the cities are x1 x2..... xn where cost cij denotes the cost of travelling from city xi to xj. The travelling salesperson problem is to find a route starting and ending at x1 that will take in all cities with the minimum cost.

**Algorithm:**

* Travelling salesman problem takes a graph G {V, E} as an input and declare another graph as the output (say G’) which will record the path the salesman is going to take from one node to another.
* The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
* The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
* Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.
* Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.
* However, if the origin is mentioned in the given problem, then the solution must always start from that node only. Let us look at some example problems to understand this better.

**CODE:**

import java.util.\*;

public class TSPGREEDY {

public static void main(String[] args) {

int[

][] tsp\_g = {

{ -1, 10, 20, 30 },

{ 10, -1, 20, 25 },

{ 15, 30, -1, 10 },

{ 20, 15, 40, -1 }

};

int cost = 0;

int count = 0;

int j = 0, i = 0;

int min = Integer.MAX\_VALUE;

List<Integer> visited = new ArrayList<>();

// The problem starts from 0th index city

visited.add(0);

int[] path = new int[tsp\_g.length];

while (i < tsp\_g.length && j < tsp\_g[i].length) {

if (count >= tsp\_g[i].length - 1) {

break;

}

// If the city is unvisited and has minimum cost, update the cost

if (j != i && !(visited.contains(j))) {

if (tsp\_g[i][j] < min) {

min = tsp\_g[i][j];

path[count] = j + 1;

}

}

j++;

// Check all paths from the

// ith indexed city

if (j == tsp\_g[i].length) {

cost += min;

min = Integer.MAX\_VALUE;

visited.add(path[count] - 1);

j = 0;

i = path[count] - 1;

count++;

}

}

// Update the ending city in array

// from city which was last visited

i = path[count - 1] - 1;

for (j = 0; j < tsp\_g.length; j++) {

if ((i != j) && tsp\_g[i][j] < min) {

min = tsp\_g[i][j];

path[count] = j + 1;

}

}

cost += min;

// Started from the node where

// we finished as well.

System.out.print("Minimum Cost is : ");

System.out.println(cost);

}

}

QUESTION:

* 1. Define Traveling sales person Algortihm ?
  2. Give the example of Travelling sales person ?
  3. What is the time complexity of the Travelling Sales person using Greedy approach and Dynamic Programming ?
  4. What are the challenges of Travelling Sales person Algorithm ?
  5. Attach code and output for same problem Statement

ANS:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Marks Obtained** | | | | **Dated Signature of Teacher** |
| **Attendance**  **(2)** | **Learning Approach (4)** | **Implementation**  **(4)** | **Total (10)** |  |
|  |  |  |  |

# **Experiment No – 10**

**AIM:** Study Bicoloring problem

**PROBLEM STATEMENT**: To Study Bi-coloring problem.

**COURSE OBJECTIVE:** Study Bi-coloring problem

**COURSE OUTCOMES:** student will be able to solve the Bi-coloring problem

**THEORY:**

Graph coloring is the procedure of assignment of colors to each vertex of a graph G such that no adjacent vertices get same color. The objective is to minimize the number of colors while coloring a graph. The smallest number of colors required to color a graph G is called its chromatic number of that graph. Graph coloring problem is a NP Complete problem.

## Method to Color a Graph

The steps required to color a graph G with n number of vertices are as follows −

**Step 1** − Arrange the vertices of the graph in some order.

**Step 2** − Choose the first vertex and color it with the first color.

**Step 3** − Choose the next vertex and color it with the lowest numbered color that has not been colored on any vertices adjacent to it. If all the adjacent vertices are colored with this color, assign a new color to it. Repeat this step until all the vertices are colored.

## Applications of Graph Coloring

Some applications of graph coloring include −

* [Register Allocation](https://en.wikipedia.org/wiki/Register_allocation)
* Map Coloring
* Bipartite Graph Checking
* [Mobile Radio Frequency Assignment](https://www.zib.de/groetschel/teaching/SS2012/GraphCol%20and%20FrequAssignment.pdf)
* Making time table, etc.

**import** java.util.LinkedList;

**import** java.util.\*;

**import** java.io.\*;

// The class shows an undirected graph using an adjacency list

**public** **class** Graphs

{

**private** **int** N; // No. of nodes

**private** LinkedList<Integer> adjList[]; //Adjacency List

//Constructor

Graphs(**int** n)

{

N = n;

adjList = **new** LinkedList[n];

**for** (**int** i = 0; i < n; i++)

{

adjList[i] = **new** LinkedList();

}

}

// Method to create an edge into the graph

// from node x to y and y to x

**void** addingEdge(**int** x, **int** y)

{

// The Graph is not directed

adjList[x].add(y);

adjList[y].add(x);

}

// A method that finds the chromatic number of a graph

**void** findChromticNo(**int** arr[])

{

// caculating the size of the array

**int** size = arr.length;

Set<Integer> hashSet = **new** HashSet<Integer>();

// iterating over every node and

// storing its color in the hashset

**for**(**int** j = 0; j < size; j++)

{

// hashset only contains unique numbers.

hashSet.add(arr[j]);

}

// finding the chromatic Number of the graph

**int** chromaticNo = hashSet.size();

System.out.println("The chromatic number of the graph is: " + chromaticNo);

}

// Assigning colors (beginning from 0) to all the nodes and

// displaying the assignment of colors

**void** greedyColorNodes()

{

**int** res[] = **new** **int**[N];

// Initializing all the vertices as unassigned

Arrays.fill(res, -1);

// Assiging the first color to the first vertex

res[0] = 0;

// A temporary array in order to keep the available colors. A False

// value of the avail[clr] will mean that the color clr has been assigned

// to one of its adjacent nodes

**boolean** avail[] = **new** **boolean**[N];

// In the beginning, all of the colors are available

Arrays.fill(avail, **true**);

// Assign colors to theh remaining N - 1 nodes

**for** (**int** n = 1; n < N; n++)

{

// Processing all the adjacent nodes and flag their colors as unavailable

Iterator<Integer> itr = adjList[n].iterator() ;

**while** (itr.hasNext())

{

**int** i = itr.next();

**if** (res[i] != -1)

    avail[res[i]] = **false**;

}

// Find the first color that is available

**int** clr;

**for** (clr = 0; clr < N; clr++)

{

**if** (avail[clr])

{

**break**;

}

}

res[n] = clr; // Assigning the found color

// For the next iteration, resetting the values back to true

Arrays.fill(avail, **true**);

}

// printing the result

**for** (**int** n = 0; n < N; n++)

{

System.out.println("Node " + n + " ---> Color - " + res[n]);

}

// for finding the chromatic number of the graph

findChromticNo(res);

}

// main method

**public** **static** **void** main(String argvs[])

{

// creating a graph that contains

// 5 nodes

Graphs graph1 = **new** Graphs(5);

// creating edges between nodes

graph1.addingEdge(0, 1);

graph1.addingEdge(0, 2);

graph1.addingEdge(1, 2);

graph1.addingEdge(1, 3);

graph1.addingEdge(2, 3);

graph1.addingEdge(3, 4);

System.out.println("Coloring of the graph 1 is: ");

// invoking the method greedyColorNodes() to color the nodes

graph1.greedyColorNodes();

System.out.println();

// creating a graph that contains

// 4 nodes

Graphs graph2 = **new** Graphs(4);

System.out.println("Coloring of the graph 2 is: ");

// creating edges between nodes

graph2.addingEdge(0, 1);

graph2.addingEdge(0, 2);

graph2.addingEdge(1, 3);

graph2.addingEdge(2, 3);

// invoking the method greedyColorNodes() to color the nodes

graph2.greedyColorNodes();

}

}

QUESTION:

1. What is Bicoloring problem?

2. What is Time complexity of Bicoloring problem?

3. What are application Bicoloring Problems?

4. How is a Bicolor problem solved?

5. Attach code and output for same problem Statement

ANS:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Marks Obtained** | | | | **Dated Signature of Teacher** |
| **Attendance**  **(2)** | **Learning Approach (4)** | **Implementation**  **(4)** | **Total (10)** |  |
|  |  |  |  |

# **Experiment No – 11**

**AIM:** To implement mini project

**PROBLEM STATEMENT**:

Attach Code and Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Marks Obtained** | | | | **Dated Signature of Teacher** |
| **Attendance**  **(2)** | **Learning Approach (4)** | **Implementation**  **(4)** | **Total (10)** |  |
|  |  |  |  |