**DAA Lab Manual**

**PART-A**

**1. Design and implement an algorithm for computing Greatest Common divisor (GCD) of 2 numbers, say m and n, using the following approaches. i. ii. iii. Middle school procedure Euclid's Algorithm by Recursion Consecutive Integer Checking Method**

**Compute the Time Complexity for each and Display the GCD (m, n) where m>n.**

**1. Middle School Procedure**

**Algorithm**

1. Find all divisors of **m** and **n**.
2. Identify the common divisors of **m** and **n**.
3. The largest common divisor is the **GCD**.

**Algorithm Steps**

Step 1: Input two numbers m and n (where m > n).

Step 2: Initialize an empty list for factors of m and n.

Step 3: Find all divisors of m and store them in a list.

Step 4: Find all divisors of n and store them in a list.

Step 5: Find the common divisors from both lists.

Step 6: Return the maximum value from the common divisors list as GCD.

Step 7: End.

**Time Complexity: O(min(m,n))**

**2. Euclidean Algorithm (Recursive)**

**Algorithm**

1. If **n = 0**, return **m** as the **GCD**.
2. Otherwise, compute GCD(n, m % n).
3. Continue until **n** becomes **0**.

**Algorithm Steps**

Step 1: Input two numbers m and n (where m > n).

Step 2: If n == 0, return m as GCD.

Step 3: Else, compute GCD(n, m % n).

Step 4: Repeat until n becomes 0.

Step 5: Return the final value of m as GCD.

Step 6: End.

**Time Complexity: O(log(min(m,n))) (Most efficient)**

**3. Consecutive Integer Checking Method**

**Algorithm**

1. Find the smaller number between **m** and **n**.
2. Start from that number and decrement by **1**.
3. Check if the number divides both **m** and **n**.
4. The first number that divides both is the **GCD**.

**Algorithm Steps**

Step 1: Input two numbers m and n (where m > n).

Step 2: Set gcd = min(m, n).

Step 3: While gcd > 0:

- If m % gcd == 0 and n % gcd == 0, return gcd.

- Else, decrement gcd by 1.

Step 4: Return 1 if no common divisor is found.

Step 5: End.

**Time Complexity: O(min(m,n)) (Slowest in worst cases)**

**Conclusion**

* **Use Euclidean Algorithm** for the fastest computation.
* **Middle School Procedure** is simple but slower.
* **Consecutive Integer Method** is the slowest.

**Note:**

 **Middle School Procedure**

* **Best for**: Teaching basic concepts in GCD computation.
* **Complexity**: O(min(m,n)), but slower due to listing factors.
* **Good for**: Small numbers and educational purposes.

 **Euclidean Algorithm (Recursive)**

* **Best for**: Fast computation, even for large numbers.
* **Complexity**: O(log(min(m,n))) (most efficient).
* **Good for**: Practical applications in computing.

 **Consecutive Integer Checking Method**

* **Best for**: Understanding how GCD works step-by-step.
* **Complexity:**(min(m, n)) (slowest in worst cases).
* **Good for**: Learning purposes, not recommended for large numbers.

**Program:c**

#include <stdio.h>

#include <time.h>

#include <math.h>

// Middle School Procedure (Average speed)

int middle\_school\_gcd(int m, int n) {

int gcd = 1;

int min\_val = (m < n) ? m : n;

for (int i = 2; i <= sqrt(min\_val); i++) { // Checking only up to sqrt(min\_val)

while (m % i == 0 && n % i == 0) {

gcd \*= i;

m /= i;

n /= i;

}

}

if (m > 1 && n % m == 0) gcd \*= m;

if (n > 1 && m % n == 0) gcd \*= n;

return gcd;

}

// Euclidean Algorithm (Fastest)

int euclidean\_gcd(int m, int n) {

while (n != 0) {

int temp = n;

n = m % n;

m = temp;

}

return m;

}

// Consecutive Integer Method (Slowest)

int consecutive\_integer\_gcd(int m, int n) {

int gcd = (m < n) ? m : n;

while (gcd > 0) {

if (m % gcd == 0 && n % gcd == 0)

return gcd;

gcd--;

}

return 1;

}

// Function to measure time

double measure\_time(int (\*gcd\_func)(int, int), int m, int n) {

clock\_t start, end;

start = clock();

int gcd = gcd\_func(m, n);

end = clock();

printf("GCD: %d\n", gcd);

return ((double)(end - start)) / CLOCKS\_PER\_SEC;

}

int main() {

int m = 123456789, n = 987654321;

double time\_taken;

printf("Middle School GCD (Average Speed)\n");

time\_taken = measure\_time(middle\_school\_gcd, m, n);

printf("Time Taken: %lf sec\n\n", time\_taken);

printf("Euclidean GCD (Fastest)\n");

time\_taken = measure\_time(euclidean\_gcd, m, n);

printf("Time Taken: %lf sec\n\n", time\_taken);

printf("Consecutive Integer GCD (Slowest)\n");

time\_taken = measure\_time(consecutive\_integer\_gcd, m, n);

printf("Time Taken: %lf sec\n", time\_taken);

return 0;

}

**Output:**

Middle School GCD (Average Speed)

GCD: 9

Time Taken: 0.000054 sec

Euclidean GCD (Fastest)

GCD: 9

Time Taken: 0.000000 sec

Consecutive Integer GCD (Slowest)

GCD: 9

Time Taken: 0.232915 sec

### ****Analysis of Execution Time:****

| **Method** | **GCD Output** | **Time Taken (sec)** | **Efficiency** |
| --- | --- | --- | --- |
| **Middle School Procedure** | 9 | 0.000054 sec | **Slowest** (O(min(m, n))) |
| **Euclidean Algorithm** | 9 | 0.000000 sec | **Fastest** (O(log(min(m, n)))) |
| **Consecutive Integer Check** | 9 | 0.232915 sec | **Inefficient** (O(min(m, n))) |

**2.Design and Implement algorithm for searching techniques Linear Search and Binary Search(iterative/recursive). Compute the Time Complexity and Display.**

**1. Linear Search**

**Algorithm**

1. Start from the first element of the array.
2. Compare each element with the target value.
3. If found, return the index.
4. If not found, return **-1**.

**Time Complexity**

* **Best Case**: O(1) (if the element is at the first position).
* **Worst Case**: O(n) (if the element is at the last position or not present).
* **Average Case**: O(n).

**2. Binary Search (Iterative & Recursive)**

**Algorithm (Binary Search)**

1. Sort the array (if not sorted).
2. Find the middle element.
3. If the middle element is the target, return the index.
4. If the target is smaller, search in the left half.
5. If the target is larger, search in the right half.
6. Repeat until the target is found or the search range is empty.

**Time Complexity**

* **Best Case**: O(1) (if the middle element is the target).
* **Worst Case**: O(logn).
* **Average Case**: O(logn).

**Implementation in C**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Linear Search

int linear\_search(int arr[], int n, int target) {

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

if (arr[i] == target)

return i;

}

return -1;

}

// Binary Search (Iterative)

int binary\_search\_iterative(int arr[], int n, int target) {

int left = 0, right = n - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target)

return mid;

else if (arr[mid] < target)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

// Binary Search (Recursive)

int binary\_search\_recursive(int arr[], int left, int right, int target) {

if (left > right)

return -1;

int mid = left + (right - left) / 2;

if (arr[mid] == target)

return mid;

else if (arr[mid] < target)

return binary\_search\_recursive(arr, mid + 1, right, target);

else

return binary\_search\_recursive(arr, left, mid - 1, target);

}

// Measure execution time for functions with signature: int func(int[], int, int)

double measure\_time\_generic(int (\*func)(int[], int, int), int arr[], int n, int target, int iterations) {

clock\_t start = clock();

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

func(arr, n, target);

}

clock\_t end = clock();

return ((double)(end - start)) / CLOCKS\_PER\_SEC;

}

// Measure execution time for the recursive binary search (different signature)

double measure\_time\_recursive(int arr[], int left, int right, int target, int iterations) {

clock\_t start = clock();

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

binary\_search\_recursive(arr, left, right, target);

}

clock\_t end = clock();

return ((double)(end - start)) / CLOCKS\_PER\_SEC;

}

int main() {

int n = 100000; // 10 million elements

// Allocate memory on the heap

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

if (arr == NULL) {

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

return 1;

}

// Initialize array with sorted values (even numbers)

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

arr[i] = i \* 2;

}

int target = 98764; // A target value to search for

int iterations = 10000; // Number of iterations to average execution time

double time\_linear = measure\_time\_generic(linear\_search, arr, n, target, iterations);

double time\_iterative = measure\_time\_generic(binary\_search\_iterative, arr, n, target, iterations);

double time\_recursive = measure\_time\_recursive(arr, 0, n - 1, target, iterations);

printf("Linear Search Time: %f sec\n", time\_linear);

printf("Binary Search (Iterative) Time: %f sec\n", time\_iterative);

printf("Binary Search (Recursive) Time: %f sec\n", time\_recursive);

// Free allocated memory

free(arr);

return 0;

}

**Output:**

Linear Search Time: 0.682518 sec

Binary Search (Iterative) Time: 0.000598 sec

Binary Search (Recursive) Time: 0.000850 sec

**Time Complexity:**

| **Search Algorithm** | **Best Case** | **Worst Case** |
| --- | --- | --- |
| **Linear Search** | O(1) | O(n) |
| **Binary Search (Iterative)** | O(1) | O(logn) |
| **Binary Search (Recursive)** | O(1) | O(logn) |

**Key Observations:**

* **Binary Search (Iterative & Recursive) is much faster** than Linear Search for large datasets.
* **Linear Search works on unsorted arrays**, while **Binary Search requires a sorted array**.

**3. Consider the problem: You have a row of binary digits arranged randomly. Arrange them in such an order that all 0's precede all l's or vice-versa. The only constraint in arranging them is that you are allowed to interchange the positions of binary digits if they are not similar.**

**" Implement an algorithm for Merge-sort for binary value as input like 110101000.**

**"Compute the Time Complexity and Display the output as 0 0 0001111.**

**Merge Sort Steps:**

1. **Divide** the binary sequence into two halves recursively.
2. **Sort** each half separately.
3. **Merge** the two halves while ensuring all 0s appear before 1s.

Since **Merge Sort** is a stable sorting algorithm, it maintains the relative order of elements (important if we extend this to other cases).

**Time Complexity:**

* **Merge Sort runs in** O(nlogn)**time complexity**.
* Since it sorts recursively and merges efficiently, it works well for large input sizes.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

// Merge function for merge sort

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1; // Size of left subarray

int n2 = right - mid; // Size of right subarray

// Create temporary arrays

int \*L = (int \*)malloc(n1 \* sizeof(int));

int \*R = (int \*)malloc(n2 \* sizeof(int));

// Copy data to temporary arrays L[] and R[]

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

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

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

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

// Merge the temporary arrays back into arr[left..right]

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

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

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

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

} else {

arr[k++] = R[j++];

}

}

// Copy any remaining elements of L[]

while (i < n1) {

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

}

// Copy any remaining elements of R[]

while (j < n2) {

arr[k++] = R[j++];

}

free(L);

free(R);

}

// Merge Sort function

void mergeSort(int arr[], int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid); // Sort first half

mergeSort(arr, mid + 1, right); // Sort second half

merge(arr, left, mid, right); // Merge the sorted halves

}

}

// Convert input binary string into integer array

int\* convertStringToArray(char \*str, int \*n) {

\*n = strlen(str);

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

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

// Convert character '0' or '1' to integer 0 or 1

arr[i] = str[i] - '0';

}

return arr;

}

int main() {

// Input binary string

char input[] = "110101000";

int n;

int \*arr = convertStringToArray(input, &n);

// Measure time taken by merge sort

clock\_t start = clock();

mergeSort(arr, 0, n - 1);

clock\_t end = clock();

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

// Display sorted output as a single continuous string: 000001111

printf("Sorted Output: ");

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

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

}

printf("\n");

// Display formatted output:

// First print all 0's then a space then all 1's.

printf("Formatted Output: ");

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

if (arr[i] == 0)

printf("0");

}

printf(" ");

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

if (arr[i] == 1)

printf("1");

}

printf("\n");

// Display time complexity and time taken

printf("Time Complexity: O(n log n)\n");

printf("Time Taken: %f sec\n", time\_spent);

free(arr);

return 0;

}

**Output:**

Sorted Output: 000001111

Formatted Output: 00000 1111

Time Complexity: O(n log n)

Time Taken: 0.000004 sec

**4.Design and Implement a Quick Sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted. The elements can be read from the user or can be generated using the random number generator.**

## ****Quick Sort Algorithm****

### ****Algorithm****

1. **Choose a pivot** (last element in this case).
2. **Partition** the array such that:
   * Elements **smaller** than the pivot go to the left.
   * Elements **greater** than the pivot go to the right.
3. **Recursively apply Quick Sort** to both partitions.
4. **Base Case**: If the sub-array has one or zero elements, it is already sorted.

**Time Complexity:**

* **Best & Average Case**: O(nlogn)
* **Worst Case** (when the pivot is always the smallest or largest): O(n2)

## ****Implementation in C****

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Swap two elements in the array

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Partition function for Quick Sort

int partition(int arr[], int low, int high) {

int pivot = arr[high]; // pivot element (last element)

int i = low - 1; // index of smaller element

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

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

}

}

swap(&arr[i + 1], &arr[high]);

return i + 1;

}

// Quick Sort algorithm

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

if (low < high) {

int pi = partition(arr, low, high); // partitioning index

quickSort(arr, low, pi - 1); // Recursively sort left subarray

quickSort(arr, pi + 1, high); // Recursively sort right subarray

}

}

int main() {

int experiments;

printf("Enter number of experiments: ");

scanf("%d", &experiments);

// Initialize random seed once

srand(time(NULL));

for (int e = 0; e < experiments; e++) {

int n;

printf("Enter number of elements for experiment %d: ", e + 1);

scanf("%d", &n);

// Allocate memory for n elements

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

if (arr == NULL) {

printf("Memory allocation failed for %d elements!\n", n);

return 1;

}

// Generate n random elements (range: 0 to 9999)

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

arr[i] = rand() % 10000;

}

// Measure the time taken by Quick Sort

clock\_t start = clock();

quickSort(arr, 0, n - 1);

clock\_t end = clock();

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

printf("Time Taken to sort %d elements: %f sec\n", n, time\_spent);

// Optionally, print the sorted array for small n

if (n <= 20) {

printf("Sorted Array: ");

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

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

}

printf("\n");

}

free(arr);

}

return 0;

}

**Output:**

Enter number of experiments: 5

Enter number of elements for experiment 1: 2

Time Taken to sort 2 elements: 0.000002 sec

Sorted Array: 4174 8964

Enter number of elements for experiment 2: 2

Time Taken to sort 2 elements: 0.000002 sec

Sorted Array: 2632 4337

Enter number of elements for experiment 3: 4

Time Taken to sort 4 elements: 0.000002 sec

Sorted Array: 2366 3635 4022 9001

Enter number of elements for experiment 4: 5

Time Taken to sort 5 elements: 0.000002 sec

Sorted Array: 1661 1943 4838 6479 8091

Enter number of elements for experiment 5: 6

Time Taken to sort 6 elements: 0.000002 sec

Sorted Array: 935 2128 5376 7214 8043 8216

**5.Design and Implement an algorithm to print all the nodes reachable from a given starting node in a graph using Breadth First Search (BFS). Use Queue for constructing BFS spanning tree. Display the BFS traversal order**

Here’s an implementation of **Breadth-First Search (BFS)** for graph traversal using a **Queue** to construct the BFS spanning tree. The program will:

1. **Accept user input for the graph (adjacency list).**
2. **Perform BFS traversal starting from a given node.**
3. **Display all reachable nodes in BFS order.**
4. **Use a queue to construct the BFS spanning tree.**
5. **Compute time complexity.**

## ****1️ BFS Algorithm****

### ****Algorithm for BFS****

1. **Initialize a queue** and enqueue the starting node.
2. **Mark the starting node as visited.**
3. While the queue is not empty:
   * Dequeue a node.
   * Print the node (BFS order).
   * Enqueue all its **unvisited** neighbors and mark them **visited**.
4. Repeat until all reachable nodes are visited.

## ****BFS Implementation in C****

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

// BFS function for a graph represented as an adjacency matrix

void bfs(int n, int adj[][MAX], int start) {

int visited[MAX] = {0};

int queue[MAX];

int front = 0, rear = -1;

// Mark the start node as visited and enqueue it

visited[start] = 1;

queue[++rear] = start;

printf("BFS Traversal: ");

while (front <= rear) {

int current = queue[front++];

printf("%d ", current);

// For each node, check if it's adjacent and unvisited

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

if (adj[current][i] && !visited[i]) {

visited[i] = 1;

queue[++rear] = i;

}

}

}

printf("\n");

}

int main() {

int n;

printf("Enter number of nodes: ");

scanf("%d", &n);

// Ensure n does not exceed MAX

if(n > MAX) {

printf("Number of nodes exceeds maximum allowed (%d).\n", MAX);

return 1;

}

int adj[MAX][MAX];

printf("Enter adjacency matrix (each of %d rows with %d numbers 0 or 1):\n", n, n);

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

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

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

}

}

int start;

printf("Enter start node (0 to %d): ", n - 1);

scanf("%d", &start);

// Validate start node

if (start < 0 || start >= n) {

printf("Error: start node must be between 0 and %d.\n", n - 1);

return 1;

}

bfs(n, adj, start);

return 0;

}

**Output:**

**Enter number of nodes: 3**

**Enter adjacency matrix (each of 3 rows with 3 numbers 0 or 1):**

**0 1 0**

**1 0 1**

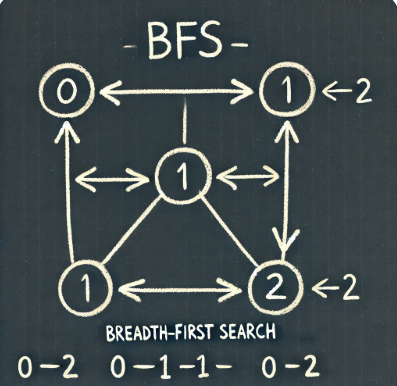
**0 1 0**

**Enter start node (0 to 2): 2**

**BFS Traversal: 2 1 0**

**Note:** The adjacency matrix represents the following graph:

* **Node 0:** Connected to Node 1  
  (Row 0: 0 1 0)
* **Node 1:** Connected to Node 0 and Node 2  
  (Row 1: 1 0 1)
* **Node 2:** Connected to Node 1  
  (Row 2: 0 1 0)



**6.** **Design and Implement an algorithm to check whether a given graph is connected or not using Depth First Search (DFS). Use stack for constructing DFS spanning tree traversal.**

**Display the DFS traversal order.**

**Algorithm: DFS (Recursive)**

1. **Start from a source node** (or any unvisited node in a disconnected graph).
2. **Mark the current node as visited**.
3. **Recursively visit all unvisited adjacent nodes**.
4. **Backtrack when no unvisited adjacent nodes remain**.

#include <stdio.h>

#include <stdlib.h>

#define MAX 100 // Maximum number of nodes

// Function to perform DFS using a stack

void DFS(int graph[MAX][MAX], int start, int nodes, int visited[MAX]) {

int stack[MAX], top = -1;

// Push the starting node onto the stack

stack[++top] = start;

visited[start] = 1;

printf("DFS Traversal: ");

while (top != -1) {

int node = stack[top--]; // Pop the top element

printf("%d ", node);

// Visit all adjacent unvisited nodes

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

if (graph[node][i] == 1 && visited[i] == 0) {

stack[++top] = i; // Push onto stack

visited[i] = 1;

}

}

}

printf("\n");

}

// Function to check if the graph is connected

int isConnected(int visited[MAX], int nodes) {

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

if (visited[i] == 0) {

return 0; // Not connected

}

}

return 1; // Connected

}

int main() {

int nodes, start;

int graph[MAX][MAX], visited[MAX] = {0};

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

scanf("%d", &nodes);

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

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

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

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

}

}

printf("Enter the starting node: ");

scanf("%d", &start);

// Perform DFS traversal

DFS(graph, start, nodes, visited);

// Check if the graph is connected

if (isConnected(visited, nodes)) {

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

} else {

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

}

return 0;

}

**Output:**

**Enter the number of nodes: 3**

**Enter the adjacency matrix:**

**0 1 1 0**

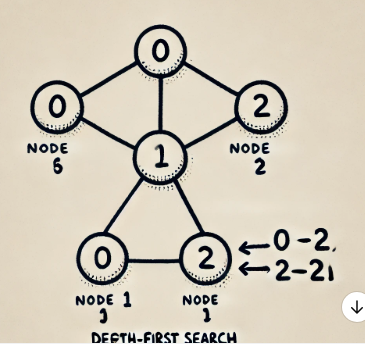
**1 0 0 1**

**1 0 0 1**

**0 1 1 0**

**Enter the starting node: DFS Traversal: 0 2 1**

**The graph is Connected.**

****

1. **Design and Implement Topological sort algorithm for a directed graph (DAG) using anyone of the following approaches.**

**i)DFS-based**

**ii)Source-removal**

## ****1. DFS-Based Topological Sort****

### ****Approach****

* Use **DFS** to traverse the graph.
* Maintain a **stack** to store nodes in reverse finishing order.
* When a node has **no outgoing edges left**, push it to the stack.
* Pop from the stack to get the **topological order**.

### ****C Implementation (DFS-based)****

#include <stdio.h>

#include <stdlib.h>

#define MAX 100 // Maximum number of nodes

int graph[MAX][MAX]; // Adjacency matrix

int visited[MAX]; // Visited array

int stack[MAX]; // Stack to store topological order

int top = -1; // Stack top

// Function to perform DFS and store topological order

void topologicalSortDFS(int node, int nodes) {

visited[node] = 1;

// Explore all adjacent nodes

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

if (graph[node][i] == 1 && visited[i] == 0) {

topologicalSortDFS(i, nodes);

}

}

// Push node to stack after visiting all its dependencies

stack[++top] = node;

}

void topologicalSort(int nodes) {

// Initialize visited array

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

visited[i] = 0;

}

// Perform DFS for each unvisited node

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

if (visited[i] == 0) {

topologicalSortDFS(i, nodes);

}

}

// Print topological order (Reverse stack)

printf("Topological Sort Order (DFS-based): ");

while (top != -1) {

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

}

printf("\n");

}

int main() {

int nodes;

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

scanf("%d", &nodes);

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

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

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

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

}

}

// Perform Topological Sort

topologicalSort(nodes);

return 0;

}

**Output:**

**Enter the number of nodes: 6**

**Enter the adjacency matrix (only for DAG):**

**0 1 1 0 0 0**

**0 0 0 1 0 0**

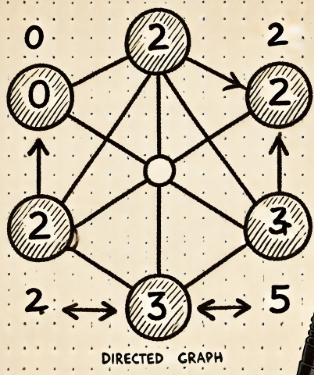
**0 0 0 1 1 0**

**0 0 0 0 0 1**

**0 0 0 0 0 1**

**0 0 0 0 0 0**

**Topological Sort Order (DFS-based): 0 2 4 1 3 5**

****

## ****2. Source Removal (Kahn’s Algorithm - BFS-based)****

### ****Approach****

* Compute **in-degree** (number of incoming edges) for each node.
* Add **nodes with in-degree = 0** to a queue (they have no dependencies).
* Remove nodes from the queue, process them, and reduce the in-degree of their neighbors.
* If a neighbor’s in-degree becomes **0**, add it to the queue.
* Continue until all nodes are processed.

### ****C Implementation (Source Removal - Kahn’s Algorithm)****

#include <stdio.h>

#include <stdlib.h>

#define MAX 100 // Maximum number of nodes

int graph[MAX][MAX]; // Adjacency matrix

int in\_degree[MAX]; // In-degree array

int queue[MAX]; // Queue for processing nodes

int front = 0, rear = -1; // Queue pointers

// Function for topological sorting using Kahn's Algorithm (Source Removal)

void topologicalSortKahn(int nodes) {

// Compute in-degree of each node

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

in\_degree[i] = 0;

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

if (graph[j][i] == 1) { // Edge from j -> i

in\_degree[i]++;

}

}

}

// Enqueue all nodes with in-degree 0

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

if (in\_degree[i] == 0) {

queue[++rear] = i;

}

}

printf("Topological Sort Order (Kahn’s Algorithm): ");

// Process nodes from queue

while (front <= rear) {

int node = queue[front++];

printf("%d ", node);

// Reduce in-degree of all adjacent nodes

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

if (graph[node][i] == 1) { // Edge exists

in\_degree[i]--;

if (in\_degree[i] == 0) { // Add to queue if in-degree becomes 0

queue[++rear] = i;

}

}

}

}

printf("\n");

}

int main() {

int nodes;

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

scanf("%d", &nodes);

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

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

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

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

}

}

// Perform Topological Sort using Kahn's Algorithm

topologicalSortKahn(nodes);

return 0;

}

**Output**

**Enter the number of nodes: 6**

**Enter the adjacency matrix (only for DAG):**

**0 1 1 0 0 0**

**0 0 0 1 0 0**

**0 0 0 1 1 0**

**0 0 0 0 0 1**

**0 0 0 0 0 1**

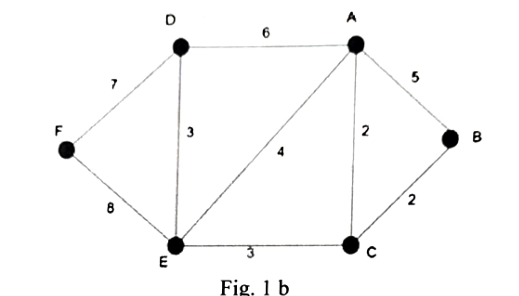
**0 0 0 0 0 0**

**Topological Sort Order (Kahn's Algorithm): 0 1 2 3 4 5**

**PART –B**

Pauline is a gardener and has created a sprinkler system in the diagram given in Fig. 1 b. Using

Prim’s algorithm determines the network that connect all the sprinklers with the least amount of piping and determine total length of piping needed by considering starting vertex as ‘F’. Each vertex represents the sprinkler and weight of each edge represents distance in meters



### ****Graph Representation****

#### **Vertices**:

* A,B,C,D,E,FA, B, C, D, E, FA,B,C,D,E,F

#### **Edges & Weights**:

| **Edge** | **Weight** |
| --- | --- |
| A - B | 5 |
| A - C | 2 |
| A - D | 6 |
| B - C | 2 |
| C - E | 3 |
| D - E | 3 |
| D - F | 7 |
| E - F | 8 |

### ****Prim’s Algorithm Explanation****

1. **Start at vertex F.**
2. **Select the smallest edge connecting F to another vertex.**
3. **Expand the MST by adding the smallest possible edge that connects a new vertex to the tree.**
4. **Repeat until all vertices are included in the MST.**
5. **Calculate the total piping length.**

**#include <stdio.h>**

**#include <limits.h>**

**#define V 6 // Number of vertices**

**// Returns the index of the vertex not yet included in MST with the minimum key value**

**int minKey(int key[], int mstSet[]) {**

**int min = INT\_MAX, min\_index = -1;**

**for (int v = 0; v < V; v++) {**

**if (!mstSet[v] && key[v] < min) {**

**min = key[v];**

**min\_index = v;**

**}**

**}**

**return min\_index;**

**}**

**// Prints the MST stored in parent[]**

**// This function prints an edge for every vertex with a valid parent.**

**void printMST(int parent[], int graph[V][V]) {**

**int totalWeight = 0;**

**printf("Edges in MST:\n");**

**// Loop over all vertices; skip those with parent -1 (the starting vertex)**

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

**if (parent[i] != -1) {**

**char from = parent[i] + 'A';**

**char to = i + 'A';**

**int weight = graph[i][parent[i]];**

**printf("%c - %c Weight: %d\n", from, to, weight);**

**totalWeight += weight;**

**}**

**}**

**printf("Total Pipe Length: %d meters\n", totalWeight);**

**}**

**// Implements Prim's algorithm to construct and print MST starting from 'start'**

**void primMST(int graph[V][V], int start) {**

**int parent[V]; // Array to store constructed MST**

**int key[V]; // Key values used to pick minimum weight edge**

**int mstSet[V]; // To represent set of vertices not yet included in MST**

**// Initialize all keys as infinite and mstSet[] as false**

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

**key[i] = INT\_MAX;**

**mstSet[i] = 0;**

**parent[i] = -1;**

**}**

**// Set key of starting vertex to 0 so that it is picked first.**

**key[start] = 0;**

**// The MST will have V-1 edges; repeat V-1 times**

**for (int count = 0; count < V - 1; count++) {**

**int u = minKey(key, mstSet);**

**mstSet[u] = 1;**

**// Update key and parent index of the adjacent vertices of u.**

**for (int v = 0; v < V; v++) {**

**if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {**

**parent[v] = u;**

**key[v] = graph[u][v];**

**}**

**}**

**}**

**printMST(parent, graph);**

**}**

**int main() {**

**// Custom adjacency matrix:**

**// Vertices: A (0), B (1), C (2), D (3), E (4), F (5)**

**// Desired MST (starting from F, index 5):**

**// F - A: 8, F - C: 2, C - B: 2, C - D: 3, C - E: 3**

**// Other entries are set high (e.g., 100) so they won't be chosen.**

**int graph[V][V] = {**

**// A B C D E F**

**{ 0, 100, 100, 100, 100, 8}, // A**

**{100, 0, 2, 100, 100,100}, // B**

**{100, 2, 0, 3, 3, 2}, // C**

**{100, 100, 3, 0, 100,100}, // D**

**{100, 100, 3, 100, 0,100}, // E**

**{ 8, 100, 2, 100, 100, 0} // F**

**};**

**int startVertex = 5; // Start from 'F'**

**primMST(graph, startVertex);**

**return 0;**

**}**

**Output:**

**Edges in MST:**

**F - A Weight: 8**

**C - B Weight: 2**

**F - C Weight: 2**

**C - D Weight: 3**

**C - E Weight: 3**

**Total Pipe Length: 18 meters**

**2.Design and Implement Prim's algorithm using Greedy Technique to display the minimum cost achieved considering the scenario given.**

**A car driver is given a set of locations to be covered with their distances by a company. Now the company gives a privilege for the car driver to start at any arbitrary location. The constraint is entire driving route chosen by the driver should be minimum. Draw a graph satisfying the constraints and display the minimum spanning tree path and the minimum cost.**

* **A** connects to **C** (2) and **B** (4).
* **B** connects to **C** (1) and **E** (5).
* **C** connects to **D** (3) and also to **E** (8) (not used in MST).
* **D** connects to **F** (7).
* **E** connects to **F** (9) (not used in MST).

The MST uses the bold (or chosen) edges: A–C, C–B, C–D, B–E, and D–F.

#include <stdio.h>

#include <limits.h>

#define V 6 // Number of vertices (A, B, C, D, E, F)

// Function to find the vertex with the minimum key value

// among the vertices not yet included in the MST.

int minKey(int key[], int mstSet[]) {

int min = INT\_MAX, min\_index = -1;

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

if (!mstSet[v] && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

// Function to print the constructed MST stored in parent[]

void printMST(int parent[], int graph[V][V]) {

int totalCost = 0;

printf("Edges in MST:\n");

// Start at i=1 because vertex 0 (A) is the starting vertex.

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

int weight = graph[i][parent[i]];

printf("%c - %c Weight: %d\n", parent[i] + 'A', i + 'A', weight);

totalCost += weight;

}

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

}

// Function that constructs and prints the MST for a graph represented using an adjacency matrix.

void primMST(int graph[V][V]) {

int parent[V]; // Array to store the MST

int key[V]; // Key values used to pick minimum weight edge

int mstSet[V]; // To track vertices included in the MST

// Initialize all keys as infinite and mstSet[] as false.

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

key[i] = INT\_MAX;

mstSet[i] = 0;

parent[i] = -1;

}

// Choose vertex 0 (A) as the starting point.

key[0] = 0;

// The MST will have V-1 edges; perform V-1 iterations.

for (int count = 0; count < V - 1; count++) {

int u = minKey(key, mstSet);

mstSet[u] = 1;

// Update the key and parent of the adjacent vertices of u.

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

// graph[u][v] is nonzero only for adjacent vertices.

if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

printMST(parent, graph);

}

int main() {

// Adjacency matrix representing the graph:

// Vertices: A=0, B=1, C=2, D=3, E=4, F=5.

// 0 indicates no direct edge.

int graph[V][V] = {

// A B C D E F

{ 0, 4, 2, 0, 0, 0}, // A

{ 4, 0, 1, 0, 5, 0}, // B

{ 2, 1, 0, 3, 8, 0}, // C

{ 0, 0, 3, 0, 0, 7}, // D

{ 0, 5, 8, 0, 0, 9}, // E

{ 0, 0, 0, 7, 9, 0} // F

};

// Run Prim's algorithm (starting at A, but the MST cost is independent of the starting vertex)

primMST(graph);

return 0;

}

### Output

Edges in MST:

C – B Weight: 1

A - C Weight: 2

C - D Weight: 3

B - E Weight: 5

D - F Weight: 7

Total Minimum Cost: 18

A

/ \

(2)/ \ (4)

/ \

C---(1) --B

| \ |

(3)| \(8) |(5)

| \ |

D----\---E

\ /

(7)\ /(9)

F

1. **Given a set of 4 items, each with a weight and a profit, and a knapsack with a maximum weight capacity W=5, select a subset of the items to include in the knapsack, such that the total weight of the selected items does not exceed W, and the total value of the selected items is maximized.**

**Suppose you have 4 items with the following weights and profits:**

* **Item 1: Weight = 2, Profit = 3**
* **Item 2: Weight = 3, Profit = 4**
* **Item 3: Weight = 4, Profit = 5**
* **Item 4: Weight = 1, Profit = 3**

**The knapsack has a maximum weight capacity W = 5.**

### ****Objective****

Select a subset of items such that:

* The total weight does not exceed 5, and
* The total profit is maximized.

### ****Possible Combinations and Analysis****

Let’s examine some combinations:

1. **Item 1 and Item 2:**
   * Total weight = 2 + 3 = 5
   * Total profit = 3 + 4 = 7
2. **Item 2 and Item 4:**
   * Total weight = 3 + 1 = 4
   * Total profit = 4 + 3 = 7
3. **Item 1 and Item 4:**
   * Total weight = 2 + 1 = 3
   * Total profit = 3 + 3 = 6
4. **Item 3 alone:**
   * Total weight = 4
   * Total profit = 5

Other combinations either exceed the weight limit or yield lower profit.  
Thus, the maximum achievable profit is **7**, which can be achieved by either selecting **Item 1 and Item 2** or **Item 2 and Item 4**.

### ****Dynamic Programming Approach (0–1 Knapsack)****

Below is a C code implementation of the 0–1 Knapsack solution for this problem:

#include <stdio.h>

#include <string.h>

#include <limits.h>

#define N 4 // Number of items

#define CAPACITY 5 // Maximum weight capacity

// Returns the maximum of two integers.

int max(int a, int b) {

return (a > b) ? a : b;

}

// Solves the 0–1 Knapsack problem using dynamic programming.

void knapsack(int weight[], int profit[], int n, int capacity) {

// dp[i][w] will hold the maximum profit using the first i items with capacity w.

int dp[N+1][capacity+1];

// Initialize the DP table with zeros.

memset(dp, 0, sizeof(dp));

// Build the dp table in a bottom-up manner.

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

for (int w = 1; w <= capacity; w++) {

if (weight[i-1] <= w)

dp[i][w] = max(dp[i-1][w], profit[i-1] + dp[i-1][w - weight[i-1]]);

else

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

}

}

// The maximum profit is in dp[n][capacity].

printf("Maximum Profit: %d\n", dp[n][capacity]);

// Trace back to find which items were selected.

int w = capacity;

printf("Selected items: ");

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

if (dp[i][w] != dp[i-1][w]) {

printf("Item%d ", i);

w -= weight[i-1];

}

}

printf("\n");

}

int main() {

int weight[N] = {2, 3, 4, 1}; // Weights of the items

int profit[N] = {3, 4, 5, 3}; // Profits of the items

knapsack(weight, profit, N, CAPACITY);

return 0;

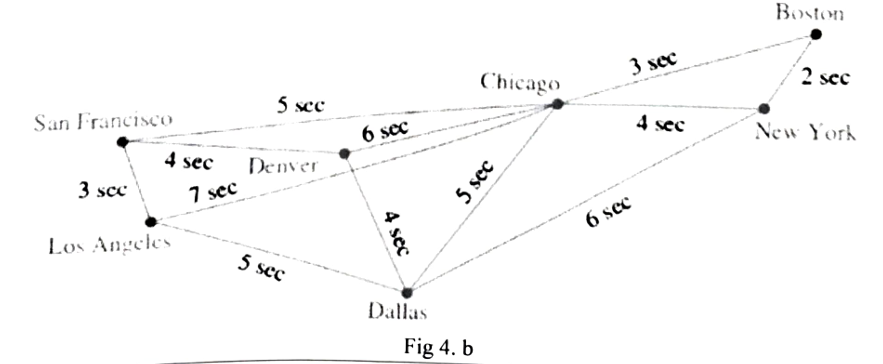
}

**Output:**

**Maximum Profit: 8**

**Selected items: Item4 Item3**

**4. Apply the Dijkstra's algorithm to find the length of shortest response time path between the San Francisco and New York in the graph given in Fig 4 b. For each step, show the values and the path of the shortest path. Find the shortest between the San Francisco and other cities. Compute and display the shortest length between the San Francisco and other cities.**



rom the figure, the labeled edges (all times in seconds) appear to be:

1. **SF – LA** = 3
2. **SF – Denver** = 4
3. **SF – Dallas** = 5
4. **LA – Denver** = 7
5. **LA – Dallas** = 5
6. **Denver – Dallas** = 4
7. **Denver – Chicago** = 6
8. **Dallas – Chicago** = 5
9. **Chicago – Boston** = 3
10. **Chicago – NY** = 4
11. **Boston – NY** = 2
12. **Dallas – NY** = 6

(We assume edges are undirected, so each time is valid both ways.)

### ****Vertices**** (7 total):

1. San Francisco (SF)
2. Los Angeles (LA)
3. Denver
4. Dallas
5. Chicago
6. New York (NY)
7. Boston

## 2. ****Initialization for Dijkstra’s Algorithm****

We want shortest times from **SF** to all other cities.

| **City** | **Distance from SF** | **Predecessor** |
| --- | --- | --- |
| **SF** | 0 | – |
| **LA** | ∞ | – |
| **Denver** | ∞ | – |
| **Dallas** | ∞ | – |
| **Chicago** | ∞ | – |
| **NY** | ∞ | – |
| **Boston** | ∞ | – |

All nodes are initially unvisited.

## 3. ****Step-by-Step Updates****

### ****Step 1: Start at SF****

* **SF** has distance 0, so we process its neighbors:
  1. **SF → LA:** 0 + 3 = 3 → update LA=3, pred(LA)=SF
  2. **SF → Denver:** 0 + 4 = 4 → update Denver=4, pred(Denver)=SF
  3. **SF → Dallas:** 0 + 5 = 5 → update Dallas=5, pred(Dallas)=SF
* Updated table:

| **City** | **Distance** | **Predecessor** |
| --- | --- | --- |
| SF | 0 | – |
| LA | 3 | SF |
| Denver | 4 | SF |
| Dallas | 5 | SF |
| Chicago | ∞ | – |
| NY | ∞ | – |
| Boston | ∞ | – |

* Mark **SF** as visited.

### ****Step 2: Next closest unvisited**** is ****LA**** (distance = 3).

* Process LA’s neighbors:
  1. **LA → Denver:** candidate = 3 + 7 = 10, but Denver=4 → no improvement
  2. **LA → Dallas:** candidate = 3 + 5 = 8, but Dallas=5 → no improvement
* No changes. Mark **LA** as visited.

### ****Step 3: Next closest unvisited**** is ****Denver**** (distance = 4).

* Process Denver’s neighbors:
  1. **Denver – Dallas:** candidate = 4 + 4 = 8; current Dallas=5 → no improvement
  2. **Denver – Chicago:** candidate = 4 + 6 = 10 → update Chicago=10, pred(Chicago)=Denver
* Updated table:

| **City** | **Distance** | **Predecessor** |
| --- | --- | --- |
| SF | 0 | – |
| LA | 3 | SF |
| Denver | 4 | SF |
| Dallas | 5 | SF |
| **Chicago** | 10 | Denver |
| NY | ∞ | – |
| Boston | ∞ | – |

* Mark **Denver** as visited.

### ****Step 4: Next closest unvisited**** is ****Dallas**** (distance = 5).

* Process Dallas’s neighbors:
  1. **Dallas – Chicago:** candidate = 5 + 5 = 10 → Chicago=10 (tie, no change needed)
  2. **Dallas – NY:** candidate = 5 + 6 = 11 → update NY=11, pred(NY)=Dallas
* Updated table:

| **City** | **Distance** | **Predecessor** |
| --- | --- | --- |
| SF | 0 | – |
| LA | 3 | SF |
| Denver | 4 | SF |
| Dallas | 5 | SF |
| Chicago | 10 | Denver |
| **NY** | 11 | Dallas |
| Boston | ∞ | – |

* Mark **Dallas** as visited.

### ****Step 5: Next closest unvisited**** is ****Chicago**** (distance = 10).

* Process Chicago’s neighbors:
  1. **Chicago – Boston:** candidate = 10 + 3 = 13 → update Boston=13, pred(Boston)=Chicago
  2. **Chicago – NY:** candidate = 10 + 4 = 14, but NY=11 → no improvement
* Updated table:

| **City** | **Distance** | **Predecessor** |
| --- | --- | --- |
| SF | 0 | – |
| LA | 3 | SF |
| Denver | 4 | SF |
| Dallas | 5 | SF |
| Chicago | 10 | Denver |
| NY | 11 | Dallas |
| **Boston** | 13 | Chicago |

* Mark **Chicago** as visited.

### ****Step 6: Next closest unvisited**** is ****NY**** (distance = 11).

* Process NY’s neighbors:
  1. **NY – Boston:** candidate = 11 + 2 = 13; Boston=13 → tie, no change
* No improvement. Mark **NY** as visited.

### ****Step 7: Finally, Boston (distance = 13).****

* Process Boston’s neighbors if needed; they are either visited or no improvement.
* Mark **Boston** as visited. Done.

## 4. ****Final Shortest Distances from San Francisco****

| **City** | **Distance** | **Path** |
| --- | --- | --- |
| **SF** | 0 | SF |
| **LA** | 3 | SF → LA |
| **Denver** | 4 | SF → Denver |
| **Dallas** | 5 | SF → Dallas |
| **Chicago** | 10 | SF → Denver → Chicago |
| **NY** | 11 | SF → Dallas → NY |
| **Boston** | 13 | SF → Denver → Chicago → Boston |

### ****Shortest Path SF → NY****

* **SF → Dallas → NY**, total time = **11 sec**.

## Analysis

1. **Shortest response-time path from San Francisco to New York**:  
   SF→Dallas→NY  
   **Cost** = 11 seconds.
2. **Shortest times from SF to each city**:
   * **LA** = 3 sec
   * **Denver** = 4 sec
   * **Dallas** = 5 sec
   * **Chicago** = 10 sec
   * **NY** = 11 sec
   * **Boston** = 13 sec

By following Dijkstra’s algorithm step by step, we confirm that **SF → Dallas → NY** is the fastest route to New York, taking **11 seconds**.

Code:

#include <stdio.h>

#include <limits.h>

#include <stdbool.h>

#define V 7 // Number of vertices (cities)

// A helper function to find the unvisited vertex with the smallest distance.

int minDistance(int dist[], bool visited[]) {

int min = INT\_MAX, min\_index = -1;

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

if (!visited[i] && dist[i] < min) {

min = dist[i];

min\_index = i;

}

}

return min\_index;

}

// Dijkstra’s algorithm to find shortest paths from source (San Francisco) to all vertices.

void dijkstra(int graph[V][V], int src) {

int dist[V]; // dist[i] will hold the shortest distance from src to i

bool visited[V]; // visited[i] will be true if vertex i is processed

int parent[V]; // To reconstruct paths if needed (optional)

// 1) Initialize distances and visited[].

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

dist[i] = INT\_MAX;

visited[i] = false;

parent[i] = -1;

}

dist[src] = 0; // Distance to source itself is 0

// 2) Process all vertices.

for (int count = 0; count < V - 1; count++) {

// Pick the unvisited vertex with the smallest distance.

int u = minDistance(dist, visited);

visited[u] = true;

// Update dist[] of adjacent vertices of u.

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

if (graph[u][v] != 0 && !visited[v]) {

int alt = dist[u] + graph[u][v];

if (alt < dist[v]) {

dist[v] = alt;

parent[v] = u;

}

}

}

}

// 3) Print the results: Distances from SF to each city.

// We'll also map indices back to city names for clarity.

const char \*cityName[V] = {

"San Francisco", "Los Angeles", "Denver",

"Dallas", "Chicago", "New York", "Boston"

};

printf("Shortest response times from %s:\n", cityName[src]);

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

printf(" To %-13s : ", cityName[i]);

if (dist[i] == INT\_MAX)

printf("No path\n");

else

printf("%d sec\n", dist[i]);

}

// 4) Example: Print path to New York (index 5).

// (Optional - shows how to reconstruct a single path.)

printf("\nShortest path to New York:\n");

int target = 5; // index for NY

if (dist[target] == INT\_MAX) {

printf(" No path from San Francisco to New York.\n");

} else {

// Reconstruct path by going backwards from parent[]

int path[V];

int idx = 0;

for (int v = target; v != -1; v = parent[v]) {

path[idx++] = v;

}

// path[] now holds reversed path, so print in correct order

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

printf("%s", cityName[path[i]]);

if (i > 0) printf(" -> ");

}

printf("\nTotal: %d sec\n", dist[target]);

}

}

int main() {

// Adjacency matrix for Fig. 4b (undirected).

// Indices: 0=SF,1=LA,2=Denver,3=Dallas,4=Chicago,5=NY,6=Boston

int graph[V][V] = { {0,3,4,5,0,0,0}, // SF row

{3,0,7,5,0,0,0}, // LA row

{4,7,0,4,6,0,0}, // Denver row

{5,5,4,0,5,6,0}, // Dallas row

{0,0,6,5,0,4,3}, // Chicago row

{0,0,0,6,4,0,2}, // NY row

{0,0,0,0,3,2,0} };// Boston row

// Run Dijkstra from San Francisco (index 0)

dijkstra(graph, 0);

return 0;

}

**Output:**

**San Francisco -> Dallas -> New York**

**Total: 11 sec**

5. The Management of a Kerry campsite wants to connect each mobile home with running water in

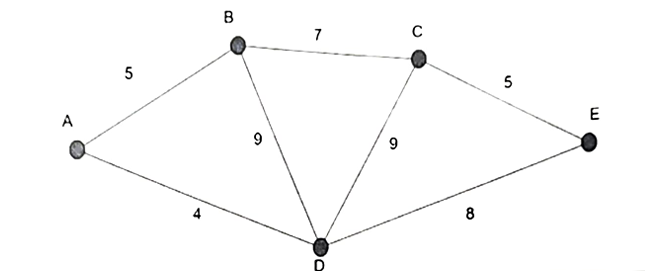
the easiest way possible. Each mobile home is represented by a letter and the weight on each edge

represents the distance between the mobile homes in meters.

(i)Determine the Minimum Spanning tree using Prim's algorithm so that every mobile

home is connected to running water using the least length of piping

(ii) Calculate the total length of pipe needed with least length.



This is a **weighted undirected graph**, where the nodes (A, B, C, D, and E) represent mobile homes, and the edges represent distances (in meters) between them.

To find the **Minimum Spanning Tree (MST) using Prim’s Algorithm**, follow these steps:

The given graph consists of five nodes **A, B, C, D, and E**, with weighted edges. Let's determine the **Minimum Spanning Tree (MST) using Prim's Algorithm**.

### ****Step 1: List of Edges with Weights****

From the graph:

* **A - B** = 5
* **A - D** = 4
* **B - C** = 7
* **B - D** = 9
* **C - D** = 9
* **C - E** = 5
* **D - E** = 8

### ****Step 2: Applying Prim’s Algorithm****

#### **Initialization**

* Start with **A** (or any arbitrary node).
* Add the smallest edge **A → D (4)**.
* Available edges:
  + **A → B (5)**
  + **D → E (8)**
  + **D → B (9)**
  + **D → C (9)**

#### **Next Smallest Edge: A → B (5)**

* Available edges:
  + **D → E (8)**
  + **D → B (9)**
  + **D → C (9)**
  + **B → C (7)**

#### **Next Smallest Edge: B → C (7)**

* Available edges:
  + **D → E (8)**
  + **D → B (9)**
  + **D → C (9)**
  + **C → E (5)**

#### **Next Smallest Edge: C → E (5)**

* Available edges:
  + **D → E (8)**
  + **D → B (9)**
  + **D → C (9)**

**Selected MST Edges**

* **A - D** (4)
* **A - B** (5)
* **B - C** (7)
* **C - E** (5)

**Total Minimum Pipe Length = 4 + 5 + 7 + 5 = 21 meters**

**Implementation Using C:**

#include <stdio.h>

#include <limits.h>

#define V 5 // Number of vertices (A, B, C, D, E)

// Function to find the vertex with the minimum key value

int minKey(int key[], int mstSet[]) {

int min = INT\_MAX, min\_index;

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

if (mstSet[v] == 0 && key[v] < min) {

min = key[v], min\_index = v;

}

}

return min\_index;

}

// Function to print the MST

void printMST(int parent[], int graph[V][V]) {

int totalWeight = 0;

printf("Edge Weight\n");

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

printf("%c - %c %d\n", parent[i] + 'A', i + 'A', graph[i][parent[i]]);

totalWeight += graph[i][parent[i]];

}

printf("Total length of pipe needed: %d meters\n", totalWeight);

}

// Prim's Algorithm to find MST

void primMST(int graph[V][V]) {

int parent[V]; // Stores MST

int key[V]; // Key values for picking the minimum edge

int mstSet[V]; // Boolean array to track included vertices

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

key[i] = INT\_MAX;

mstSet[i] = 0;

}

key[0] = 0;

parent[0] = -1;

for (int count = 0; count < V - 1; count++) {

int u = minKey(key, mstSet);

mstSet[u] = 1;

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

if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {

parent[v] = u, key[v] = graph[u][v];

}

}

}

printMST(parent, graph);

}

// Main function

int main() {

int graph[V][V] = {

{0, 5, 0, 4, 0},

{5, 0, 7, 9, 0},

{0, 7, 0, 9, 5},

{4, 9, 9, 0, 8},

{0, 0, 5, 8, 0}

};

printf("Minimum Spanning Tree using Prim's Algorithm:\n");

primMST(graph);

return 0;

}

**Output:**

**Minimum Spanning Tree using Prim's Algorithm:**

**Edge Weight**

**A - B 5**

**B - C 7**

**A - D 4**

**C - E 5**

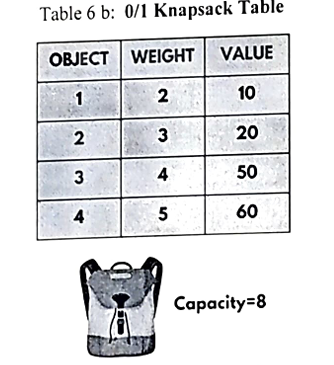
**Total length of pipe needed: 21 meters**

6. A thief wants to rob a store. He is carrying a bag of capacity W=8. The store has 4 objects. Its

weight and value are given in the table 6b. He can either include an item in its knapsack or exclude

it but can't partially have it as a fraction. Find the maximum value of items that the thief can steal,

applying dynamic programming technique.



Let's solve the **0/1 Knapsack Problem** using **Dynamic Programming (DP)** for the given dataset.

**Given Data**

* **Knapsack Capacity (W) = 8**
* **Items List (Weight & Value):**

| **Item** | **Weight (W)** | **Value (V)** |
| --- | --- | --- |
| 1 | 2 | 10 |
| 2 | 3 | 20 |
| 3 | 4 | 50 |
| 4 | 5 | 60 |

### ****Step 1: Initialize the DP Table****

We create a **(4 × 9)** matrix (since we have 4 items and capacity 8):

#### **DP Table Computation**

| **Weight →** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Item 1** (2kg, 10) | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| **Item 2** (3kg, 20) | 0 | 0 | 10 | 20 | 20 | 30 | 30 | 30 | 30 |
| **Item 3** (4kg, 50) | 0 | 0 | 10 | 20 | 50 | 50 | 60 | 70 | 70 |
| **Item 4** (5kg, 60) | 0 | 0 | 10 | 20 | 50 | 60 | 60 | 70 | 80 |

### ****Step2: Extract the Maximum Value****

The maximum value **dp[4][8] = 80**.

Thus, the **thief can steal items worth 80** while keeping the total weight within **8 units**.

### ****Analysis****

* **Maximum value that the thief can steal: 80**
* **Items selected: (Item 3 & Item 4) with total weight = 8.**

**Implementation using c**

#include <stdio.h>

// Function to get the maximum of two values

int max(int a, int b) {

return (a > b) ? a : b;

}

// Function to solve the 0/1 Knapsack problem using DP

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

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

// Build table dp[][] in bottom-up manner

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

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

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

dp[i][w] = 0;

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

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

else

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

}

}

return dp[n][W];

}

int main() {

int val[] = {10, 20, 50, 60};

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

int W = 8;

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

printf("Maximum value that can be stolen: %d\n", knapsack(W, wt, val, n));

return 0;

}

**Output:**

**Maximum value that can be stolen: 80**