

Ramakrishna Mission Vidyamandira

(An Autonomous College Under University of Calcutta)

Computer Science (Honors) Semester IV 2021

Paper: 4CMSMJC1 Practical

|  |
| --- |
| **Submitted by** |
| Class Roll Number: 340  B.Sc.  4th Semester  Batch: 2023-27 |

**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SI NO.** | **ASSIGNMNET STATEMENT** | **D-O-A** | **D-O-S** | **SIGNATURE** |
| **1.a** | **Write a c program to implement quick sort using divide and conquer (taking last element as pivot).** | **1/2/2025** |  |  |
| **1.b** | **Write a c program to implement quick sort using divide and conquer (taking first element as pivot).** | **1/2/2025** |  |  |
| **2.a** | **Write a c program to implement merge sort using divide and conquer.** | **1/2/2025** |  |  |
| **2.b** | **Write a c program to implement merge sort using iterative way.** | **1/2/2025** |  |  |
| **3.a** | **Write a c program to implement Strassen's Matrix Multiplication algorithm for square matrices using divide and conquer.** | **19/2/2025** |  |  |
| **3.b** | **Modify and implement Strassen's Matrix Multiplication algorithm so that it works with non-square matrices as well.** | **19/2/2025** |  |  |
| **4.a** | **Write a c program to implement matrix chain multiplication problem using dynamic programming.** | **4/3/2025** |  |  |
| **4.b** | **MCM problem using top-down approach of dynamic programming (memorization).** | **4/3/2025** |  |  |
| **5** | **Write a c program to implement fractional knapsack problem using greedy method.** | **8/3/2025** |  |  |
| **6** | **Write a c program to implement 0-1 knapsack problem using dynamic programming.** | **8/3/2025** |  |  |
| **7** | **Write a c program to implement nqueens problem using backtracking.** | **12/3/2025** |  |  |
| **8.a** | **Write a c program to implement breadth first search using adjacency matrix representation.** | **10/3/2025** |  |  |
| **8.b** | **Write a c program to implement breadth first search using adjacency list representation.** | **10/3/2025** |  |  |
| **9.a** | **Write a c program of depth first search using adjacency matrix.** | **11/3/2025** |  |  |
| **9.b** | **Write a c program of depth first search using list representation.** | **11/3/2025** |  |  |
| **10** | **Write a c program to implement krushkal algorithm for a graph.** | **22/3/2025** |  |  |
| **11** | **Write a c program to implement prims algorithm for a graph.** | **22/3/2025** |  |  |
| **12** | **WAP to implement Dijkstra algorithm to implement single source shortest path problem.** | **25/3/2025** |  |  |
| **13** | **WAP to implement bell-man ford algorithm to implement single source shortest path problem.** | **26/3/2025** |  |  |
| **14** | **WAP to implement floyd-warshall algorithm to implement single source shortest path problem.** | **26/3/2025** |  |  |

**1.a) Write a c program to implement quick sort using divide and conquer (taking last element as pivot).**

**Algorithm:**

**QUICKSORT (A, p, r)**

1. **If p < r:**
2. **Set pivot = A[r] (Choose the last element as the pivot).**
3. **Set index = p−1 (Pointer for the smaller element).**
4. **For i = p to r − 1:**
5. **If A [ i ] ≤ pivot:**
6. **Increment index by 1.**
7. **Swap A [ i ] with A[index].**
8. **Increment index by 1.**
9. **Swap A[index] with A[r] (Place pivot in correct position).**
10. **QUICKSORT (A, p, index - 1).**
11. **QUICKSORT (A, index + 1, r).**

**Source Code:**

**#include <stdio.h>**

**int patition(int arr[], int low, int high)**

**{**

**int pivot = arr[high];**

**int i = low - 1;**

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

**{**

**int temp = 0;**

**if (arr[j] < pivot)**

**{**

**i++;**

**temp = arr[i];**

**arr[i] = arr[j];**

**arr[j] = temp;**

**}**

**}**

**i++;**

**int temp = arr[i];**

**arr[i] = pivot;**

**arr[high] = temp;**

**return i;**

**}**

**void quick\_sort(int arr[], int low, int high)**

**{**

**if (low < high)**

**{**

**int pivot\_index = patition(arr, low, high);**

**quick\_sort(arr, low, pivot\_index - 1);**

**quick\_sort(arr, pivot\_index + 1, high);**

**}**

**}**

**int main()**

**{**

**int n;**

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

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

**int arr[n];**

**printf("Enter the unsorted array: ");**

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

**{**

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

**}**

**quick\_sort(arr, 0, n - 1);**

**printf("The sorted array is: ");**

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

**{**

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

**}**

**}**

**Output:**

**Type-1**

**Enter the size of array: 4**

**Enter the unsorted array: 8 3 2 9**

**The sorted array is: 2 3 8 9**

**Type-2**

**Enter the size of array: 5**

**Enter the unsorted array: 9 3 12 6 34**

**The sorted array is: 3 6 9 12 34**

**1.b) Write a c program to implement quick sort using divide and conquer (taking first element as pivot).**

**Algorithm:**

**QUICKSORT (A, p, r)**

1. **If p < r:**
2. **Set pivot = A [p] (Choose the first element as pivot).**
3. **Set index = p (Pointer for the smaller element).**
4. **For i = p + 1 to r:**
5. **If A [i] ≤ pivot:**
6. **Increment index by 1.**
7. **Exchange A [i] with A [index]:**
8. **Swap A[p] with A[index] (Place pivot in correct position).**
9. **QUICKSORT (A, p, index - 1).**
10. **QUICKSORT (A, index + 1, r).**

**Source Code:**

**#include <stdio.h>**

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

**{**

**int pivot = arr[low];**

**int i = low + 1;**

**for (int j = low + 1; j <= high; j++)**

**{**

**if (arr[j] < pivot)**

**{**

**int temp = arr[i];**

**arr[i] = arr[j];**

**arr[j] = temp;**

**i++;**

**}**

**}**

**int temp = arr[low];**

**arr[low] = arr[i - 1];**

**arr[i - 1] = temp;**

**return i - 1;**

**}**

**void quick\_sort(int arr[], int low, int high)**

**{**

**if (low < high)**

**{**

**int pivot\_index = partition(arr, low, high);**

**quick\_sort(arr, low, pivot\_index - 1);**

**quick\_sort(arr, pivot\_index + 1, high);**

**}**

**}**

**int main()**

**{**

**int n;**

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

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

**int arr[n];**

**printf("Enter the unsorted array: ");**

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

**{**

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

**}**

**quick\_sort(arr, 0, n - 1);**

**printf("The sorted array is: ");**

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

**{**

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

**}**

**return 0;**

**}**

**Output:**

**Type-1**

**Enter the size of array: 4**

**Enter the unsorted array: 5 1 6 3**

**The sorted array is: 1 3 5 6**

**Type-2**

**Enter the size of array: 6**

**Enter the unsorted array: 65 23 1 8 56 78**

**The sorted array is: 1 8 23 56 65 78**

**2.a) Write a c program to implement merge sort using divide and conquer.**

**Algorithm:**

**MERGE (A, p, q, r)**

1. **Set n1 = q - p + 1 (Size of left subarray).**
2. **Set n2 = r - q (Size of right subarray).**
3. **Create two temporary arrays L [1...n1] and R [1...n2].**
4. **For i=0 to n1−1:**
5. **Copy A [p + i] to L[i].**
6. **For j=0 to n2−1:**
7. **Copy A [q + 1 + j] to R[j].**
8. **Initialize i = 0, j = 0, k = p.**
9. **While i<n1 and j<n2:**
10. **If L[i]≤R[j]:**
11. **A[k] = L[i], increment i.**
12. **Else:**
13. **A[k] = R[j], increment j.**
14. **Increment k.**
15. **While i<n1:**
16. **Copy remaining elements A[k] = L[i], increment i, k.**
17. **While j<n2:**
18. **Copy remaining elements A[k] = R[j], increment j, k.**

**MERGE-SORT (A, p, r)**

1. **If p<r:**
2. **Compute q = p + (r - p) / 2 (Middle index).**
3. **MERGE-SORT (A, p, q).**
4. **MERGE-SORT (A, q + 1, r).**
5. **MERGE (A, p, q, r).**

**Source Code:**

**#include <stdio.h>**

**void conqueer(int arr[], int si, int mid, int ei)**

**{**

**int range = (ei - si + 1);**

**int merge[range];**

**int index\_1 = si;**

**int index\_2 = mid + 1;**

**int x = 0;**

**while (index\_1 <= mid && index\_2 <= ei)**

**{**

**if (arr[index\_1] <= arr[index\_2])**

**{**

**merge[x] = arr[index\_1];**

**x++;**

**index\_1++;**

**}**

**else**

**{**

**merge[x] = arr[index\_2];**

**x++;**

**index\_2++;**

**}**

**}**

**while (index\_1 <= mid)**

**{**

**merge[x] = arr[index\_1];**

**x++;**

**index\_1++;**

**}**

**while (index\_2 <= ei)**

**{**

**merge[x] = arr[index\_2];**

**x++;**

**index\_2++;**

**}**

**for (int i = 0, j = si; i < range; i++, j++)**

**{**

**arr[j] = merge[i];**

**}**

**}**

**void divide(int arr[], int si, int ei)**

**{ // si->starting index and ei->ending index**

**if (si >= ei)**

**{**

**return;**

**}**

**int mid = si + (ei - si) / 2;**

**divide(arr, si, mid);**

**divide(arr, mid + 1, ei);**

**conqueer(arr, si, mid, ei);**

**}**

**int main()**

**{**

**int n;**

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

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

**int arr[n];**

**printf("Enter the elements: ");**

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

**{**

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

**}**

**divide(arr, 0, n - 1);**

**printf("The sorted array is :");**

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

**{**

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

**}**

**}**

**Output:**

**Type-1**

**Enter the size of array: 5**

**Enter the elements: 45 23 12 67 1**

**The sorted array is :1 12 23 45 67**

**Type-2**

**Enter the size of array: 4**

**Enter the elements: -23 -1 -21 -65**

**The sorted array is :-65 -23 -21 -1**

**2.b) Write a c program to implement merge sort using iterative way.**

**Algorithm:**

**MERGE (A, p, q, r)**

1. **Set n1 = q - p + 1 (Size of left subarray).**
2. **Set n2 = r - q (Size of right subarray).**
3. **Create two temporary arrays L [1...n1] and R [1...n2].**
4. **For i=0 to n1−1:**
5. **Copy A [p + i] to L[i].**
6. **For j=0 to n2−1:**
7. **Copy A [q + 1 + j] to R[j].**
8. **Initialize i = 0, j = 0, k = p.**
9. **While i<n1and j<n2:**
10. **If L[i]≤R[j]:**
11. **A[k] = L[i], increment i.**
12. **Else:**
13. **A[k] = R[j], increment j.**
14. **Increment k.**
15. **While i<n1:**
16. **Copy remaining elements A[k] = L[i], increment i, k.**
17. **While j<n2:**
18. **Copy remaining elements A[k] = R[j], increment j, k.**

**ITERATIVE-MERGE-SORT (A, n)**

1. **Set blk\_size = 1.**
2. **While blk\_size<n:**
3. **For i=0 to n−1 step 2×blk\_size:**
4. **Compute mid = i + blk\_size - 1.**
5. **Compute right = min (i + 2 \* blk\_size - 1, n - 1).**
6. **MERGE (A, i, mid, right).**
7. **Double blk\_size = 2 × blk\_size.**

**Source Code:**

**#include <stdio.h>**

**// Function to merge two sorted subarrays into a single sorted array**

**void sort(int arr[], int st\_in, int md\_in, int ed\_in)**

**{**

**int l\_len = md\_in - st\_in + 1; // Length of left subarray**

**int r\_len = ed\_in - md\_in; // Length of right subarray**

**int left\_arr[l\_len], right\_arr[r\_len]; // Temporary arrays**

**// Copy data to left and right subarrays**

**for (int i = 0; i < l\_len; i++)**

**{**

**left\_arr[i] = arr[st\_in + i];**

**}**

**for (int j = 0; j < r\_len; j++)**

**{**

**right\_arr[j] = arr[md\_in + j + 1];**

**}**

**int i = 0, j = 0; // Indexes for left and right subarrays**

**// Merge elements back into original array in sorted order**

**while (i < l\_len && j < r\_len)**

**{**

**if (left\_arr[i] <= right\_arr[j])**

**{**

**arr[st\_in] = left\_arr[i];**

**i++;**

**}**

**else**

**{**

**arr[st\_in] = right\_arr[j];**

**j++;**

**}**

**st\_in++;**

**}**

**// Copy remaining elements of left subarray if any**

**while (i < l\_len)**

**{**

**arr[st\_in] = left\_arr[i];**

**i++;**

**st\_in++;**

**}**

**// Copy remaining elements of right subarray if any**

**while (j < r\_len)**

**{**

**arr[st\_in] = right\_arr[j];**

**j++;**

**st\_in++;**

**}**

**}**

**// Function to perform iterative merge sort**

**void merge(int arr[], int n)**

**{**

**// Iteratively merge subarrays of size 1, 2, 4, 8, ... until sorted**

**for (int blk\_len = 1; blk\_len < n; blk\_len \*= 2)**

**{**

**for (int i = 0; i < n; i += (2 \* blk\_len))**

**{**

**int right;**

**// Determine the right boundary of the current merge**

**if (i + (2 \* blk\_len) - 1 < n - 1)**

**{**

**right = i + (2 \* blk\_len) - 1;**

**}**

**else**

**{**

**right = n - 1;**

**}**

**// Merge the current pair of subarrays**

**sort(arr, i, i + blk\_len - 1, right);**

**}**

**}**

**}**

**// Function to print the array**

**void print\_arr(int arr[], int n)**

**{**

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

**{**

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

**}**

**printf("\n");**

**}**

**int main()**

**{**

**int a;**

**printf("Enter the size of array: \n");**

**scanf("%d", &a);**

**int arr[a];**

**printf("Enter the elements: \n");**

**for (int i = 0; i < a; i++)**

**{**

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

**}**

**int en\_in = a;**

**// Perform iterative merge sort**

**merge(arr, a);**

**printf("The sorted array is:\n");**

**for (int i = 0; i < en\_in; i++)**

**{**

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

**}**

**printf("\n");**

**return 0;**

**}**

**Output->**

**Type-1**

**Enter the size of array: 5**

**Enter the elements: 6 3 1 8 7**

**The sorted array is: 1 3 6 7 8**

**Type-2**

**Enter the size of array: 6**

**Enter the elements: 8 3 6 1 4 9**

**The sorted array is: 1 3 4 6 8 9**

**3.a) Write a c program to implement Strassen's Matrix Multiplication algorithm for square matrices using divide and  
conquer.**

**Algorithm:**

**MATRIX-ADD (A, B, C, size)**

1. **For i=0 to size−1:**
2. **For j=0 to size−1:**
3. **C[i][j] = A[i][j] + B[i][j].**

**MATRIX-SUB (A, B, C, size)**

1. **For i=0 to size−:**
2. **For j=0 to size−1:**
3. **C[i][j] = A[i][j] - B[i][j].**

**STRASSEN-MULTIPLY (A, B, C, size)**

1. **If size=1:**
2. **C [0][0] = A [0][0] × B [0][0].**
3. **Return.**
4. **Else:**
5. **Set resize = size / 2.**
6. **Divide A into four submatrices: A11, A12, A21, A22.**
7. **Divide B into four submatrices: B11, B12, B21, B22​.**
8. **Compute seven intermediate matrices:**
   * **M1​= STRASSEN-MULTIPLY((A11+A22), (B11+B22))**
   * **M2​= STRASSEN-MULTIPLY((A21+A22), B11​)**
   * **M3​= STRASSEN-MULTIPLY (A11​, (B12−B22))**
   * **M4​= STRASSEN-MULTIPLY (A22​, (B21−B11))**
   * **M5​= STRASSEN-MULTIPLY((A11+A12), B22)**
   * **M6​= STRASSEN-MULTIPLY((A21−A11), (B11+B12))**
   * **M7​= STRASSEN-MULTIPLY((A12−A22), (B21+B22))**
9. **Compute final submatrices of C:**
   * **C11=M1+M4−M5+M7**
   * **C12=M3+M5**
   * **C21=M2+M4**
   * **C22=M1−M2+M3+M6**
10. **Merge C11, C12, C21, C22 into final matrix C.**

**Source Code:**

**#include <stdio.h>**

**#include <math.h>**

**// Function to add two matrices**

**void add(int size, int arr\_a[size][size], int arr\_b[size][size], int res[size][size])**

**{**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

**res[i][j] = arr\_a[i][j] + arr\_b[i][j];**

**}**

**}**

**}**

**// Function to subtract two matrices**

**void sub(int size, int arr\_a[size][size], int arr\_b[size][size], int res[size][size])**

**{**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

**res[i][j] = arr\_a[i][j] - arr\_b[i][j];**

**}**

**}**

**}**

**// Strassen's matrix multiplication function**

**void strassen(int size, int arr\_a[size][size], int arr\_b[size][size], int res[size][size])**

**{**

**if (size == 1)**

**{**

**res[0][0] = arr\_a[0][0] \* arr\_b[0][0];**

**return;**

**}**

**else**

**{**

**int resize = size / 2;**

**// Dividing matrices into 4 submatrices**

**int a11[resize][resize], a12[resize][resize], a21[resize][resize], a22[resize][resize];**

**int b11[resize][resize], b12[resize][resize], b21[resize][resize], b22[resize][resize];**

**for (int i = 0; i < resize; i++)**

**{**

**for (int j = 0; j < resize; j++)**

**{**

**a11[i][j] = arr\_a[i][j];**

**b11[i][j] = arr\_b[i][j];**

**a12[i][j] = arr\_a[i][j + resize];**

**b12[i][j] = arr\_b[i][j + resize];**

**a21[i][j] = arr\_a[i + resize][j];**

**b21[i][j] = arr\_b[i + resize][j];**

**a22[i][j] = arr\_a[i + resize][j + resize];**

**b22[i][j] = arr\_b[i + resize][j + resize];**

**}**

**}**

**// Intermediate matrices**

**int m1[resize][resize], m2[resize][resize], m3[resize][resize], m4[resize][resize], m5[resize][resize], m6[resize][resize], m7[resize][resize];**

**int temp1[resize][resize], temp2[resize][resize];**

**// Computing the 7 matrix multiplications**

**add(resize, a11, a22, temp1);**

**add(resize, b11, b22, temp2);**

**strassen(resize, temp1, temp2, m1);**

**add(resize, a21, a22, temp1);**

**strassen(resize, temp1, b11, m2);**

**sub(resize, b12, b22, temp1);**

**strassen(resize, a11, temp1, m3);**

**sub(resize, b21, b11, temp1);**

**strassen(resize, a22, temp1, m4);**

**add(resize, a11, a12, temp1);**

**strassen(resize, temp1, b22, m5);**

**sub(resize, a21, a11, temp1);**

**add(resize, b11, b12, temp2);**

**strassen(resize, temp1, temp2, m6);**

**sub(resize, a12, a22, temp1);**

**add(resize, b21, b22, temp2);**

**strassen(resize, temp1, temp2, m7);**

**// Computing final quadrants of result matrix**

**int res11[resize][resize], res12[resize][resize], res21[resize][resize], res22[resize][resize];**

**add(resize, m1, m4, temp1);**

**sub(resize, temp1, m5, temp2);**

**add(resize, temp2, m7, res11);**

**add(resize, m3, m5, res12);**

**add(resize, m2, m4, res21);**

**sub(resize, m1, m2, temp1);**

**add(resize, temp1, m3, temp2);**

**add(resize, temp2, m6, res22);**

**// Merging results into final matrix**

**for (int i = 0; i < resize; i++)**

**{**

**for (int j = 0; j < resize; j++)**

**{**

**res[i][j] = res11[i][j];**

**res[i][j + resize] = res12[i][j];**

**res[i + resize][j] = res21[i][j];**

**res[i + resize][j + resize] = res22[i][j];**

**}**

**}**

**}**

**}**

**// Function to print a matrix**

**void print\_arr(int size, int arr[size][size])**

**{**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

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

**}**

**printf("\n");**

**}**

**printf("\n");**

**}**

**// Function to get the next power of 2 greater than or equal to size**

**int cov(int size)**

**{**

**for (int i = 0;; i++)**

**{**

**if (pow(2, i) >= size)**

**{**

**return pow(2, i);**

**}**

**}**

**}**

**int main()**

**{**

**printf("Enter the dimension of the square matrix n:\n");**

**int size;**

**scanf("%d", &size);**

**int new\_size = cov(size);**

**int arr\_a[new\_size][new\_size];**

**int arr\_b[new\_size][new\_size];**

**// Initializing matrices to 0**

**for (int i = 0; i < new\_size; i++)**

**{**

**for (int j = 0; j < new\_size; j++)**

**{**

**arr\_a[i][j] = 0;**

**arr\_b[i][j] = 0;**

**}**

**}**

**// Input matrices**

**printf("Enter the elements of the first matrix:\n");**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

**scanf("%d", &arr\_a[i][j]);**

**}**

**}**

**printf("Enter the elements of the second matrix:\n");**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

**scanf("%d", &arr\_b[i][j]);**

**}**

**}**

**int arr\_res[new\_size][new\_size];**

**strassen(new\_size, arr\_a, arr\_b, arr\_res);**

**printf("Multiplication of two matrices is:\n");**

**print\_arr(new\_size, arr\_res);**

**}**

**Output:**

**Type-1**

**Enter the dimension of the square matrix n:**

**4**

**Enter the elements of the first matrix:**

**4 3 2 1**

**8 7 6 5**

**12 11 10 9**

**16 15 14 13**

**Enter the elements of the second matrix:**

**20 19 18 17**

**21 22 23 24**

**28 27 26 25**

**29 30 31 32**

**Multiplication of two matrices is:**

**228     226     224     222**

**620     618     616     614**

**1012    1010    1008    1006**

**1404    1402    1400    1398**

**Type-2**

**Enter the dimension of the square matrix n:**

**2**

**Enter the elements of the first matrix:**

**34 23**

**54 87**

**Enter the elements of the second matrix:**

**12 15**

**32 73**

**Multiplication of two matrices is:**

**1144    2189**

**3432    7161**

**3.b) modify and implement Strassen's Matrix Multiplication algorithm so that it works with non-square matrices as well.**

**Algorithm:**

**MATRIX-ADD (A, B, C, size)**

1. **For i=0 to size−1:**
2. **For j=0 to size−1:**
3. **C[i][j] = A[i][j] + B[i][j].**

**MATRIX-SUB (A, B, C, size)**

1. **For i=0 to size−:**
2. **For j=0 to size−1:**
3. **C[i][j] = A[i][j] - B[i][j].**

**STRASSEN-MULTIPLY (A, B, C, size)**

1. **If size=1:**
2. **C [0][0] = A [0][0] × B [0][0].**
3. **Return.**
4. **Else:**
5. **Set resize = size / 2.**
6. **Divide A into four submatrices: A11, A12, A21, A22.**
7. **Divide B into four submatrices: B11, B12, B21, B22​.**
8. **Compute seven intermediate matrices:**
   * **M1​= STRASSEN-MULTIPLY((A11+A22), (B11+B22))**
   * **M2​= STRASSEN-MULTIPLY((A21+A22), B11​)**
   * **M3​= STRASSEN-MULTIPLY (A11​, (B12−B22))**
   * **M4​= STRASSEN-MULTIPLY (A22​, (B21−B11))**
   * **M5​= STRASSEN-MULTIPLY((A11+A12), B22)**
   * **M6​= STRASSEN-MULTIPLY((A21−A11), (B11+B12))**
   * **M7​= STRASSEN-MULTIPLY((A12−A22), (B21+B22))**
9. **Compute final submatrices of C:**
   * **C11=M1+M4−M5+M7**
   * **C12=M3+M5**
   * **C21=M2+M4**
   * **C22=M1−M2+M3+M6**
10. **Merge C11, C12, C21, C22 into final matrix C.**

**Source Code:**

**#include <stdio.h>**

**#include <math.h>**

**// Function to add two matrices**

**void add(int size, int arr\_a[size][size], int arr\_b[size][size], int res[size][size])**

**{**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

**res[i][j] = arr\_a[i][j] + arr\_b[i][j];**

**}**

**}**

**}**

**// Function to subtract two matrices**

**void sub(int size, int arr\_a[size][size], int arr\_b[size][size], int res[size][size])**

**{**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

**res[i][j] = arr\_a[i][j] - arr\_b[i][j];**

**}**

**}**

**}**

**// Strassen's Matrix Multiplication function**

**void strassen(int size, int arr\_a[size][size], int arr\_b[size][size], int res[size][size])**

**{**

**if (size == 1)**

**{**

**res[0][0] = arr\_a[0][0] \* arr\_b[0][0]; // Base case**

**return;**

**}**

**else**

**{**

**int resize = size / 2;**

**// Dividing matrices into sub-matrices**

**int a11[resize][resize], a12[resize][resize];**

**int a21[resize][resize], a22[resize][resize];**

**int b11[resize][resize], b12[resize][resize];**

**int b21[resize][resize], b22[resize][resize];**

**for (int i = 0; i < resize; i++)**

**{**

**for (int j = 0; j < resize; j++)**

**{**

**a11[i][j] = arr\_a[i][j];**

**b11[i][j] = arr\_b[i][j];**

**a12[i][j] = arr\_a[i][j + resize];**

**b12[i][j] = arr\_b[i][j + resize];**

**a21[i][j] = arr\_a[i + resize][j];**

**b21[i][j] = arr\_b[i + resize][j];**

**a22[i][j] = arr\_a[i + resize][j + resize];**

**b22[i][j] = arr\_b[i + resize][j + resize];**

**}**

**}**

**// Intermediate matrices**

**int res11[resize][resize], res12[resize][resize];**

**int res21[resize][resize], res22[resize][resize];**

**int m1[resize][resize], m2[resize][resize], m3[resize][resize], m4[resize][resize], m5[resize][resize], m6[resize][resize], m7[resize][resize];**

**int temp1[resize][resize], temp2[resize][resize];**

**// Computing the 7 Strassen products**

**add(resize, a11, a22, temp1);**

**add(resize, b11, b22, temp2);**

**strassen(resize, temp1, temp2, m1);**

**add(resize, a21, a22, temp1);**

**strassen(resize, temp1, b11, m2);**

**sub(resize, b12, b22, temp1);**

**strassen(resize, a11, temp1, m3);**

**sub(resize, b21, b11, temp1);**

**strassen(resize, a22, temp1, m4);**

**add(resize, a11, a12, temp1);**

**strassen(resize, temp1, b22, m5);**

**sub(resize, a21, a11, temp1);**

**add(resize, b11, b12, temp2);**

**strassen(resize, temp1, temp2, m6);**

**sub(resize, a12, a22, temp1);**

**add(resize, b21, b22, temp2);**

**strassen(resize, temp1, temp2, m7);**

**// Computing final result sub-matrices**

**add(resize, m1, m4, temp1);**

**sub(resize, temp1, m5, temp2);**

**add(resize, temp2, m7, res11);**

**add(resize, m3, m5, res12);**

**add(resize, m2, m4, res21);**

**sub(resize, m1, m2, temp1);**

**add(resize, temp1, m3, temp2);**

**add(resize, temp2, m6, res22);**

**// Combining results into final matrix**

**for (int i = 0; i < resize; i++)**

**{**

**for (int j = 0; j < resize; j++)**

**{**

**res[i][j] = res11[i][j];**

**res[i][j + resize] = res12[i][j];**

**res[i + resize][j] = res21[i][j];**

**res[i + resize][j + resize] = res22[i][j];**

**}**

**}**

**}**

**}**

**// Function to print a matrix**

**void print\_arr(int size, int arr[size][size])**

**{**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

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

**}**

**printf("\n");**

**}**

**printf("\n");**

**}**

**// Function to find next power of 2 for padding**

**int cov(int size)**

**{**

**for (int i = 0;; i++)**

**{**

**if (pow(2, i) >= size)**

**{**

**return pow(2, i);**

**}**

**}**

**}**

**// Function to find the maximum of three numbers**

**int find\_max(int a1, int a2, int b2)**

**{**

**return (a1 > a2) ? ((a1 > b2) ? a1 : b2) : ((a2 > b2) ? a2 : b2);**

**}**

**int main()**

**{**

**// Input matrix dimensions**

**printf("enter the dimention of the first matrix a x b:\n");**

**int a1, a2;**

**scanf("%d %d", &a1, &a2);**

**printf("enter the dimention of the second matrix a x b:\n");**

**int b1, b2;**

**scanf("%d %d", &b1, &b2);**

**if (a2 == b1)**

**{ // Check if multiplication is possible**

**int size = find\_max(a1, a2, b2);**

**int new\_size = cov(size);**

**int arr\_a[new\_size][new\_size];**

**int arr\_b[new\_size][new\_size];**

**// Initializing matrices with zeros**

**for (int i = 0; i < new\_size; i++)**

**{**

**for (int j = 0; j < new\_size; j++)**

**{**

**arr\_a[i][j] = 0;**

**arr\_b[i][j] = 0;**

**}**

**}**

**// Input first matrix**

**printf("enter the elements of the first matrix: \n");**

**for (int i = 0; i < a1; i++)**

**{**

**for (int j = 0; j < a2; j++)**

**{**

**scanf("%d", &arr\_a[i][j]);**

**}**

**}**

**printf("enter the elements of the second matrix: \n");**

**for (int i = 0; i < b1; i++)**

**{**

**for (int j = 0; j < b2; j++)**

**{**

**scanf("%d", &arr\_b[i][j]);**

**}**

**}**

**int arr\_res[new\_size][new\_size];**

**strassen(new\_size, arr\_a, arr\_b, arr\_res);**

**printf("multiplication of two matrices is:\n");**

**print\_arr(new\_size, arr\_res);**

**}**

**else**

**{**

**printf("multiplication can not be done!!");**

**}**

**}**

**Output:**

**Type-1**

**enter the dimention of the first matrix a x b:**

**3 4**

**enter the dimention of the second matrix a x b:**

**4 5**

**enter the elements of the first matrix:**

**23 45 32 67**

**12 21 34 21**

**87 45 36 1**

**enter the elements of the second matrix:**

**98 23 11 26 45**

**23 18 46 27 82**

**32 46 28 45 28**

**98 46 28 64 35**

**multiplication of two matrices is:**

**10879   5893    5095    7541    7966    0       0       0**

**4805    3184    2638    3753    3949    0       0       0**

**10811   4513    4063    5161    8648    0       0       0**

**0       0       0       0       0       0       0       0**

**0       0       0       0       0       0       0       0**

**0       0       0       0       0       0       0       0**

**0       0       0       0       0       0       0       0**

**0       0       0       0       0       0       0       0**

**Type-2**

**enter the dimention of the first matrix a x b:**

**4 5**

**enter the dimention of the second matrix a x b:**

**6 7**

**multiplication can not be done!!**

**4.a) Write a c program to implement matrix chain multiplication problem using dynamic programming.**

**Algorithm:**

**MATRIX-CHAIN-ORDER(p)**

**1. n = length[p] - 1**

**2. let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables**

**3. for i = 1 to n**

**4. do m[i, i] = 0**

**5. for l = 2 to n ▷ l is the chain length**

**6. do for i = 1 to n - l + 1**

**7. do j = i + l - 1**

**8. m[i, j] = ∞**

**9. for k = i to j - 1**

**10. do q = m[i, k] + m[k + 1, j] + p[i-1] \* p[k] \* p[j]**

**11. if q < m[i, j]**

**12. then m[i, j] = q**

**13. s[i, j] = k**

**14. return m and s**

**PRINT-OPTIMAL-PARENS (s, i, j)**

**1. if i == j**

**2. then print "A\_i"**

**3. else print "("**

**4. PRINT-OPTIMAL-PARENS (s, i, s[i, j])**

**5. PRINT-OPTIMAL-PARENS (s, s[i, j] + 1, j)**

**6. print ")"**

**Source Code:**

**#include <stdio.h>**

**#define a 100  // Define a constant for the max matrix size**

**// Function to compute the Matrix Chain Multiplication (MCM) cost and split points**

**void mcm(int n, int arr[], int m[a][a], int s[a][a])**

**{**

**// Initialize the cost of multiplying a single matrix to 0**

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

**{**

**m[i][i] = 0;**

**}**

**// Iterate over chain lengths from 2 to n**

**for (int l = 2; l <= n; l++)**

**{**

**for (int i = 1; i <= n - l + 1; i++)**

**{**

**int j = i + l - 1;**

**m[i][j] = 1000000; // Initialize to a large value (infinity)**

**// Try all possible places to split the product**

**for (int k = i; k <= j - 1; k++)**

**{**

**int q = m[i][k] + m[k + 1][j] + arr[i - 1] \* arr[k] \* arr[j];**

**// Update minimum cost and store the best split point**

**if (q < m[i][j])**

**{**

**m[i][j] = q;**

**s[i][j] = k;**

**}**

**}**

**}**

**}**

**}**

**// Function to print the optimal parenthesization of matrices**

**void parens(int s[a][a], int i, int j)**

**{**

**if (i == j)**

**{**

**printf(" A%d ", i); // Print individual matrix**

**}**

**else**

**{**

**printf("("); // Print opening parenthesis**

**parens(s, i, s[i][j]); // Recursively print left sub-chain**

**parens(s, s[i][j] + 1, j); // Recursively print right sub-chain**

**printf(")"); // Print closing parenthesis**

**}**

**}**

**int main()**

**{**

**printf("Enter the number of mattrix:\n");**

**printf("as example 2 for (1,2,3) matrix\n");**

**int size;**

**scanf("%d", &size); // Read number of matrices**

**int arr[size + 1]; // Array to store matrix dimensions**

**// Read dimensions of matrices**

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

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

**}**

**// Display input matrix dimensions**

**printf("input matrix dimensions are : \n");**

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

**printf("(%d %d) ", arr[i], arr[i+1]);**

**}**

**printf("\n");**

**int m[a][a] = {0}; // Cost matrix**

**int s[a][a] = {0}; // Split matrix**

**// Compute optimal multiplication order and cost**

**mcm(size, arr, m, s);**

**// Print cost matrix**

**printf("The cost matrix is:\n");**

**for (int i = 1; i <= size; i++)**

**{**

**for (int j = 1; j <= size; j++)**

**{**

**if(i == j || i < j){**

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

**}**

**else{**

**printf("\t");**

**}**

**}**

**printf("\n");**

**}**

**// Print parenthesization matrix**

**printf("The parenthesis matrix is:\n");**

**for (int i = 1; i <= size; i++)**

**{**

**for (int j = 1; j <= size; j++)**

**{**

**if(i < j){**

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

**}**

**else{**

**printf("\t");**

**}**

**}**

**printf("\n");**

**}**

**// Print the optimal parenthesization sequence**

**printf("The optimal solution is\n");**

**parens(s, 1, size);**

**printf("\n");**

**return 0;**

**}**

**Output:**

**Type-1**

**Enter the number of mattrix:**

**as example 2 for (1,2,3) matrix**

**5**

**5 7 6 3 9 3**

**input matrix dimensions are :**

**(5 7) (7 6) (6 3) (3 9) (9 3)**

**The cost matrix is:**

**0       210     231     366     357**

**0       126     315     261**

**0       162     135**

**0       81**

**0**

**The parenthesis matrix is:**

**1       1       3       3**

**2       3       2**

**3       3**

**4**

**The optimal solution is**

**(( A1 ( A2  A3 ))( A4  A5 ))**

**Type-2**

**Enter the number of mattrix:**

**as example 2 for (1,2,3) matrix**

**3**

**4 2 6 4**

**input matrix dimensions are :**

**(4 2) (2 6) (6 4)**

**The cost matrix is:**

**0       48      80**

**0       48**

**0**

**The parenthesis matrix is:**

**1       1**

**2**

**The optimal solution is**

**( A1 ( A2  A3 ))**

**4.b) MCM problem using top-down approach of dynamic programming (memorization).**

**Algorithm:**

**MCM(A, i, j)**

**1. If i == j:**

**2. Return 0.**

**3. If dp[i][j] ≠ -1:**

**4. Return dp[i][j] (Use stored result).**

**5. Initialize min = large value.**

**6. For k = i to j-1:**

**7. Compute cost = MCM(A, i, k) + MCM(A, k+1, j) + (A[i-1] × A[k] × A[j]).**

**8. If cost < min:**

**9. min = cost.**

**10. Store split index split[i][j] = k.**

**11. Store result dp[i][j] = min.**

**12. Return dp[i][j].**

**OPTIMAL-PARENTHESES(i, j)**

**1. If i == j:**

**2. Print "A\_i".**

**3. Else:**

**4. Print "(".**

**5. Call OPTIMAL-PARENTHESES(i, split[i][j]).**

**6. Call OPTIMAL-PARENTHESES(split[i][j] + 1, j).**

**7. Print ")".**

**Source Code:**

**#include <stdio.h>**

**#define N 100**

**int dp[N][N]; // Memoization table for storing results of subproblems**

**int split[N][N]; // Table to store split indices for optimal parenthesization**

**// Function to find the minimum number of multiplications required**

**int mcm(int arr[], int i, int j)**

**{**

**if (i == j)**

**return 0; // Only one matrix, no multiplication needed**

**if (dp[i][j] != -1)**

**return dp[i][j]; // Use stored result if available**

**int min = 1000000; // Initialize with a large value**

**for (int k = i; k < j; k++)**

**{**

**// Recursively calculate minimum cost of multiplying matrices**

**int sum = mcm(arr, i, k) + mcm(arr, k + 1, j) + arr[i - 1] \* arr[k] \* arr[j];**

**if (sum < min)**

**{**

**min = sum;**

**split[i][j] = k; // Store the split index for optimal solution**

**}**

**}**

**return dp[i][j] = min; // Store the computed value in memoization table**

**}**

**// Function to print optimal parenthesization**

**void parenthesis(int i, int j)**

**{**

**if (i == j)**

**{**

**printf(" A%d ", i);**

**}**

**else**

**{**

**printf("(");**

**parenthesis(i, split[i][j]);     // Print left part**

**parenthesis(split[i][j] + 1, j); // Print right part**

**printf(")");**

**}**

**}**

**int main()**

**{**

**printf("Enter the number of matrices:\n");**

**printf("As example, enter 2 for matrices with dimensions (1,2,3)like that \n");**

**int size;**

**scanf("%d", &size); // Read number of matrices**

**int arr[size + 1]; // Array to store matrix dimensions**

**// Read dimensions of matrices**

**printf("Enter the matrix dimensions: \n");**

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

**{**

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

**}**

**// Display input matrix dimensions**

**printf("Input matrix dimensions are: \n");**

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

**{**

**printf("(%d %d) ", arr[i], arr[i + 1]);**

**}**

**printf("\n");**

**// Initialize memoization table with -1**

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

**{**

**for (int j = 0; j < N; j++)**

**{**

**dp[i][j] = -1;**

**}**

**}**

**// Compute minimum multiplication cost**

**int min\_multiplications = mcm(arr, 1, size);**

**printf("Minimum number of multiplications is: %d\n", min\_multiplications);**

**printf("Optimal parenthesization is: ");**

**parenthesis(1, size);**

**printf("\n");**

**return 0;**

**}**

**Output:**

**Type-1**

**Enter the number of matrices:**

**As example, enter 2 for matrices with dimensions (1,2,3)like that**

**4**

**Enter the matrix dimensions:**

**5 4 3 2 1**

**Input matrix dimensions are:**

**(5 4) (4 3) (3 2) (2 1)**

**Minimum number of multiplications is: 38**

**Optimal parenthesization is: ( A1 ( A2 ( A3  A4 )))**

**Type-2**

**Enter the number of matrices:**

**As example, enter 2 for matrices with dimensions (1,2,3)like that**

**5**

**Enter the matrix dimensions:**

**64 23 21 12 8 3**

**Input matrix dimensions are:**

**(64 23) (23 21) (21 12) (12 8) (8 3)**

**Minimum number of multiplications is: 6909**

**Optimal parenthesization is: ( A1 ( A2 ( A3 ( A4  A5 ))))**

**5) Write a c program to implement fractional knapsack problem using greedy method.**

**Algorithm:**

**KnapsackGreedy(items, W)**

**1. Sort items by value-to-weight ratio in descending order.**

**2. Initialize total\_value = 0 and remaining\_capacity = W.**

**3. For each item (weight, value):**

**4. If weight ≤ remaining\_capacity:**

**5. Take the whole item.**

**6. Update total\_value += value.**

**7. Decrease remaining\_capacity -= weight.**

**8. Else:**

**9. Take fraction of item that fits.**

**10. Update total\_value += (value/weight) × remaining\_capacity.**

**11. Break (knapsack is full).**

**12. Return total\_value.**

**Source Code:**

**#include<stdio.h>**

**// Function to swap two integers**

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

**{**

**int temp = \*a;**

**\*a = \*b;**

**\*b = temp;**

**}**

**// QuickSort function to sort items based on profit-to-weight ratio**

**void quick(int a, int b, int arr[a][b], int st\_in, int en\_in)**

**{**

**if (st\_in < en\_in)**

**{**

**int pivot = arr[2][en\_in]; // Choosing the last element as pivot**

**int index = st\_in - 1;   // Pointer for the smaller element**

**// Partitioning the array based on profit-to-weight ratio**

**for (int i = st\_in; i < en\_in; i++)**

**{**

**if (arr[2][i] >= pivot) // If current element has higher or equal ratio**

**{**

**index++;**

**swap(&arr[0][i], &arr[0][index]); // Swap profits**

**swap(&arr[1][i], &arr[1][index]); // Swap weights**

**swap(&arr[2][i], &arr[2][index]); // Swap ratios**

**}**

**}**

**index++;**

**swap(&arr[0][index], &arr[0][en\_in]); // Swap pivot profit to correct position**

**swap(&arr[1][index], &arr[1][en\_in]); // Swap pivot weight to correct position**

**swap(&arr[2][index], &arr[2][en\_in]); // Swap pivot ratio to correct position**

**// Recursively sorting left and right subarrays**

**quick(a, b, arr, st\_in, index - 1);**

**quick(a, b, arr, index + 1, en\_in);**

**}**

**}**

**// Function to print items in a structured format**

**void print\_items(int a, int b, int arr[a][b]){**

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

**if(i == 0){**

**printf("profit:\t");**

**}**

**else if(i == 1){**

**printf("weight:\t");**

**}**

**else{**

**printf("ratio:\t");**

**}**

**for(int j = 0; j < b; j++){**

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

**}**

**printf("\n");**

**}**

**printf("\n");**

**}**

**int main(){**

**int a = 3, b; // a represents profit, weight, and ratio; b represents the number of items**

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

**scanf("%d", &b);**

**int arr[a][b]; // 2D array to store profit, weight, and ratio**

**printf("Enter profit along with item weight:\n");**

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

**printf("profit: ");**

**scanf("%d", &arr[0][i]);**

**printf("weight: ");**

**scanf("%d", &arr[1][i]);**

**arr[2][i] = arr[0][i] / arr[1][i]; // Calculate profit-to-weight ratio**

**}**

**printf("Before sorting the items by profit ratio:\n");**

**print\_items(a, b, arr);**

**printf("After sorting the items by profit ratio:\n");**

**quick(a, b, arr, 0, b - 1);**

**print\_items(a, b, arr);**

**int taken[2][b]; // Array to store selected items**

**printf("Enter knapsack capacity: \n");**

**int knapsack\_cap;**

**scanf("%d", &knapsack\_cap);**

**int w = 0; // Total weight of selected items**

**int p = 0; // Total profit of selected items**

**int i = 0;**

**// Selecting items based on profit-to-weight ratio**

**for(i; i < b; i++){**

**if(w + arr[1][i] <= knapsack\_cap){ // If item can be fully taken**

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

**taken[1][i] = arr[1][i]; // Store weight**

**p += arr[0][i];**

**taken[0][i] = arr[0][i]; // Store profit**

**}**

**else{ // Take a fraction of the item if capacity is exceeded**

**int w\_rest = (knapsack\_cap - w);**

**taken[1][i] = w\_rest;**

**int p\_rest = arr[2][i] \* w\_rest;**

**taken[0][i] = p\_rest;**

**w += w\_rest;**

**p += p\_rest;**

**break;**

**}**

**}**

**// Printing selected items**

**printf("list of taken items: \n");**

**for(int j = 0; j < 2; j++){**

**if(j == 0){**

**printf("profit:\t");**

**}**

**else{**

**printf("weight:\t");**

**}**

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

**printf("%d\t", taken[j][k]);**

**}**

**printf("\n");**

**}**

**printf("\n");**

**printf("total %d items were taken\nand profit is %d", w, p);**

**}**

**Output:**

**Enter the number of items:**

**4**

**Enter profit along with item weight:**

**profit: 280**

**weight: 40**

**profit: 100**

**weight: 10**

**profit: 120**

**weight: 20**

**profit: 120**

**weight: 24**

**Before sorting the items by profit ratio:**

**profit: 280     100     120     120**

**weight: 40      10      20      24**

**ratio:  7       10      6       5**

**After sorting the items by profit ratio:**

**profit: 100     280     120     120**

**weight: 10      40      20      24**

**ratio:  10      7       6       5**

**Enter knapsack capacity:**

**60**

**list of taken items:**

**profit: 100     280     60**

**weight: 10      40      10**

**total 60 items were taken**

**and profit is 440**

**6) write a c program to implement 0-1 knapsack problem using dynamic programming.**

**Algorithm:**

**KnapsackDP(n, W, val, wt)**

**1. Create a DP table matrix[n+1][W+1].**

**2. Initialize matrix[0][j] = 0 for all j (0 capacity case).**

**3. Initialize matrix[i][0] = 0 for all i (0 items case).**

**3. For i = 1 to n:**

**4. For j = 1 to W:**

**5. If j >= wt[i-1]:**

**6. matrix[i][j] = max(val[i-1] + matrix[i-1][j - wt[i-1]], matrix[i-1][j]).**

**7. Else:**

**8. matrix[i][j] = matrix[i-1][j].**

**9. Return matrix[n][W].**

**Source Code:**

**#include <stdio.h>**

**int main() {**

**int n, capacity;**

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

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

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

**scanf("%d", &capacity);**

**int profit[100], weight[100];**

**float ratio[100];**

**printf("Enter profit of each item:\n");**

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

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

**}**

**printf("Enter weight of each item:\n");**

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

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

**}**

**// Calculate profit/weight ratio**

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

**ratio[i] = (float) profit[i] / weight[i];**

**}**

**// big to small**

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

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

**if (ratio[i] < ratio[j]) {**

**//ratio part**

**float tempR = ratio[i];**

**ratio[i] = ratio[j];**

**ratio[j] = tempR;**

**// profit part**

**int tempP = profit[i];**

**profit[i] = profit[j];**

**profit[j] = tempP;**

**//weight part**

**int tempW = weight[i];**

**weight[i] = weight[j];**

**weight[j] = tempW;**

**}**

**}**

**}**

**int remaining = capacity;**

**float totalProfit = 0;**

**printf("\nItems taken in the knapsack:\n");**

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

**if (weight[i] <= remaining) {**

**// full**

**printf("Item %d: 100%% [Profit: %d, Weight: %d]\n", i + 1, profit[i], weight[i]);**

**totalProfit += profit[i];**

**remaining -= weight[i];**

**} else {**

**// others**

**float fraction = (float) remaining / weight[i];**

**printf("Item %d: %.2f%% [Profit: %d, Weight: %d]\n", i + 1, fraction \* 100, profit[i], weight[i]);**

**totalProfit += profit[i] \* fraction;**

**break;**

**}**

**}**

**printf("\n The Maximum Profit: %.2f\n", totalProfit);**

**return 0;**

**}**

**Output:**

**Enter number of items: 4**

**Enter capacity of the knapsack: 60**

**Enter profit of each item:**

**280 100 120 120**

**Enter weight of each item:**

**40 10 20 24**

**Items taken in the knapsack:**

**Item 1: 100% [Profit: 100, Weight: 10]**

**Item 2: 100% [Profit: 280, Weight: 40]**

**Item 3: 50.00% [Profit: 120, Weight: 20]**

**The Maximum Profit: 440.00**

**7) write a c program to implement nqueens problem using backtracking.**

**Algorithm:**

**NQueens(k, n)**

**1. For i = 1 to n:**

**2. If Place(k, i) is true:**

**3. x[k] = i // Store column position of queen**

**4. If k == n:**

**5. Print x[1:n] // Print solution when all queens are placed**

**6. Else:**

**7. NQueens(k+1, n) // Recursively place the next queen**

**Place(k, i)**

**1. For j = 1 to (k-1): // Check previous rows**

**2. If x[j] == i or Abs(x[j] - i) == Abs(j - k): // Column or diagonal conflict**

**3. Return false**

**4. Return true**

**Source Code:**

**#include<stdio.h>**

**#include<stdbool.h>**

**#include<math.h>**

**// Function to print the board configuration**

**void board(int x[], int n) {**

**printf("solutions are : \n");**

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

**printf(" %d ", x[j]+1);**

**}**

**printf("\n");**

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

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

**if (x[i] == j) {**

**printf(" Q ");**

**} else {**

**printf(" - ");**

**}**

**}**

**printf("\n");**

**}**

**}**

**// Function to check if a queen can be placed at position (k, i)**

**bool place(int x[], int k, int i) {**

**for (int j = 0; j < k; j++) {**

**if ((x[j] == i) || (fabs(x[j] - i) == fabs(j - k))) {**

**return false;**

**}**

**}**

**return true;**

**}**

**// Recursive function to solve the N-Queens problem**

**void nqueen(int x[], int k, int n) {**

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

**if (place(x, k, i)) {**

**x[k] = i;**

**if (k == n - 1) {**

**board(x, n);**

**printf("\n");**

**} else {**

**nqueen(x, k + 1, n);**

**}**

**}**

**}**

**}**

**int main() {**

**int n;**

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

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

**int x[n]; // Array to store queen positions**

**nqueen(x, 0, n);**

**return 0;**

**}**

**Output:**

**Enter the number of queens: 4**

**solutions are :**

**2  4  1  3**

**-  Q  -  -**

**-  -  -  Q**

**Q  -  -  -**

**-  -  Q  -**

**solutions are :**

**3  1  4  2**

**-  -  Q  -**

**Q  -  -  -**

**-  -  -  Q**

**-  Q  -  -**

**Enter the number of queens: 5**

**solutions are :**

**1  3  5  2  4**

**Q  -  -  -  -**

**-  -  Q  -  -**

**-  -  -  -  Q**

**-  Q  -  -  -**

**-  -  -  Q  -**

**solutions are :**

**1  4  2  5  3**

**Q  -  -  -  -**

**-  -  -  Q  -**

**-  Q  -  -  -**

**-  -  -  -  Q**

**-  -  Q  -  -**

**solutions are :**

**2  4  1  3  5**

**-  Q  -  -  -**

**-  -  -  Q  -**

**Q  -  -  -  -**

**-  -  Q  -  -**

**-  -  -  -  Q**

**solutions are :**

**2  5  3  1  4**

**-  Q  -  -  -**

**-  -  -  -  Q**

**-  -  Q  -  -**

**Q  -  -  -  -**

**-  -  -  Q  -**

**solutions are :**

**3  1  4  2  5**

**-  -  Q  -  -**

**Q  -  -  -  -**

**-  -  -  Q  -**

**-  Q  -  -  -**

**-  -  -  -  Q**

**solutions are :**

**3  5  2  4  1**

**-  -  Q  -  -**

**-  -  -  -  Q**

**-  Q  -  -  -**

**-  -  -  Q  -**

**Q  -  -  -  -**

**solutions are :**

**4  1  3  5  2**

**-  -  -  Q  -**

**Q  -  -  -  -**

**-  -  Q  -  -**

**-  -  -  -  Q**

**-  Q  -  -  -**

**solutions are :**

**4  2  5  3  1**

**-  -  -  Q  -**

**-  Q  -  -  -**

**-  -  -  -  Q**

**-  -  Q  -  -**

**Q  -  -  -  -**

**solutions are :**

**5  2  4  1  3**

**-  -  -  -  Q**

**-  Q  -  -  -**

**-  -  -  Q  -**

**Q  -  -  -  -**

**-  -  Q  -  -**

**solutions are :**

**5  3  1  4  2**

**-  -  -  -  Q**

**-  -  Q  -  -**

**Q  -  -  -  -**

**-  -  -  Q  -**

**-  Q  -  -  -**

**8.a) Write a c program to implement breadth first search using adjacency matrix representation.**

**Algorithm:**

**BFS(adj\_mat, node\_size, start)**

**1. Initialize an empty queue q.**

**2. Declare an array visited[node\_size] and set all values to 0.**

**3. Read the adjacency matrix adj\_mat[node\_size][node\_size].**

**4. Read the starting node start.**

**5. Mark visited[start] = 1 and enqueue start into q.**

**6. While the queue is not empty:**

**7. Dequeue a node node from q.**

**8. For each adjacent node i (from 0 to node\_size - 1):**

**9. If adj\_mat[node][i] == 1 and visited[i] == 0:**

**10. Mark visited[i] = 1.**

**11. Enqueue i into q.**

**12. Print the visited node i.**

**Source Code:**

**#include<stdio.h>**

**#include<stdlib.h>**

**// Structure for queue implementation**

**struct queue{**

**int size;**

**int rear;**

**int front;**

**int \* arr;**

**};**

**// Function to check if the queue is empty**

**int isEmpty(struct queue \* q){**

**if(q->rear == q->front){**

**return 1;**

**}**

**else{**

**return 0;**

**}**

**}**

**// Function to check if the queue is full**

**int isFull(struct queue \* q){**

**if(q->rear == q->size-1){**

**return 1;**

**}**

**else{**

**return 0;**

**}**

**}**

**// Function to display the queue elements**

**void show\_queue(struct queue \* q){**

**int a = (q->front == -1)?0:q->front+1;**

**int b = (q->rear == -1)?0:q->rear;**

**if(isEmpty(q)){**

**printf("queue: ");**

**}**

**else{**

**printf("queue: ");**

**for(int i = a; i <= b; i++){**

**printf("%d ", q->arr[i]);**

**}**

**}**

**printf("\n");**

**}**

**// Function to add an element to the queue**

**void enqueue(struct queue \* q, int val){**

**if(isFull(q)){**

**printf("queue is full\n");**

**}**

**else{**

**q->rear++;**

**q->arr[q->rear] = val;**

**}**

**}**

**// Function to remove an element from the queue**

**int dequeue(struct queue \* q){**

**int a = -1;**

**if(isEmpty(q)){**

**printf("queue is empty!");**

**}**

**else{**

**q->front++;**

**a = q->arr[q->front];**

**}**

**return a;**

**}**

**int main(){**

**// Initializing the queue**

**struct queue q;**

**q.rear = q.front = -1;**

**q.size = 20;**

**q.arr = (int\*) malloc(q.size \* sizeof(int));**

**int node\_size;**

**printf("Enter the number of graph nodes: \n");**

**scanf("%d", &node\_size);**

**// Initializing visited array to track visited nodes**

**int visited[node\_size];**

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

**visited[i] = 0;**

**}**

**// Input adjacency matrix**

**int adj\_mat[node\_size][node\_size];**

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

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

**for(int j = 0; j < node\_size; j++){**

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

**}**

**}**

**int start;**

**printf("Enter the starting node: \n");**

**scanf("%d", &start);**

**printf("node is %d \n", start);**

**visited[start] = 1;**

**enqueue(&q, start);**

**// BFS traversal**

**while(!isEmpty(&q)){**

**int node = dequeue(&q);**

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

**if(adj\_mat[node][i] == 1 && visited[i] == 0){**

**visited[i] = 1;**

**enqueue(&q, i);**

**printf("node is %d \n", i);**

**}**

**}**

**}**

**printf("Finally Breadth first search is completed!!");**

**return 0;**

**}**

**Output:**

**Type-1**

**Enter the number of graph nodes:**

**5**

**Enter the adjacency matrix of the graph:**

**1 0 0 1 1**

**1 0 1 0 1**

**0 1 0 1 0**

**0 1 1 0 0**

**1 0 1 0 1**

**Enter the starting node:**

**0**

**node is 0**

**node is 3**

**node is 4**

**node is 1**

**node is 2**

**Finally Breadth first search is completed!!**

**Type-1**

**Enter the number of graph nodes:**

**7**

**Enter the adjacency matrix of the graph:**

**0 1 1 1 0 0 0**

**1 0 1 0 0 0 0**

**1 1 0 1 1 0 0**

**1 0 1 0 1 0 0**

**0 0 1 1 0 1 1**

**0 0 0 0 1 0 0**

**1 0 1 0 1 0 0**

**Enter the starting node:**

**0**

**node is 0**

**node is 1**

**node is 2**

**node is 3**

**node is 4**

**node is 5**

**node is 6**

**Finally Breadth first search is completed!!**

**8.b) Write a c program to implement breadth first search using adjacency list representation.**

**Algorithm:**

**BFS\_AdjList(adj\_list, len, start)**

**1. Initialize an empty queue q.**

**2. Declare an array visited[100] and set all values to 0.**

**3. Read the number of graph nodes len.**

**4. Declare an adjacency list adj\_list[len].**

**5. For each node i in the graph:**

**6. Read the node value and store it in adj\_list[i].node.**

**7. Read the number of edges n connected to adj\_list[i].node.**

**8. Initialize link = NULL.**

**9. For each edge j (from 0 to n-1):**

**10. Read the connected node value.**

**11. Create a new node a and set a->node = connected node.**

**12. If j == 0:**

**13. Set adj\_list[i].add = a.**

**14. Set link = a.**

**15. Else:**

**16. Set link->add = a.**

**17. Update link = a.**

**18. Print the adjacency list.**

**19. Read the starting node start.**

**20. Mark visited[start] = 1 and enqueue start into q.**

**21. While the queue is not empty:**

**22. Dequeue a node node from q.**

**23. Find the adjacency list entry corresponding to node.**

**24. Set link = adj\_list[i].add where adj\_list[i].node == node.**

**25. While link is not NULL:**

**26. If visited[link->node] == 0:**

**27. Mark visited[link->node] = 1.**

**28. Enqueue link->node into q.**

**29. Print the visited node link->node.**

**30. Update link = link->add.**

**31. Print "Breadth-First Search is completed!!".**

**Source Code:**

**#include<stdio.h>**

**#include<stdlib.h>**

**// Structure to represent a graph node**

**struct node{**

**int node;**

**struct node \* add;**

**};**

**// Structure for queue implementation**

**struct queue{**

**int size;**

**int rear;**

**int front;**

**int \* adj\_list;**

**};**

**// Function to check if the queue is empty**

**int isEmpty(struct queue \* q){**

**if(q->rear == q->front){**

**return 1;**

**}**

**else{**

**return 0;**

**}**

**}**

**// Function to check if the queue is full**

**int isFull(struct queue \* q){**

**if(q->rear == q->size-1){**

**return 1;**

**}**

**else{**

**return 0;**

**}**

**}**

**// Function to display the queue elements**

**void show\_queue(struct queue \* q){**

**int a = (q->front == -1)?0:q->front+1;**

**int b = (q->rear == -1)?0:q->rear;**

**if(isEmpty(q)){**

**printf("queue: ");**

**}**

**else{**

**printf("queue: ");**

**for(int i = a; i <= b; i++){**

**printf("%d ", q->adj\_list[i]);**

**}**

**}**

**printf("\n");**

**}**

**// Function to add an element to the queue**

**void enqueue(struct queue \* q, int val){**

**if(isFull(q)){**

**printf("queue is full\n");**

**}**

**else{**

**q->rear++;**

**q->adj\_list[q->rear] = val;**

**}**

**}**

**// Function to remove an element from the queue**

**int dequeue(struct queue \* q){**

**int a = -1;**

**if(isEmpty(q)){**

**printf("queue is empty");**

**}**

**else{**

**q->front++;**

**a = q->adj\_list[q->front];**

**}**

**return a;**

**}**

**int main(){**

**int len;**

**printf("enter the number of graph nodes: \n");**

**scanf("%d", &len);**

**struct queue q;**

**q.rear = q.front = -1;**

**q.size = 20;**

**q.adj\_list = (int\*) malloc(q.size \* sizeof(int));**

**int visited[100];**

**for(int i = 0; i < 100; i++){**

**visited[i] = 0;**

**}**

**struct node adj\_list[len];**

**// Creating adjacency list**

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

**printf("enter the node: \n");**

**scanf("%d",&adj\_list[i].node);**

**int n;**

**printf("number of edges connected to node no %d:\n",adj\_list[i].node);**

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

**struct node \*link = NULL;**

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

**printf("enter connected node:\n");**

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

**scanf("%d",&a->node);**

**a->add = NULL;**

**if(j == 0){**

**adj\_list[i].add = a;**

**link = a;**

**}**

**else {**

**link->add = a;**

**link = a;**

**}**

**}**

**}**

**// Printing adjacency list**

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

**struct node \* link;**

**link = adj\_list[i].add;**

**printf("%d -> ",adj\_list[i].node);**

**do{**

**printf("%d ", link->node);**

**link = link->add;**

**}while(link != NULL);**

**printf("\n");**

**}**

**int start;**

**printf("enter the starting node: \n");**

**scanf("%d", &start);**

**printf("node is %d \n", start);**

**visited[start] = 1;**

**enqueue(&q, start);**

**// BFS traversal**

**while(!isEmpty(&q)){**

**int node = dequeue(&q);**

**struct node \* link = NULL;**

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

**if(adj\_list[i].node == node){**

**link = adj\_list[i].add;**

**break;**

**}**

**}**

**while(link != NULL){**

**if(visited[link->node] == 0){**

**visited[link->node] = 1;**

**enqueue(&q, link->node);**

**printf("node is %d \n", link->node);**

**}**

**link = link->add;**

**}**

**}**

**printf("breadth first search is completed!!");**

**return 0;**

**}**

**Output:**

**Type-1**

**enter the number of graph nodes:**

**5**

**enter the node:**

**1**

**number of edges connected to node no 1:**

**2**

**enter connected node:**

**2**

**enter connected node:**

**4**

**enter the node:**

**2**

**number of edges connected to node no 2:**

**3**

**enter connected node:**

**1**

**enter connected node:**

**4**

**enter connected node:**

**3**

**enter the node:**

**4**

**number of edges connected to node no 4:**

**4**

**enter connected node:**

**1**

**enter connected node:**

**2**

**enter connected node:**

**3**

**enter connected node:**

**5**

**enter the node:**

**3**

**number of edges connected to node no 3:**

**3**

**enter connected node:**

**2**

**enter connected node:**

**4**

**enter connected node:**

**5**

**enter the node:**

**5**

**number of edges connected to node no 5:**

**2**

**enter connected node:**

**4**

**enter connected node:**

**3**

**1 -> 2 4**

**2 -> 1 4 3**

**4 -> 1 2 3 5**

**3 -> 2 4 5**

**5 -> 4 3**

**enter the starting node:**

**1**

**node is 1**

**node is 2**

**node is 4**

**node is 3**

**node is 5**

**breadth first search is completed!!**

**Type-2**

**enter the number of graph nodes:**

**7**

**enter the node:**

**0**

**number of edges connected to node no 0:**

**3**

**enter connected node:**

**0**

**enter connected node:**

**2**

**enter connected node:**

**3**

**enter the node:**

**1**

**number of edges connected to node no 1:**

**2**

**enter connected node:**

**0**

**enter connected node:**

**3**

**enter the node:**

**2**

**number of edges connected to node no 2:**

**2**

**enter connected node:**

**0**

**enter connected node:**

**3**

**enter the node:**

**3**

**number of edges connected to node no 3:**

**4**

**enter connected node:**

**1**

**enter connected node:**

**0**

**enter connected node:**

**2**

**enter connected node:**

**4**

**enter the node:**

**4**

**number of edges connected to node no 4:**

**4**

**enter connected node:**

**2**

**enter connected node:**

**3**

**enter connected node:**

**5**

**enter connected node:**

**6**

**enter the node:**

**5**

**number of edges connected to node no 5:**

**1**

**enter connected node:**

**4**

**enter the node:**

**6**

**number of edges connected to node no 6:**

**1**

**enter connected node:**

**4**

**0 -> 0 2 3**

**1 -> 0 3**

**2 -> 0 3**

**3 -> 1 0 2 4**

**4 -> 2 3 5 6**

**5 -> 4**

**6 -> 4**

**enter the starting node:**

**0**

**node is 0**

**node is 2**

**node is 3**

**node is 1**

**node is 4**

**node is 5**

**node is 6**

**breadth first search is completed!!**

**9.a) write a c program of depth first search using adjacency matrix.**

**Algorithm:**

**DFS(G):**

**1 for each vertex u ∈ G.V**

**2 u.color = WHITE**

**3 u.π = NIL**

**4 time = 0**

**5 for each vertex u ∈ G.V**

**6 if u.color == WHITE**

**7 DFS-VISIT(G, u)**

**DFS-VISIT(G,u):**

**1 time = time + 1**

**2 u.d = time**

**3 u.color = GRAY**

**4 for each v ∈ G.Adj[u]**

**5 if v.color == WHITE**

**6 v.π = u**

**7 DFS-VISIT(G, v)**

**8 u.color = BLACK**

**9 time = time + 1**

**10 u.f = time**

**Source Code:**

**#include<stdio.h>**

**void dfs(int i,int size,int adj\_mat[size][size],int visited[size]){**

**visited[i] = 1;**

**printf("node is %d \n",i);**

**for(int j = 0;j<size;j++){**

**if(adj\_mat[i][j] == 1 && visited[j] == 0){**

**visited[j] = 1;**

**dfs(j,size,adj\_mat,visited);**

**}**

**}**

**}**

**int main(){**

**int node\_size;**

**printf("enter the number of graph nodes: \n");**

**scanf("%d", &node\_size);**

**// Initializing visited array to track visited nodes**

**int visited[node\_size];**

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

**visited[i] = 0;**

**}**

**// Input adjacency matrix**

**int adj\_mat[node\_size][node\_size];**

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

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

**for(int j = 0; j < node\_size; j++){**

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

**}**

**}**

**int start;**

**printf("Enter the starting node: \n");**

**scanf("%d", &start);**

**dfs(start,node\_size,adj\_mat,visited);**

**}**

**Output:**

**enter the number of graph nodes:**

**6**

**Enter the adjacency matrix of the graph:**

**0 1 1 0 0 0**

**1 0 0 1 1 0**

**1 0 0 0 0 0**

**0 1 0 0 0 1**

**0 1 0 0 0 1**

**0 0 0 1 1 0**

**Enter the starting node:**

**0**

**node is 0**

**node is 1**

**node is 3**

**node is 5**

**node is 4**

**node is 2**

**9.b) write a c program of depth first search using list representation.**

**Algorithm:**

**DFS(G):**

**1 for each vertex u ∈ G.V**

**2 u.color = WHITE**

**3 u.π = NIL**

**4 time = 0**

**5 for each vertex u ∈ G.V**

**6 if u.color == WHITE**

**7 DFS-VISIT(G, u)**

**DFS-VISIT(G,u):**

**1 time = time + 1**

**2 u.d = time**

**3 u.color = GRAY**

**4 for each v ∈ G.Adj[u]**

**5 if v.color == WHITE**

**6 v.π = u**

**7 DFS-VISIT(G, v)**

**8 u.color = BLACK**

**9 time = time + 1**

**10 u.f = time**

**Source Code:**

**#include<stdio.h>**

**#include<stdlib.h>**

**// Structure to represent a graph node**

**struct node {**

**int node;**

**struct node \*add;**

**};**

**// Function to find index of a node in the adjacency list**

**int find\_index(int value, int node\_size, struct node adj\_list[]) {**

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

**if (adj\_list[i].node == value) {**

**return i;**

**}**

**}**

**return -1;**

**}**

**// Depth First Search (DFS) function**

**void dfs(int node\_value, int node\_size, struct node adj\_list[], int visited[]) {**

**int index = find\_index(node\_value, node\_size, adj\_list);**

**if (index == -1 || visited[index] == 1) {**

**return;**

**}**

**visited[index] = 1;**

**printf("Visited node: %d\n", node\_value);**

**struct node \*link = adj\_list[index].add;**

**while (link != NULL) {**

**dfs(link->node, node\_size, adj\_list, visited);**

**link = link->add;**

**}**

**}**

**int main() {**

**int node\_size;**

**printf("Enter the number of graph nodes: \n");**

**scanf("%d", &node\_size);**

**int visited[node\_size];**

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

**visited[i] = 0;**

**}**

**struct node adj\_list[node\_size];**

**// Creating adjacency list**

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

**printf("Enter the node number: \n");**

**scanf("%d", &adj\_list[i].node);**

**adj\_list[i].add = NULL;  // Initialize adjacency list head**

**int n;**

**printf("Number of edges connected to node %d: \n", adj\_list[i].node);**

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

**struct node \*link = NULL;**

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

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

**printf("Enter connected node: \n");**

**scanf("%d", &a->node);**

**a->add = NULL;**

**if (j == 0) {**

**adj\_list[i].add = a;**

**link = a;**

**} else {**

**link->add = a;**

**link = a;**

**}**

**}**

**}**

**// Printing adjacency list**

**printf("\nGraph adjacency list:\n");**

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

**struct node \*link = adj\_list[i].add;**

**printf("%d -> ", adj\_list[i].node);**

**while (link != NULL) {**

**printf("%d ", link->node);**

**link = link->add;**

**}**

**printf("\n");**

**}**

**// Running DFS**

**int start;**

**printf("Enter the starting node: ");**

**scanf("%d", &start);**

**printf("Starting DFS from node %d\n", start);**

**dfs(start, node\_size, adj\_list, visited);**

**// Freeing allocated memory**

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

**struct node \*link = adj\_list[i].add;**

**while (link != NULL) {**

**struct node \*temp = link;**

**link = link->add;**

**free(temp);**

**}**

**}**

**return 0;**

**}**

**Output:**

**Enter the number of graph nodes:**

**6**

**Enter the node number:**

**0**

**Number of edges connected to node 0:**

**2**

**Enter connected node:**

**1**

**Enter connected node:**

**2**

**Enter the node number:**

**1**

**Number of edges connected to node 1:**

**2**

**Enter connected node:**

**3**

**Enter connected node:**

**4**

**Enter the node number:**

**2**

**Number of edges connected to node 2:**

**1**

**Enter connected node:**

**5**

**Enter the node number:**

**3**

**Number of edges connected to node 3:**

**0**

**Enter the node number:**

**4**

**Number of edges connected to node 4:**

**0**

**Enter the node number:**

**5**

**Number of edges connected to node 5:**

**0**

**Graph adjacency list:**

**0 -> 1 2**

**1 -> 3 4**

**2 -> 5**

**3 ->**

**4 ->**

**5 ->**

**Enter the starting node: 0**

**Starting DFS from node 0**

**Visited node: 0**

**Visited node: 1**

**Visited node: 3**

**Visited node: 4**

**Visited node: 2**

**Visited node: 5**

**10) write a c program to implement krushkal algorithm for a graph.**

**Algorithm:**

**MST-KRUSKAL(G, w)**

**1 A = ∅**

**2 for each vertex v ∈ G.V**

**3 MAKE-SET(v)**

**4 sort the edges of G.E into nondecreasing order by weight w**

**5 for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight**

**6 if FIND-SET(u) ≠ FIND-SET(v)**

**7 A = A ∪ {(u, v)}**

**8 UNION(u, v)**

**9 return A**

**Source Code:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <stdbool.h>**

**#define INF 99999**

**int \*parent;**

**// Function to find the set of an element i (with path compression)**

**int findSet(int i)**

**{**

**if (parent[i] == i)**

**return i;**

**return parent[i] = findSet(parent[i]);**

**}**

**// Function to perform union of two sets**

**void unionSets(int u, int v)**

**{**

**int setU = findSet(u);**

**int setV = findSet(v);**

**parent[setU] = setV;**

**}**

**// Kruskal's Algorithm**

**void kruskalMST(int \*\*graph, int V)**

**{**

**parent = (int \*)malloc(V \* sizeof(int));**

**int edgeCount = 0, minCost = 0;**

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

**parent[i] = i;**

**// Convert to edge list**

**int maxEdges = V \* V;**

**int edges[maxEdges][3]; // {u, v, weight}**

**int edgeIndex = 0;**

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

**{**

**for (int j = i + 1; j < V; j++)**

**{**

**if (graph[i][j] != 0 && graph[i][j] != INF)**

**{**

**edges[edgeIndex][0] = i;**

**edges[edgeIndex][1] = j;**

**edges[edgeIndex][2] = graph[i][j];**

**edgeIndex++;**

**}**

**}**

**}**

**// Sort edges by weight (Bubble sort)**

**for (int i = 0; i < edgeIndex - 1; i++)**

**{**

**for (int j = 0; j < edgeIndex - i - 1; j++)**

**{**

**if (edges[j][2] > edges[j + 1][2])**

**{**

**for (int k = 0; k < 3; k++)**

**{**

**int temp = edges[j][k];**

**edges[j][k] = edges[j + 1][k];**

**edges[j + 1][k] = temp;**

**}**

**}**

**}**

**}**

**printf("Edges in the Minimum Spanning Tree:\n");**

**for (int i = 0; i < edgeIndex && edgeCount < V - 1; i++)**

**{**

**int u = edges[i][0];**

**int v = edges[i][1];**

**int weight = edges[i][2];**

**if (findSet(u) != findSet(v))**

**{**

**printf("%c - %c \tWeight: %d\n", u + 'a', v + 'a', weight);**

**unionSets(u, v);**

**minCost += weight;**

**edgeCount++;**

**}**

**}**

**printf("minimum cost of spanning tree: %d\n", minCost);**

**free(parent);**

**}**

**// Main function**

**int main()**

**{**

**int V;**

**printf("enter the number of graph nodes: ");**

**scanf("%d", &V);**

**// Allocate 2D adjacency matrix**

**int \*\*graph = (int \*\*)malloc(V \* sizeof(int \*));**

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

**{**

**graph[i] = (int \*)malloc(V \* sizeof(int));**

**}**

**printf("enter the adjacency matrix (use 'i' for INF / no edge):\n");**

**char input[10];**

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

**{**

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

**{**

**scanf("%s", input);**

**if (input[0] == 'i' || input[0] == 'I')**

**graph[i][j] = INF;**

**else**

**graph[i][j] = atoi(input); // Convert string to integer**

**}**

**}**

**kruskalMST(graph, V);**

**// Free memory**

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

**{**

**free(graph[i]);**

**}**

**free(graph);**

**return 0;**

**}**

**Output:**

**Type-1**

**enter the number of graph nodes: 5**

**enter the adjacency matrix (use 'i' for INF / no edge):**

**2 i 3 i 1**

**4 i 3 i i**

**i 3 2 i 1**

**4 i i 7 1**

**i 5 i 3 i**

**Edges in the Minimum Spanning Tree:**

**a - e   Weight: 1**

**c - e   Weight: 1**

**d - e   Weight: 1**

**b - c   Weight: 3**

**minimum cost of spanning tree: 6**

**enter the number of graph nodes: 6**

**Type-2**

**enter the adjacency matrix (use 'i' for INF / no edge):**

**4 3 i i 2 9**

**4 i 6 i 5 i**

**2 i i 4 i 3**

**i i 3 i 6 i**

**6 i 1 i 4 i**

**5 i 6 2 i i**

**Edges in the Minimum Spanning Tree:**

**a - e   Weight: 2**

**a - b   Weight: 3**

**c - f   Weight: 3**

**c - d   Weight: 4**

**b - c   Weight: 6**

**minimum cost of spanning tree: 18**

**11) write a c program to implement prims algorithm for a graph.**

**Algorithm:**

**Algorithm:**

**MST-PRIM(G, w, r)**

**1 for each vertex u ∈ G.V**

**2 u.key = ∞**

**3 u.π = NIL**

**4 r.key = 0**

**5 Q = G.V**

**6 while Q ≠ ∅**

**7 u = EXTRACT-MIN(Q)**

**8 for each v ∈ G.Adj[u]**

**9 if v ∈ Q and w(u, v) < v.key**

**10 v.π = u**

**11 v.key = w(u, v)**

**Source Code:**

**#include<stdio.h>**

**#include<limits.h>**

**#include<stdlib.h>**

**#include<stdbool.h>**

**int extractMin(int key[], bool inMST[],int node\_size) {**

**int min = INT\_MAX, minIndex;**

**for (int v = 0; v < node\_size; v++) {**

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

**min = key[v];**

**minIndex = v;**

**}**

**}**

**return minIndex;**

**}**

**void printMST(int parent[], int node\_size,int graph[node\_size][node\_size]) {**

**printf("Edge\tWeight\n");**

**for (int i = 1; i < node\_size; i++) {**

**printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);**

**}**

**}**

**int main(){**

**int node\_size;**

**printf("enter the number of graph nodes: \n");**

**scanf("%d", &node\_size);**

**// Input adjacency matrix**

**int adj\_mat[node\_size][node\_size];**

**printf("enter the adjacency matrix of the graph along with the edge:\n");**

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

**for(int j = 0; j < node\_size; j++){**

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

**}**

**}**

**int start;**

**printf("enter the starting node: \n");**

**scanf("%d", &start);**

**if(start >= node\_size){**

**printf("no node exists , exit!!");**

**exit(0);**

**}**

**int key[node\_size];**

**int parent[node\_size];**

**bool inMST[node\_size];**

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

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

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

**inMST[i] = false;**

**}**

**key[start] = 0;**

**for (int count = 0; count < node\_size - 1; count++) {**

**int u = extractMin(key, inMST,node\_size);**

**inMST[u] = true;                 // Include u in MST**

**// Update key values of adjacent vertices**

**for (int v = 0; v < node\_size; v++) {**

**if (adj\_mat[u][v] && !inMST[v] && adj\_mat[u][v] < key[v]) {**

**parent[v] = u;**

**key[v] = adj\_mat[u][v];**

**}**

**}**

**}**

**printMST(parent,node\_size,adj\_mat);**

**}**

**Output**

**Type-1**

**enter the number of graph nodes:**

**5**

**enter the adjacency matrix of the graph along with the edge:**

**0 2 0 6 0**

**2 0 3 8 5**

**0 3 0 0 7**

**6 8 0 0 9**

**0 5 7 9 0**

**enter the starting node:**

**0**

**Edge    Weight**

**0 - 1   2**

**1 - 2   3**

**0 - 3   6**

**1 - 4   5**

**Type-2**

**enter the number of graph nodes:**

**9**

**enter the adjacency matrix of the graph along with the edge:**

**0 4 0 0 0 0 0 8 0**

**4 0 8 0 0 0 0 11 0**

**0 8 0 7 0 4 0 0 2**

**0 0 7 0 9 14 0 0 0**

**0 0 0 9 0 10 0 0 0**

**0 0 4 14 10 0 2 0 0**

**0 0 0 0 0 2 0 1 6**

**8 11 0 0 0 0 1 0 7**

**0 0 2 0 0 0 6 7 0**

**enter the starting node:**

**0**

**Edge    Weight**

**0 - 1   4**

**1 - 2   8**

**2 - 3   7**

**3 - 4   9**

**2 - 5   4**

**5 - 6   2**

**6 - 7   1**

**2 - 8   2**

**12) Write a C Program to implement Dijkstra algorithm to implement single source shortest path problem.**

**Algorithm:**

**INITIAL-SINGLE-SOURCE(G,s):**

**1 for each vertex v ∈ G.V**

**2 v.d = ∞**

**3 v.π = NIL**

**4 s.d = 0**

**RELAX(u,v,w):**

**1 if v.d > u.d + w(u, v)**

**2 v.d = u.d + w(u, v)**

**3 v.π = u**

**DIJKSTRA(G,w,s):**

**1 INITIALIZE-SINGLE-SOURCE(G, s)**

**2 S = ∅**

**3 Q = G.V**

**4 while Q ≠ ∅**

**5 u = EXTRACT-MIN(Q)**

**6 S = S ∪ {u}**

**7 for each vertex v ∈ G.Adj[u]**

**8 RELAX(u, v, w)**

**Source Code:**

**#include <stdio.h>**

**#include <limits.h>**

**#include <stdbool.h>**

**#include <stdlib.h>**

**#define INF INT\_MAX**

**// Function to find the vertex with the minimum distance**

**int minDistance(int dist[], bool visited[], int size)**

**{**

**int min = INF, minIndex = -1;**

**for (int v = 0; v < size; v++)**

**{**

**if (!visited[v] && dist[v] < min)**

**{**

**min = dist[v];**

**minIndex = v;**

**}**

**}**

**return minIndex;**

**}**

**// Function to print the shortest distances**

**void printSolution(int dist[], int size)**

**{**

**printf("Vertex\tDistance from Source\n");**

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

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

**}**

**// Dijkstra’s Algorithm using an adjacency matrix**

**void dijkstra(int \*\*graph, int src, int size)**

**{**

**int dist[size];**

**bool visited[size];**

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

**{**

**dist[i] = INF;**

**visited[i] = false;**

**}**

**dist[src] = 0;**

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

**{**

**int u = minDistance(dist, visited, size);**

**if (u == -1)**

**break;**

**visited[u] = true;**

**for (int v = 0; v < size; v++)**

**{**

**if (!visited[v] && graph[u][v] != INF && dist[u] != INF &&**

**dist[u] + graph[u][v] < dist[v])**

**{**

**dist[v] = dist[u] + graph[u][v];**

**}**

**}**

**}**

**printSolution(dist, size);**

**}**

**int main()**

**{**

**int size;**

**printf("Enter the number of graph nodes:\n");**

**scanf("%d", &size);**

**// Dynamic memory allocation**

**int \*\*graph = (int \*\*)malloc(size \* sizeof(int \*));**

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

**graph[i] = (int \*)malloc(size \* sizeof(int));**

**printf("Enter the adjacency matrix (use 'i' for INF / no edge):\n");**

**char input[10];**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

**scanf("%s", input);**

**if (input[0] == 'i' || input[0] == 'I')**

**graph[i][j] = INF;**

**else**

**graph[i][j] = atoi(input);**

**}**

**}**

**int source;**

**printf("Enter the source node (0 to %d):\n", size - 1);**

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

**if (source >= size || source < 0)**

**{**

**printf("Invalid source node!\n");**

**return 1;**

**}**

**dijkstra(graph, source, size);**

**// Free memory**

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

**free(graph[i]);**

**free(graph);**

**return 0;**

**}**

**Output**

**Type-1**

**Enter the number of graph nodes:**

**5**

**Enter the adjacency matrix (use 'i' for INF / no edge):**

**i 4 i 2 i**

**8 i 4 i i**

**2 i 1 i 8**

**9 i 3 i 6**

**1 i i 4 i**

**Enter the source node (0 to 4):**

**0**

**Vertex  Distance from Source**

**0       0**

**1       4**

**2       5**

**3       2**

**4       8**

**Type-2**

**Enter the number of graph nodes:**

**4**

**Enter the adjacency matrix (use 'i' for INF / no edge):**

**3 8 5 i**

**i 5 2 3**

**8 6 1 i**

**5 7 i 8**

**Enter the source node (0 to 3):**

**0**

**Vertex  Distance from Source**

**0       0**

**1       8**

**2       5**

**3       11**

**13) Write a C Program to implement bell-man ford algorithm to implement single source shortest path problem**

**Algorithm:**

**INITIAL-SINGLE-SOURCE(G,s):**

**1 for each vertex v ∈ G.V**

**2 v.d = ∞**

**3 v.π = NIL**

**4 s.d = 0**

**RELAX(u,v,w):**

**1 if v.d > u.d + w(u, v)**

**2 v.d = u.d + w(u, v)**

**3 v.π = u**

**BELLMAN-FORD(G,w,s):**

**1 INITIALIZE-SINGLE-SOURCE(G, s)**

**2 for i = 1 to |G.V| - 1**

**3 for each edge (u, v) ∈ G.E**

**4 RELAX(u, v, w)**

**5 for each edge (u, v) ∈ G.E**

**6 if v.d > u.d + w(u, v)**

**7 return FALSE**

**8 return TRUE**

**Source Code:**

**#include <stdio.h>**

**#include <limits.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#define INF INT\_MAX**

**// Function to print the solution**

**void printSolution(int dist[], int size)**

**{**

**printf("Vertex\tDistance from Source\n");**

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

**{**

**printf("%d\t", i);**

**if (dist[i] == INF)**

**printf("INF\n");**

**else**

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

**}**

**}**

**// Bellman-Ford Algorithm**

**void bellmanFord(int \*\*graph, int src, int size)**

**{**

**int dist[size];**

**// Step 1: Initialize distances**

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

**dist[i] = INF;**

**dist[src] = 0;**

**// Step 2: Relax all edges |V| - 1 times**

**for (int k = 0; k < size - 1; k++)**

**{**

**for (int u = 0; u < size; u++)**

**{**

**for (int v = 0; v < size; v++)**

**{**

**if (graph[u][v] != INF && dist[u] != INF &&**

**dist[u] + graph[u][v] < dist[v])**

**{**

**dist[v] = dist[u] + graph[u][v];**

**}**

**}**

**}**

**}**

**// Step 3: Check for negative-weight cycles**

**for (int u = 0; u < size; u++)**

**{**

**for (int v = 0; v < size; v++)**

**{**

**if (graph[u][v] != INF && dist[u] != INF &&**

**dist[u] + graph[u][v] < dist[v])**

**{**

**printf("Graph contains a negative weight cycle!\n");**

**return;**

**}**

**}**

**}**

**printSolution(dist, size);**

**}**

**int main()**

**{**

**int size;**

**printf("Enter the number of graph nodes:\n");**

**scanf("%d", &size);**

**// Allocate memory for adjacency matrix**

**int \*\*graph = (int \*\*)malloc(size \* sizeof(int \*));**

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

**graph[i] = (int \*)malloc(size \* sizeof(int));**

**printf("Enter the adjacency matrix (use 'i' for INF / no edge):\n");**

**char input[10];**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

**scanf("%s", input);**

**if (input[0] == 'i' || input[0] == 'I')**

**graph[i][j] = INF;**

**else**

**graph[i][j] = atoi(input);**

**}**

**}**

**int source;**

**printf("Enter the source node (0 to %d):\n", size - 1);**

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

**if (source < 0 || source >= size)**

**{**

**printf("Invalid source node!\n");**

**return 1;**

**}**

**bellmanFord(graph, source, size);**

**// Free memory**

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

**free(graph[i]);**

**free(graph);**

**return 0;**

**}**

**Output**

**Type-1**

**Enter the number of graph nodes:**

**4**

**Enter the adjacency matrix (use 'i' for INF / no edge):**

**8 i 2 i**

**i i 5 7**

**8 46 3 i**

**3 2 i 1**

**Enter the source node (0 to 3):**

**0**

**Vertex  Distance from Source**

**0       0**

**1       48**

**2       2**

**3       55**

**Type-2**

**Enter the number of graph nodes:**

**5**

**Enter the adjacency matrix (use 'i' for INF / no edge):**

**9 2 i i 4**

**i 2 9 i 4**

**6 3 i i 2**

**i 3 5 i 6**

**i 8 5 2 3**

**Enter the source node (0 to 4):**

**0**

**Vertex  Distance from Source**

**0       0**

**1       2**

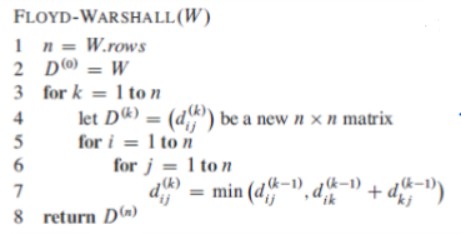
**2       9**

**3       6**

**4       4**

**14) Write a C Program to implement floyd-warshall algorithm to implement single source shortest path problem.**

**Algorithm:**



**Source Code:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <limits.h>**

**#include <string.h>**

**#define INF 99999 // A large value to represent infinity**

**// Floyd-Warshall algorithm**

**void floydWarshall(int \*\*graph, int size)**

**{**

**int \*\*dist = (int \*\*)malloc(size \* sizeof(int \*));**

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

**{**

**dist[i] = (int \*)malloc(size \* sizeof(int));**

**for (int j = 0; j < size; j++)**

**{**

**dist[i][j] = graph[i][j];**

**}**

**}**

**// Main algorithm**

**for (int k = 0; k < size; k++)**

**{**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

**if (dist[i][k] != INF && dist[k][j] != INF &&**

**dist[i][k] + dist[k][j] < dist[i][j])**

**{**

**dist[i][j] = dist[i][k] + dist[k][j];**

**}**

**}**

**}**

**}**

**// Print the matrix in formatted style (no labels)**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

**if (dist[i][j] == INF)**

**printf("%4s", "INF");**

**else**

**printf("%4d", dist[i][j]);**

**}**

**printf("\n");**

**}**

**// Free memory**

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

**{**

**free(dist[i]);**

**}**

**free(dist);**

**}**

**int main()**

**{**

**int size;**

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

**scanf("%d", &size);**

**int \*\*graph = (int \*\*)malloc(size \* sizeof(int \*));**

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

**graph[i] = (int \*)malloc(size \* sizeof(int));**

**printf("Enter the adjacency matrix (use 'i' for INF):\n");**

**char input[10];**

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

**{**

**for (int j = 0; j < size; j++)**

**{**

**scanf("%s", input);**

**if (input[0] == 'i' || input[0] == 'I')**

**graph[i][j] = INF;**

**else**

**graph[i][j] = atoi(input);**

**}**

**}**

**printf("\nShortest distance matrix:\n");**

**floydWarshall(graph, size);**

**// Free memory**

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

**{**

**free(graph[i]);**

**}**

**free(graph);**

**return 0;**

**}**

**Output**

**Type-1**

**Enter the number of vertices in the graph: 4**

**Enter the adjacency matrix (use 'i' for INF):**

**4 i 7 i**

**8 4 i 2**

**i 9 2 i**

**5 2 8 i**

**Shortest distance matrix:**

**4  16   7  18**

**7   4  10   2**

**16   9   2  11**

**5   2   8   4**

**Type-2**

**Enter the number of vertices in the graph: 6**

**Enter the adjacency matrix (use 'i' for INF):**

**1 i 3 i 5 2**

**i 5 4 i 3 6**

**3 2 6 i 8 i**

**i 3 2 8 i 5**

**i 4 7 4 i 9**

**i 9 7 3 i 2**

**Shortest distance matrix:**

**1   5   3   5   5   2**

**7   5   4   7   3   6**

**3   2   6   8   5   5**

**5   3   2   8   6   5**

**9   4   6   4   7   9**

**8   6   5   3   9   2**