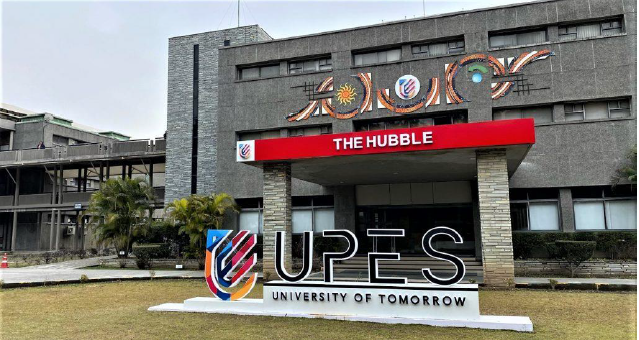


UNIVERSITY OF PETROLEUM

AND ENERGY STUDIES

2023-27

**DAA LAB file**



**NAME:** Pranav Rai

**BATCH:** 51(3rd Semester)

**SAP ID:** 500124723

**ROLL NO.:** R2142231732

**SUBMITTED TO:** Mr. Aryan

**INDEX**

|  |  |  |
| --- | --- | --- |
| S.NO. | TOPIC | SIGN. |
| 1. | LAB-1: Binary Search Tree Insertion |  |
| 2. | LAB-2: Merge Sort and Quick Sort |  |
| 3. | LAB-3: Matrix Multiplication |  |
| 4. | LAB-4: Activity Selection Problem |  |
| 5. | LAB-5: Matrix Chain Multiplication |  |
| 6. | LAB-6: Shortest Path Algorithms |  |
| 7. | LAB-7: 0/1 Knapsack Problem - Greedy vs Dynamic Programming |  |
| 8. | LAB-8: Subset Sum Problem |  |
| 9. | LAB-9: 0/1 Knapsack - Backtracking vs Branch & Bound |  |
| 10. | LAB-10: String Matching Algorithms |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**DAA LAB 500125613**

# **Algorithm\_Lab\_3rd\_sem\_500125613 📚💻**

This repository contains implementations of various algorithms completed as part of our lab exercises. Each lab focuses on a different algorithmic concept, including recursive and iterative approaches, divide and conquer, and greedy strategies. Below is a brief description of each lab:

## LAB-1: Binary Search Tree Insertion 🌳

**Topic:**

Implement the insertion inside iterative and recursive Binary Search Tree (BST) and compare their performance.

* **Description**: This lab demonstrates the implementation of both iterative and recursive approaches to inserting elements into a Binary Search Tree. The performance comparison highlights the differences between these two methods.

### CODE:

|  |
| --- |
| #include <stdio.h> |
| #include <stdlib.h> |
| #include <time.h> |
|  |
| // Structure for a BST node |
| struct Node { |
| int data; |
| struct Node\* left; |
| struct Node\* right; |
| }; |
|  |
| // Create a new node |
| struct Node\* createNode(int data) { |
| struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node)); |
| newNode->data = data; |
| newNode->left = NULL; |
| newNode->right = NULL; |
| return newNode; |
| } |
|  |
| // Iterative BST insertion |
| struct Node\* iterativeInsert(struct Node\* root, int data) { |
| struct Node\* newNode = createNode(data); |
| if (root == NULL) return newNode; |
|  |
| struct Node\* parent = NULL; |
| struct Node\* current = root; |
| while (current != NULL) { |
| parent = current; |
| if (data < current->data) |
| current = current->left; |
| else if (data > current->data) |
| current = current->right; |
| else |
| return root; // No duplicates |
| } |
|  |
| if (data < parent->data) |
| parent->left = newNode; |
| else |
| parent->right = newNode; |
|  |
| return root; |
| } |
|  |
| // Recursive BST insertion |
| struct Node\* recursiveInsert(struct Node\* root, int data) { |
| if (root == NULL) return createNode(data); |
|  |
| if (data < root->data) |
| root->left = recursiveInsert(root->left, data); |
| else if (data > root->data) |
| root->right = recursiveInsert(root->right, data); |
|  |
| return root; |
| } |
|  |
| // Utility function to print BST in-order (for verification) |
| void inorderTraversal(struct Node\* root) { |
| if (root != NULL) { |
| inorderTraversal(root->left); |
| printf("%d ", root->data); |
| inorderTraversal(root->right); |
| } |
| } |
|  |
| // Time comparison function for both insertions |
| void compareInsertionTimes(int arrays[5][10], int sizes[5]) { |
| for (int i = 0; i < 5; i++) { |
| printf("\n--- Array %d ---\n", i + 1); |
| struct Node\* root1 = NULL; // For iterative insertions |
| struct Node\* root2 = NULL; // For recursive insertions |
|  |
| // Measure time for iterative insertion |
| clock\_t startIter = clock(); |
| for (int j = 0; j < sizes[i]; j++) { |
| root1 = iterativeInsert(root1, arrays[i][j]); |
| } |
| clock\_t endIter = clock(); |
| double timeIter = ((double)(endIter - startIter)) / CLOCKS\_PER\_SEC; |
|  |
| // Measure time for recursive insertion |
| clock\_t startRecur = clock(); |
| for (int j = 0; j < sizes[i]; j++) { |
| root2 = recursiveInsert(root2, arrays[i][j]); |
| } |
| clock\_t endRecur = clock(); |
| double timeRecur = ((double)(endRecur - startRecur)) / CLOCKS\_PER\_SEC; |
|  |
| printf("Iterative Insertion Time: %f seconds\n", timeIter); |
| printf("Recursive Insertion Time: %f seconds\n", timeRecur); |
|  |
| // Optional: Print BST (for verification) |
| printf("In-order traversal (Iterative): "); |
| inorderTraversal(root1); |
| printf("\nIn-order traversal (Recursive): "); |
| inorderTraversal(root2); |
| printf("\n"); |
| } |
| } |
|  |
| // Driver function |
| int main() { |
| // Define five sample arrays |
| int arrays[5][10] = { |
| {50, 30, 20, 40, 70, 60, 80},  // 7 elements |
| {10, 20, 30, 40, 50, 60, 70, 80, 90}, // 9 elements |
| {25, 15, 50, 10, 22, 35, 70, 40, 80}, // 9 elements |
| {100, 90, 80, 70, 60}, // 5 elements |
| {5, 25, 15, 35, 20, 30, 10}  // 7 elements |
| }; |
|  |
| // Define the size of each array |
| int sizes[5] = {7, 9, 9, 5, 7}; |
|  |
| // Compare insertion times |
| compareInsertionTimes(arrays, sizes); |
|  |
| return 0; |
| } |

### OUTPUT:

### GRAPH:

## LAB-2: Merge Sort and Quick Sort 🔄

**Topic:**

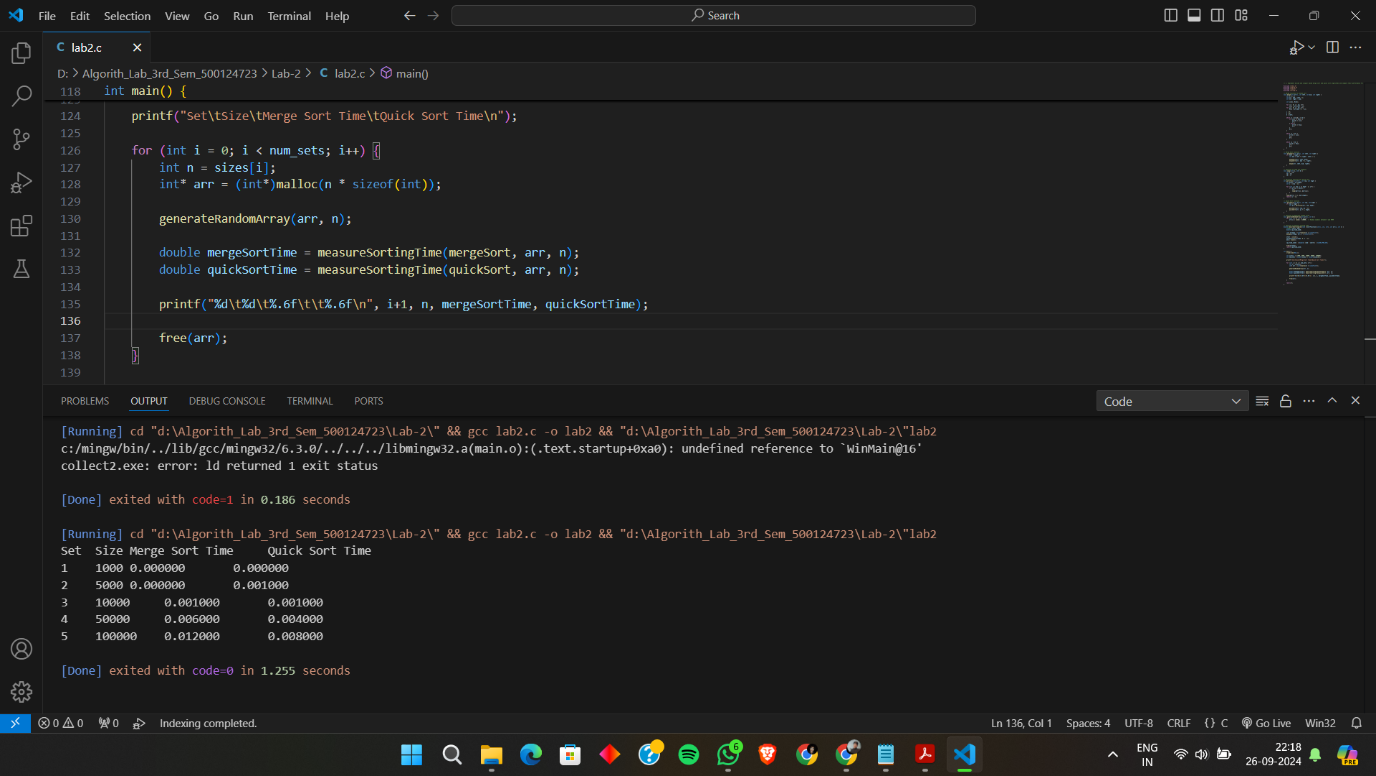
Implement divide and conquer-based Merge Sort and Quick Sort algorithms and compare their performance for the same set of elements.

* **Description**: This lab focuses on two important sorting algorithms: Merge Sort and Quick Sort. By implementing both, we compare their performance when applied to the same dataset.

### CODE:

|  |
| --- |
| #include <stdio.h> |
| #include <stdlib.h> |
| #include <time.h> |
| #include <string.h> |
|  |
| // Merge function for merge sort |
| void merge(int arr[], int left, int mid, int right) { |
| int i, j, k; |
| int n1 = mid - left + 1; |
| int n2 = right - mid; |
|  |
| int L[n1], R[n2]; |
|  |
| for (i = 0; i < n1; i++) |
| L[i] = arr[left + i]; |
| for (j = 0; j < n2; j++) |
| R[j] = arr[mid + 1 + j]; |
|  |
| i = 0; |
| j = 0; |
| k = left; |
|  |
| while (i < n1 && j < n2) { |
| if (L[i] <= R[j]) { |
| arr[k] = L[i]; |
| i++; |
| } else { |
| arr[k] = R[j]; |
| j++; |
| } |
| k++; |
| } |
|  |
| while (i < n1) { |
| arr[k] = L[i]; |
| i++; |
| k++; |
| } |
|  |
| while (j < n2) { |
| arr[k] = R[j]; |
| j++; |
| k++; |
| } |
| } |
|  |
| // Merge Sort function |
| void mergeSort(int arr[], int left, int right) { |
| if (left < right) { |
| int mid = left + (right - left) / 2; |
|  |
| mergeSort(arr, left, mid); |
| mergeSort(arr, mid + 1, right); |
|  |
| merge(arr, left, mid, right); |
| } |
| } |
|  |
| // Function to swap two elements |
| void swap(int\* a, int\* b) { |
| int t = \*a; |
| \*a = \*b; |
| \*b = t; |
| } |
|  |
| // Partition function for quick sort |
| int partition(int arr[], int low, int high) { |
| int pivot = arr[high]; |
| int i = (low - 1); |
|  |
| for (int j = low; j <= high - 1; j++) { |
| if (arr[j] < pivot) { |
| i++; |
| swap(&arr[i], &arr[j]); |
| } |
| } |
| swap(&arr[i + 1], &arr[high]); |
| return (i + 1); |
| } |
|  |
| // Quick Sort function |
| void quickSort(int arr[], int low, int high) { |
| if (low < high) { |
| int pi = partition(arr, low, high); |
|  |
| quickSort(arr, low, pi - 1); |
| quickSort(arr, pi + 1, high); |
| } |
| } |
|  |
| // Function to generate random array |
| void generateRandomArray(int arr[], int n) { |
| for (int i = 0; i < n; i++) { |
| arr[i] = rand() % 10000;  // Random numbers between 0 and 9999 |
| } |
| } |
|  |
| // Function to measure sorting time |
| double measureSortingTime(void (\*sortFunction)(int[], int, int), int arr[], int n) { |
| clock\_t start, end; |
| double cpu\_time\_used; |
|  |
| int\* arrCopy = (int\*)malloc(n \* sizeof(int)); |
| memcpy(arrCopy, arr, n \* sizeof(int)); |
|  |
| start = clock(); |
| sortFunction(arrCopy, 0, n - 1); |
| end = clock(); |
|  |
| cpu\_time\_used = ((double) (end - start)) / CLOCKS\_PER\_SEC; |
|  |
| free(arrCopy); |
| return cpu\_time\_used; |
| } |
|  |
| int main() { |
| srand(time(NULL)); |
|  |
| int sizes[] = {1000, 5000, 10000, 50000, 100000}; |
| int num\_sets = sizeof(sizes) / sizeof(sizes[0]); |
|  |
| printf("Set\tSize\tMerge Sort Time\tQuick Sort Time\n"); |
|  |
| for (int i = 0; i < num\_sets; i++) { |
| int n = sizes[i]; |
| int\* arr = (int\*)malloc(n \* sizeof(int)); |
|  |
| generateRandomArray(arr, n); |
|  |
| double mergeSortTime = measureSortingTime(mergeSort, arr, n); |
| double quickSortTime = measureSortingTime(quickSort, arr, n); |
|  |
| printf("%d\t%d\t%.6f\t\t%.6f\n", i+1, n, mergeSortTime, quickSortTime); |
|  |
| free(arr); |
| } |
|  |
| return 0; |
| } |

### OUTPUT:



### GRAPH:



## LAB-3: Matrix Multiplication ➕➗

**Topic:**

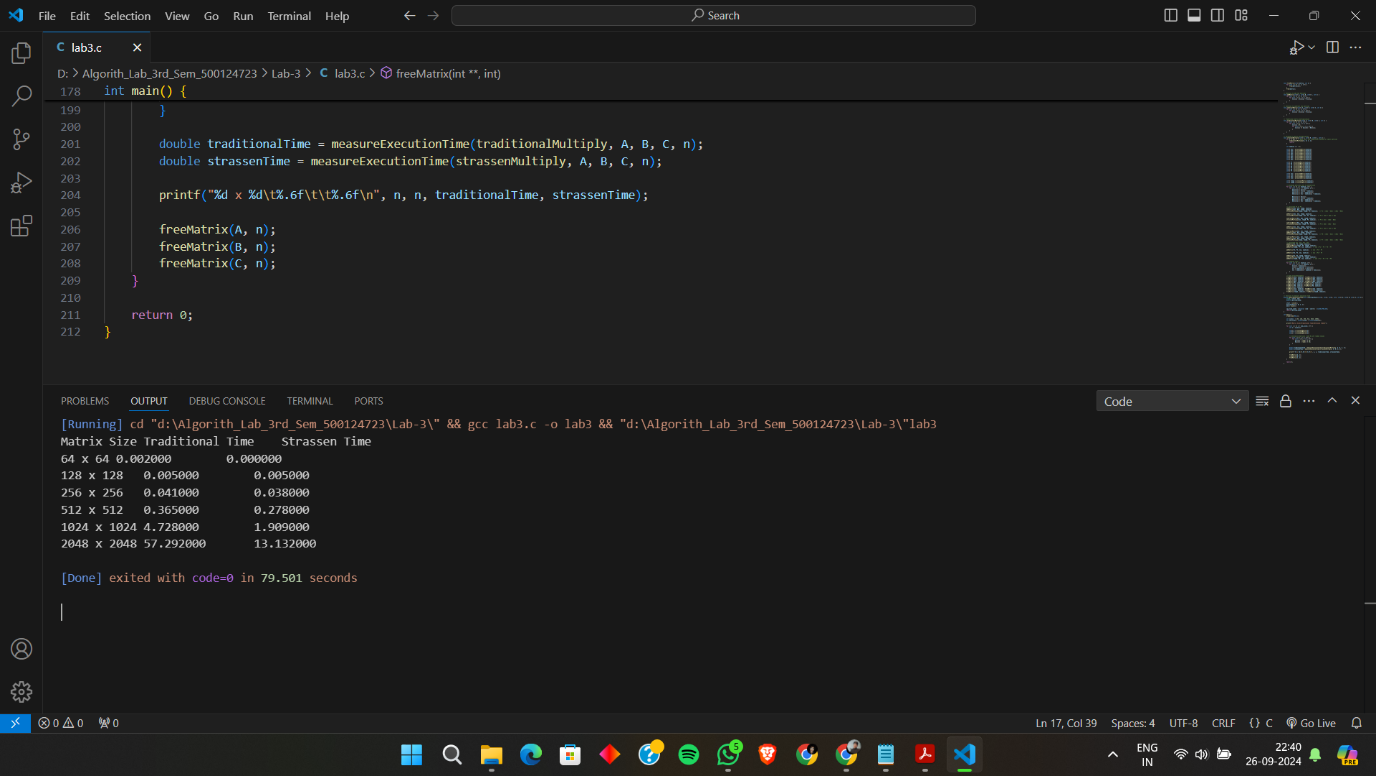
Compare the performance of Strassen's method of matrix multiplication with the traditional way of matrix multiplication.

* **Description**: In this lab, we explore matrix multiplication using Strassen's method and the traditional method, comparing their efficiency for different matrix sizes.

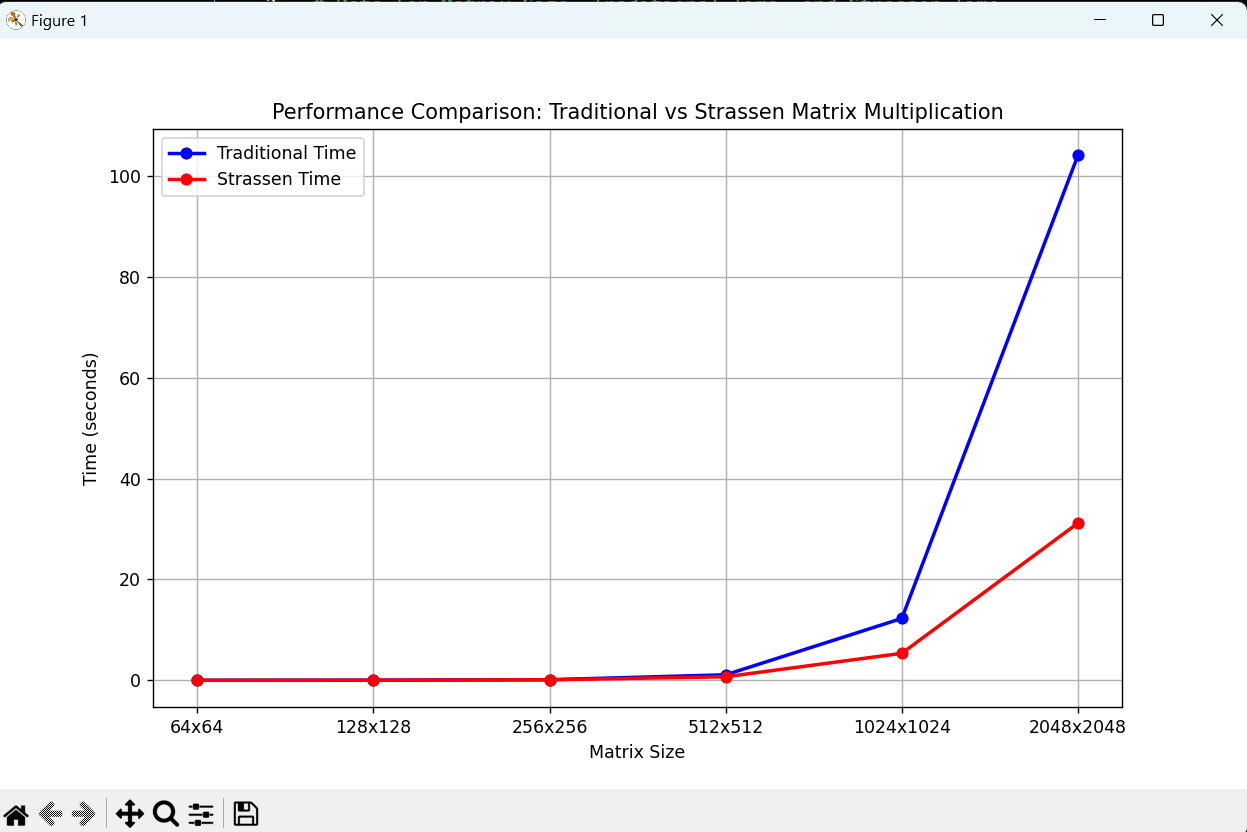
### CODE:

|  |
| --- |
| #include <stdio.h> |
| #include <stdlib.h> |
| #include <time.h> |
|  |
| // Function to allocate memory for a matrix |
| int\*\* allocateMatrix(int n) { |
| int\*\* matrix = (int\*\*)malloc(n \* sizeof(int\*)); |
| for (int i = 0; i < n; i++) { |
| matrix[i] = (int\*)malloc(n \* sizeof(int)); |
| } |
| return matrix; |
| } |
|  |
| // Function to free memory of a matrix |
| void freeMatrix(int\*\* matrix, int n) { |
| for (int i = 0; i < n; i++) { |
| free(matrix[i]); |
| } |
| free(matrix); |
| } |
|  |
| // Function to add two matrices |
| void addMatrix(int\*\* A, int\*\* B, int\*\* C, int n) { |
| for (int i = 0; i < n; i++) { |
| for (int j = 0; j < n; j++) { |
| C[i][j] = A[i][j] + B[i][j]; |
| } |
| } |
| } |
|  |
| // Function to subtract two matrices |
| void subtractMatrix(int\*\* A, int\*\* B, int\*\* C, int n) { |
| for (int i = 0; i < n; i++) { |
| for (int j = 0; j < n; j++) { |
| C[i][j] = A[i][j] - B[i][j]; |
| } |
| } |
| } |
|  |
| // Traditional matrix multiplication |
| void traditionalMultiply(int\*\* A, int\*\* B, int\*\* C, int n) { |
| for (int i = 0; i < n; i++) { |
| for (int j = 0; j < n; j++) { |
| C[i][j] = 0; |
| for (int k = 0; k < n; k++) { |
| C[i][j] += A[i][k] \* B[k][j]; |
| } |
| } |
| } |
| } |
|  |
| // Strassen's matrix multiplication |
| void strassenMultiply(int\*\* A, int\*\* B, int\*\* C, int n) { |
| if (n <= 64) {  // Base case: use traditional method for small matrices |
| traditionalMultiply(A, B, C, n); |
| return; |
| } |
|  |
| int newSize = n / 2; |
|  |
| int\*\* A11 = allocateMatrix(newSize); |
| int\*\* A12 = allocateMatrix(newSize); |
| int\*\* A21 = allocateMatrix(newSize); |
| int\*\* A22 = allocateMatrix(newSize); |
| int\*\* B11 = allocateMatrix(newSize); |
| int\*\* B12 = allocateMatrix(newSize); |
| int\*\* B21 = allocateMatrix(newSize); |
| int\*\* B22 = allocateMatrix(newSize); |
|  |
| int\*\* P1 = allocateMatrix(newSize); |
| int\*\* P2 = allocateMatrix(newSize); |
| int\*\* P3 = allocateMatrix(newSize); |
| int\*\* P4 = allocateMatrix(newSize); |
| int\*\* P5 = allocateMatrix(newSize); |
| int\*\* P6 = allocateMatrix(newSize); |
| int\*\* P7 = allocateMatrix(newSize); |
|  |
| int\*\* C11 = allocateMatrix(newSize); |
| int\*\* C12 = allocateMatrix(newSize); |
| int\*\* C21 = allocateMatrix(newSize); |
| int\*\* C22 = allocateMatrix(newSize); |
|  |
| int\*\* tempA = allocateMatrix(newSize); |
| int\*\* tempB = allocateMatrix(newSize); |
|  |
| // Dividing matrices into 4 sub-matrices |
| for (int i = 0; i < newSize; i++) { |
| for (int j = 0; j < newSize; j++) { |
| A11[i][j] = A[i][j]; |
| A12[i][j] = A[i][j + newSize]; |
| A21[i][j] = A[i + newSize][j]; |
| A22[i][j] = A[i + newSize][j + newSize]; |
|  |
| B11[i][j] = B[i][j]; |
| B12[i][j] = B[i][j + newSize]; |
| B21[i][j] = B[i + newSize][j]; |
| B22[i][j] = B[i + newSize][j + newSize]; |
| } |
| } |
|  |
| // Calculate P1 to P7 |
| addMatrix(A11, A22, tempA, newSize); |
| addMatrix(B11, B22, tempB, newSize); |
| strassenMultiply(tempA, tempB, P1, newSize);  // P1 = (A11 + A22) \* (B11 + B22) |
|  |
| addMatrix(A21, A22, tempA, newSize); |
| strassenMultiply(tempA, B11, P2, newSize);  // P2 = (A21 + A22) \* B11 |
|  |
| subtractMatrix(B12, B22, tempB, newSize); |
| strassenMultiply(A11, tempB, P3, newSize);  // P3 = A11 \* (B12 - B22) |
|  |
| subtractMatrix(B21, B11, tempB, newSize); |
| strassenMultiply(A22, tempB, P4, newSize);  // P4 = A22 \* (B21 - B11) |
|  |
| addMatrix(A11, A12, tempA, newSize); |
| strassenMultiply(tempA, B22, P5, newSize);  // P5 = (A11 + A12) \* B22 |
|  |
| subtractMatrix(A21, A11, tempA, newSize); |
| addMatrix(B11, B12, tempB, newSize); |
| strassenMultiply(tempA, tempB, P6, newSize);  // P6 = (A21 - A11) \* (B11 + B12) |
|  |
| subtractMatrix(A12, A22, tempA, newSize); |
| addMatrix(B21, B22, tempB, newSize); |
| strassenMultiply(tempA, tempB, P7, newSize);  // P7 = (A12 - A22) \* (B21 + B22) |
|  |
| // Calculate C11, C12, C21, C22 |
| addMatrix(P1, P4, tempA, newSize); |
| subtractMatrix(tempA, P5, tempB, newSize); |
| addMatrix(tempB, P7, C11, newSize);  // C11 = P1 + P4 - P5 + P7 |
|  |
| addMatrix(P3, P5, C12, newSize);  // C12 = P3 + P5 |
|  |
| addMatrix(P2, P4, C21, newSize);  // C21 = P2 + P4 |
|  |
| addMatrix(P1, P3, tempA, newSize); |
| subtractMatrix(tempA, P2, tempB, newSize); |
| addMatrix(tempB, P6, C22, newSize);  // C22 = P1 + P3 - P2 + P6 |
|  |
| // Grouping into C |
| for (int i = 0; i < newSize; i++) { |
| for (int j = 0; j < newSize; j++) { |
| C[i][j] = C11[i][j]; |
| C[i][j + newSize] = C12[i][j]; |
| C[i + newSize][j] = C21[i][j]; |
| C[i + newSize][j + newSize] = C22[i][j]; |
| } |
| } |
|  |
| // Free allocated memory |
| freeMatrix(A11, newSize); freeMatrix(A12, newSize); |
| freeMatrix(A21, newSize); freeMatrix(A22, newSize); |
| freeMatrix(B11, newSize); freeMatrix(B12, newSize); |
| freeMatrix(B21, newSize); freeMatrix(B22, newSize); |
| freeMatrix(P1, newSize); freeMatrix(P2, newSize); |
| freeMatrix(P3, newSize); freeMatrix(P4, newSize); |
| freeMatrix(P5, newSize); freeMatrix(P6, newSize); |
| freeMatrix(P7, newSize); |
| freeMatrix(C11, newSize); freeMatrix(C12, newSize); |
| freeMatrix(C21, newSize); freeMatrix(C22, newSize); |
| freeMatrix(tempA, newSize); freeMatrix(tempB, newSize); |
| } |
|  |
| // Function to measure execution time |
| double measureExecutionTime(void (\*multiplyFunc)(int\*\*, int\*\*, int\*\*, int), int\*\* A, int\*\* B, int\*\* C, int n) { |
| clock\_t start, end; |
| double cpu\_time\_used; |
|  |
| start = clock(); |
| multiplyFunc(A, B, C, n); |
| end = clock(); |
|  |
| cpu\_time\_used = ((double) (end - start)) / CLOCKS\_PER\_SEC; |
| return cpu\_time\_used; |
| } |
|  |
| int main() { |
| srand(time(NULL)); |
|  |
| int sizes[] = {64, 128, 256, 512, 1024, 2048}; |
| int num\_sizes = sizeof(sizes) / sizeof(sizes[0]); |
|  |
| printf("Matrix Size\tTraditional Time\tStrassen Time\n"); |
|  |
| for (int i = 0; i < num\_sizes; i++) { |
| int n = sizes[i]; |
|  |
| int\*\* A = allocateMatrix(n); |
| int\*\* B = allocateMatrix(n); |
| int\*\* C = allocateMatrix(n); |
|  |
| // Initialize matrices A and B with random values |
| for (int j = 0; j < n; j++) { |
| for (int k = 0; k < n; k++) { |
| A[j][k] = rand() % 10; |
| B[j][k] = rand() % 10; |
| } |
| } |
|  |
| double traditionalTime = measureExecutionTime(traditionalMultiply, A, B, C, n); |
| double strassenTime = measureExecutionTime(strassenMultiply, A, B, C, n); |
|  |
| printf("%d x %d\t%.6f\t\t%.6f\n", n, n, traditionalTime, strassenTime); |
|  |
| freeMatrix(A, n); |
| freeMatrix(B, n); |
| freeMatrix(C, n); |
| } |
|  |
| return 0; |
| } |

### OUTPUT:



### GRAPH:



## LAB-4: Activity Selection Problem 🎯

**Topic:**

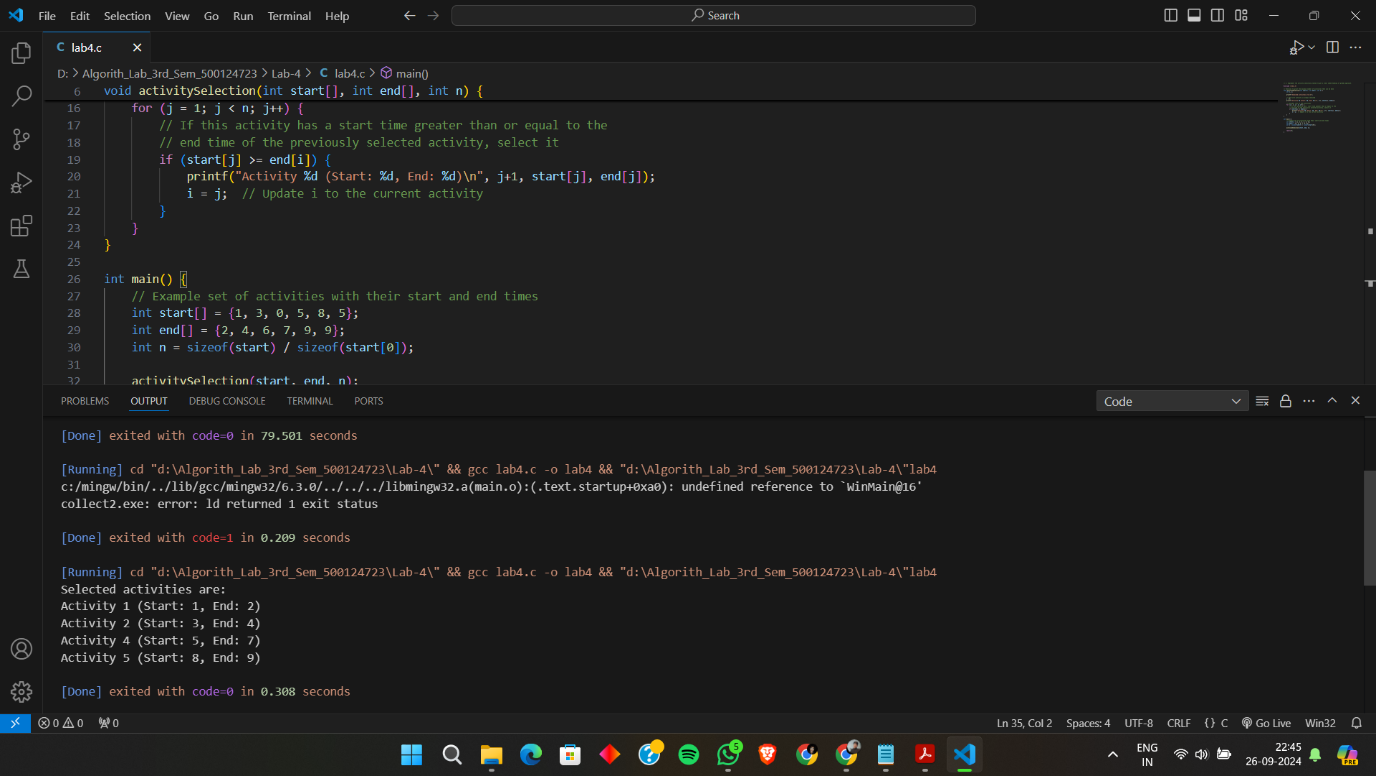
Implement the activity selection problem to get a clear understanding of the greedy approach.

* **Description**: The activity selection problem is implemented to showcase the greedy algorithm approach, which helps in selecting the maximum number of activities without overlapping.

### CODE:

|  |
| --- |
| #include <stdio.h> |
|  |
| // Function to print the maximum number of activities that can be done |
| void activitySelection(int start[], int end[], int n) { |
| int i, j; |
|  |
| printf("Selected activities are:\n"); |
|  |
| // The first activity is always selected |
| i = 0; |
| printf("Activity %d (Start: %d, End: %d)\n", i+1, start[i], end[i]); |
|  |
| // Consider rest of the activities |
| for (j = 1; j < n; j++) { |
| // If this activity has a start time greater than or equal to the |
| // end time of the previously selected activity, select it |
| if (start[j] >= end[i]) { |
| printf("Activity %d (Start: %d, End: %d)\n", j+1, start[j], end[j]); |
| i = j;  // Update i to the current activity |
| } |
| } |
| } |
|  |
| int main() { |
| // Example set of activities with their start and end times |
| int start[] = {1, 3, 0, 5, 8, 5}; |
| int end[] = {2, 4, 6, 7, 9, 9}; |
| int n = sizeof(start) / sizeof(start[0]); |
|  |
| activitySelection(start, end, n); |
|  |
| return 0; |
| } |

### OUTPUT:



## LAB-5: Matrix Chain Multiplication 🔢

**Topic:**

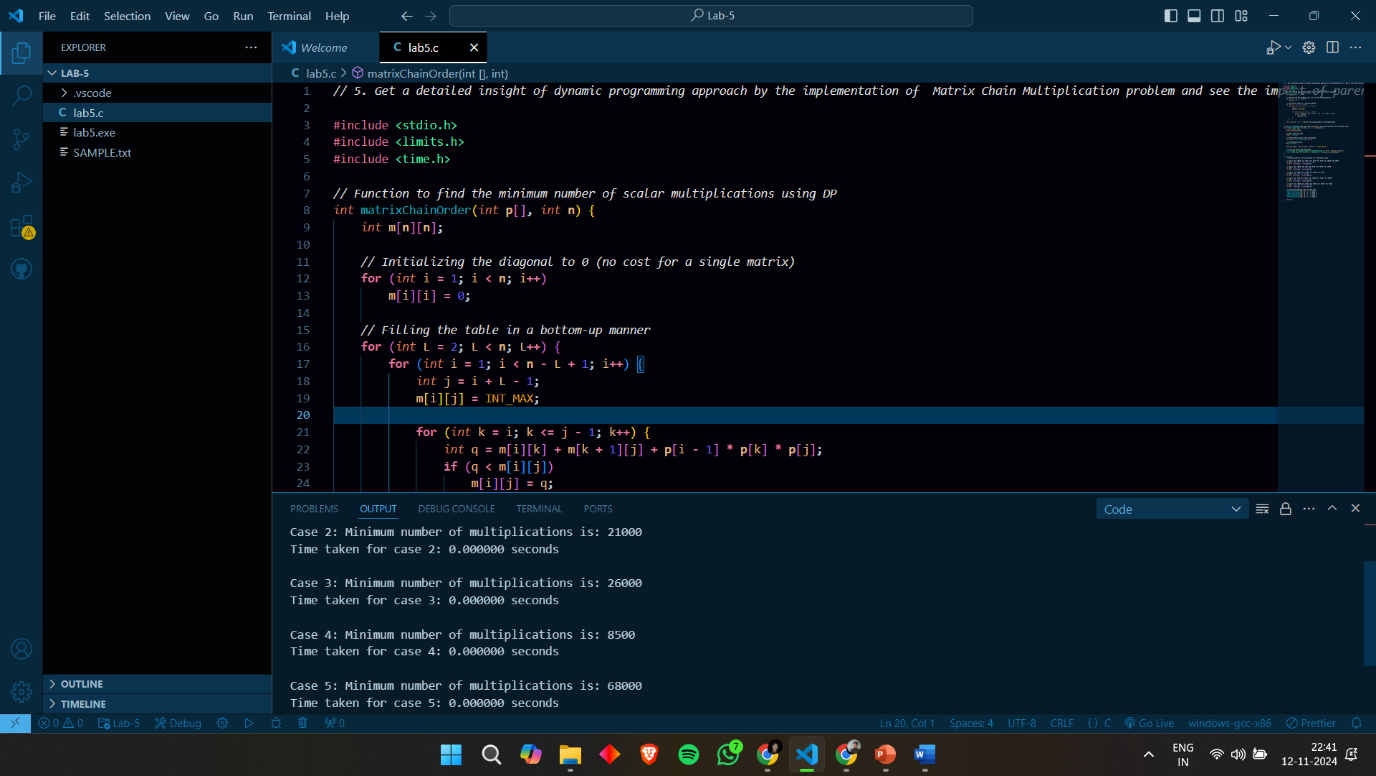
*Get a detailed insight into dynamic programming by implementing the Matrix Chain Multiplication problem and exploring the impact of parenthesis placement on time requirements.*

* **Description**: This lab introduces *dynamic programming* to optimize the computation time in matrix multiplication by adjusting the placement of parentheses. It demonstrates the effect of matrix ordering on the performance of multiplication.

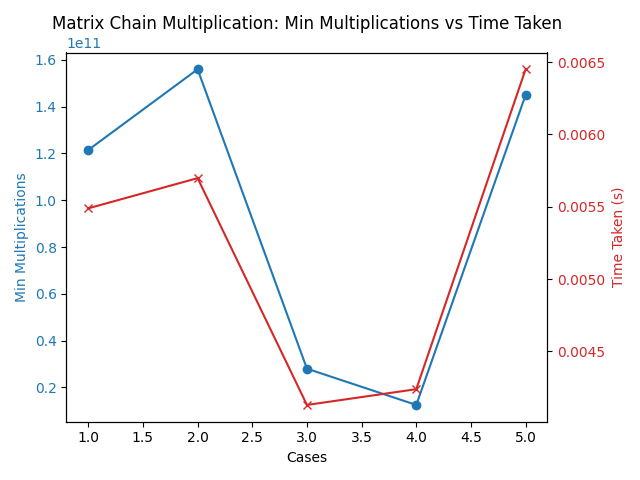
### CODE:

|  |
| --- |
| #include <stdio.h> |
| #include <limits.h> |
| #include <time.h> |
|  |
| // Function to find the minimum number of scalar multiplications using DP |
| int matrixChainOrder(int p[], int n) { |
| int m[n][n]; |
|  |
| // Initializing the diagonal to 0 (no cost for a single matrix) |
| for (int i = 1; i < n; i++) |
| m[i][i] = 0; |
|  |
| // Filling the table in a bottom-up manner |
| for (int L = 2; L < n; L++) { |
| for (int i = 1; i < n - L + 1; i++) { |
| int j = i + L - 1; |
| m[i][j] = INT\_MAX; |
|  |
| for (int k = i; k <= j - 1; k++) { |
| int q = m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j]; |
| if (q < m[i][j]) |
| m[i][j] = q; |
| } |
| } |
| } |
| return m[1][n - 1];  // Return the minimum number of multiplications |
| } |
|  |
| // Function to measure time and execute the matrix chain multiplication for different cases |
| void runMatrixChainOrder(int p[], int n, int case\_num) { |
| clock\_t start, end; |
| double cpu\_time\_used; |
|  |
| // Start measuring time |
| start = clock(); |
|  |
| // Perform matrix chain order calculation |
| int result = matrixChainOrder(p, n); |
|  |
| // Stop measuring time |
| end = clock(); |
|  |
| cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC; |
|  |
| // Print the result and time taken |
| printf("Case %d: Minimum number of multiplications is: %d\n", case\_num, result); |
| printf("Time taken for case %d: %f seconds\n\n", case\_num, cpu\_time\_used); |
| } |
|  |
| int main() { |
| // Define different sets of matrices for 5 different cases |
|  |
| // Case 1: A1: 30x35, A2: 35x15, A3: 15x5, A4: 5x10, A5: 10x20, A6: 20x25 |
| int p1[] = {30, 35, 15, 5, 10, 20, 25}; |
| int n1 = sizeof(p1) / sizeof(p1[0]); |
|  |
| // Case 2: A1: 10x30, A2: 30x5, A3: 5x60, A4: 60x40, A5: 40x30 |
| int p2[] = {10, 30, 5, 60, 40, 30}; |
| int n2 = sizeof(p2) / sizeof(p2[0]); |
|  |
| // Case 3: A1: 40x20, A2: 20x30, A3: 30x10, A4: 10x30 |
| int p3[] = {40, 20, 30, 10, 30}; |
| int n3 = sizeof(p3) / sizeof(p3[0]); |
|  |
| // Case 4: A1: 5x10, A2: 10x15, A3: 15x20, A4: 20x25, A5: 25x30 |
| int p4[] = {5, 10, 15, 20, 25, 30}; |
| int n4 = sizeof(p4) / sizeof(p4[0]); |
|  |
| // Case 5: A1: 10x20, A2: 20x30, A3: 30x40, A4: 40x50, A5: 50x60 |
| int p5[] = {10, 20, 30, 40, 50, 60}; |
| int n5 = sizeof(p5) / sizeof(p5[0]); |
|  |
| // Execute and measure time for each case |
| runMatrixChainOrder(p1, n1, 1);  // Case 1 |
| runMatrixChainOrder(p2, n2, 2);  // Case 2 |
| runMatrixChainOrder(p3, n3, 3);  // Case 3 |
| runMatrixChainOrder(p4, n4, 4);  // Case 4 |
| runMatrixChainOrder(p5, n5, 5);  // Case 5 |
|  |
| return 0; |
| } |

### OUTPUT:



### GRAPH:



## LAB-6: Shortest Path Algorithms 🚦

**Topic:**

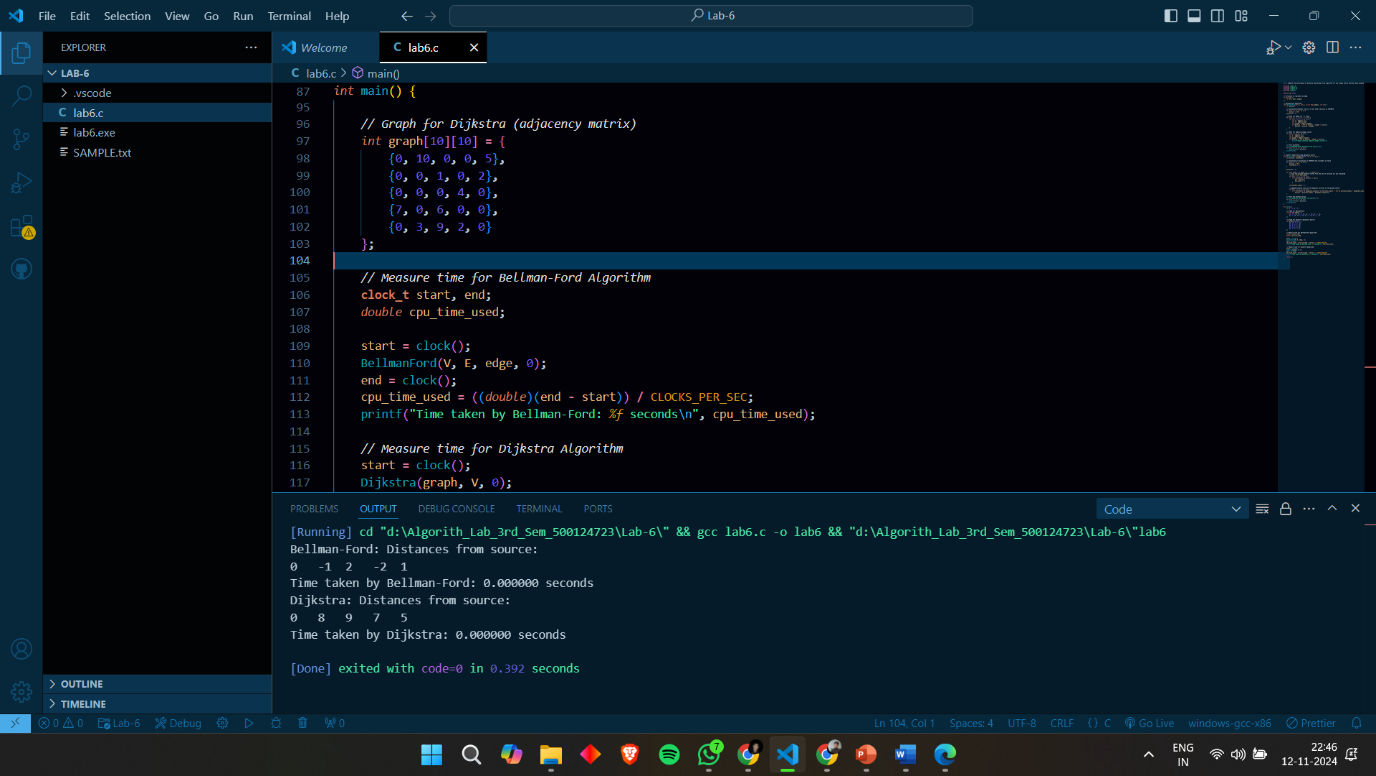
*Compare the performance of Dijkstra's and Bellman-Ford algorithms for the single-source shortest path problem.*

* **Description**: This experiment covers two essential algorithms for finding the shortest path in a weighted graph: **Dijkstra's** (*greedy approach*) and **Bellman-Ford** (*dynamic programming approach*). The comparison provides insights into their efficiency and best-use scenarios.

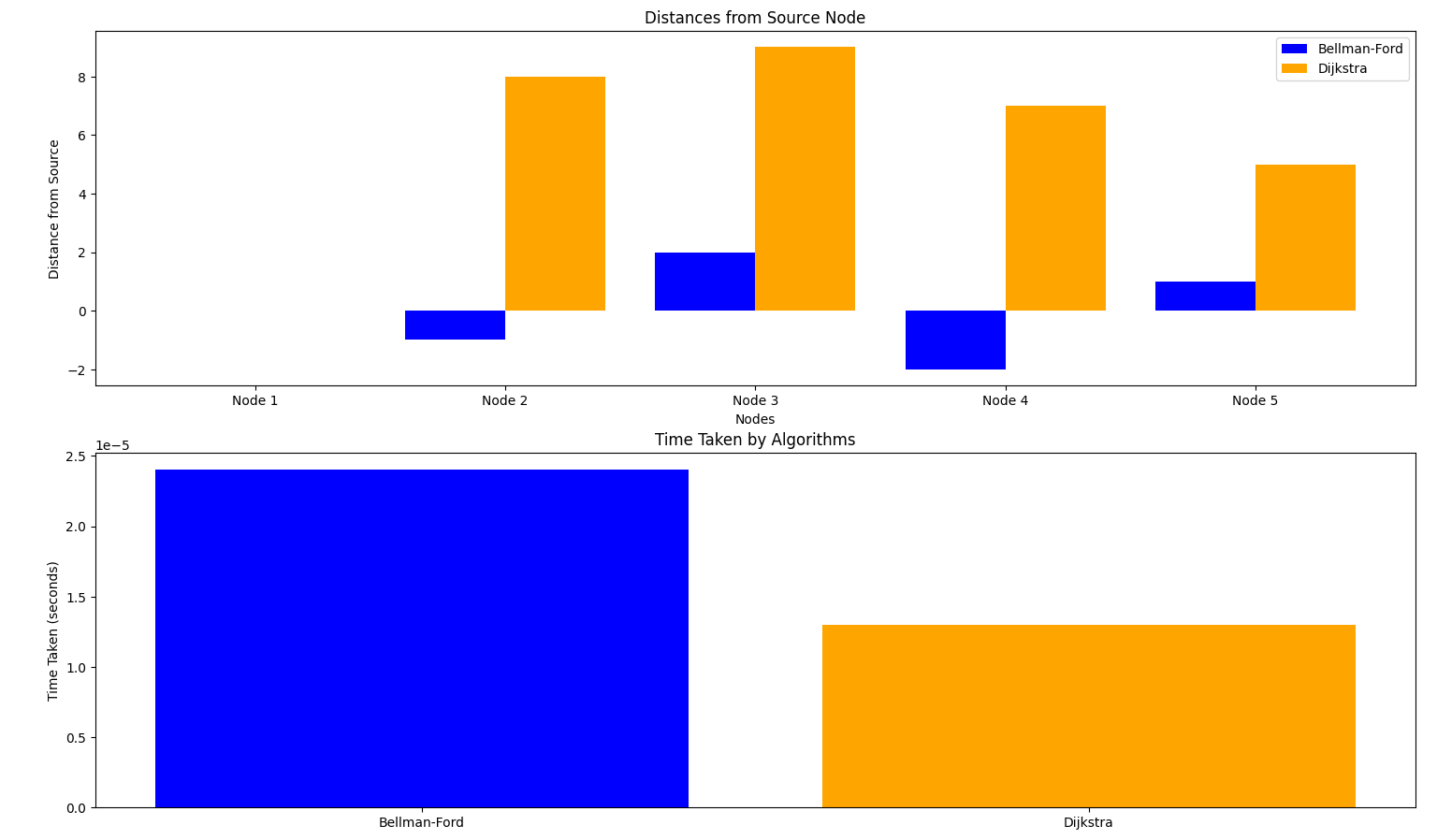
### CODE:

|  |
| --- |
| #include <stdio.h> |
| #include <stdlib.h> |
| #include <limits.h> |
| #include <time.h> |
|  |
| #define INF 99999 |
|  |
| // Structure to represent an edge |
| struct Edge { |
| int src, dest, weight; |
| }; |
|  |
| // Bellman-Ford Algorithm |
| void BellmanFord(int V, int E, struct Edge edge[], int src) { |
| int dist[V]; |
|  |
| // Initialize distances from src to all other vertices as INFINITE |
| for (int i = 0; i < V; i++) |
| dist[i] = INF; |
| dist[src] = 0; |
|  |
| // Relax all edges |V| - 1 times |
| for (int i = 1; i <= V - 1; i++) { |
| for (int j = 0; j < E; j++) { |
| int u = edge[j].src; |
| int v = edge[j].dest; |
| int weight = edge[j].weight; |
| if (dist[u] != INF && dist[u] + weight < dist[v]) |
| dist[v] = dist[u] + weight; |
| } |
| } |
|  |
| // Check for negative-weight cycles |
| for (int i = 0; i < E; i++) { |
| int u = edge[i].src; |
| int v = edge[i].dest; |
| int weight = edge[i].weight; |
| if (dist[u] != INF && dist[u] + weight < dist[v]) |
| printf("Graph contains negative weight cycle\n"); |
| } |
|  |
| // Print distances |
| printf("Bellman-Ford: Distances from source:\n"); |
| for (int i = 0; i < V; i++) |
| printf("%d\t", dist[i]); |
| printf("\n"); |
| } |
|  |
| // Dijkstra Algorithm using adjacency matrix |
| void Dijkstra(int graph[10][10], int V, int src) { |
| int dist[V], visited[V]; |
|  |
| // Initialize all distances as INFINITE and visited[] as false |
| for (int i = 0; i < V; i++) { |
| dist[i] = INF; |
| visited[i] = 0; |
| } |
|  |
| dist[src] = 0; |
|  |
| for (int count = 0; count < V - 1; count++) { |
| // Pick the minimum distance vertex from the set of vertices not yet processed |
| int min = INF, min\_index; |
| for (int v = 0; v < V; v++) |
| if (!visited[v] && dist[v] <= min) { |
| min = dist[v]; |
| min\_index = v; |
| } |
|  |
| visited[min\_index] = 1; |
|  |
| // Update distance value of the adjacent vertices of the picked vertex |
| for (int v = 0; v < V; v++) |
| if (!visited[v] && graph[min\_index][v] && dist[min\_index] != INF && dist[min\_index] + graph[min\_index][v] < dist[v]) |
| dist[v] = dist[min\_index] + graph[min\_index][v]; |
| } |
|  |
| // Print the distance array |
| printf("Dijkstra: Distances from source:\n"); |
| for (int i = 0; i < V; i++) |
| printf("%d\t", dist[i]); |
| printf("\n"); |
| } |
|  |
| int main() { |
| int V = 5, E = 8; |
|  |
| // Graph for Bellman-Ford |
| struct Edge edge[] = { |
| {0, 1, -1}, {0, 2, 4}, {1, 2, 3}, {1, 3, 2}, |
| {1, 4, 2}, {3, 2, 5}, {3, 1, 1}, {4, 3, -3} |
| }; |
|  |
| // Graph for Dijkstra (adjacency matrix) |
| int graph[10][10] = { |
| {0, 10, 0, 0, 5}, |
| {0, 0, 1, 0, 2}, |
| {0, 0, 0, 4, 0}, |
| {7, 0, 6, 0, 0}, |
| {0, 3, 9, 2, 0} |
| }; |
|  |
| // Measure time for Bellman-Ford Algorithm |
| clock\_t start, end; |
| double cpu\_time\_used; |
|  |
| start = clock(); |
| BellmanFord(V, E, edge, 0); |
| end = clock(); |
| cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC; |
| printf("Time taken by Bellman-Ford: %f seconds\n", cpu\_time\_used); |
|  |
| // Measure time for Dijkstra Algorithm |
| start = clock(); |
| Dijkstra(graph, V, 0); |
| end = clock(); |
| cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC; |
| printf("Time taken by Dijkstra: %f seconds\n", cpu\_time\_used); |
|  |
| return 0; |
| } |

### OUTPUT:



### GRAPH:



## LAB-7: 0/1 Knapsack Problem - Greedy vs Dynamic Programming 🎒

**Topic:**

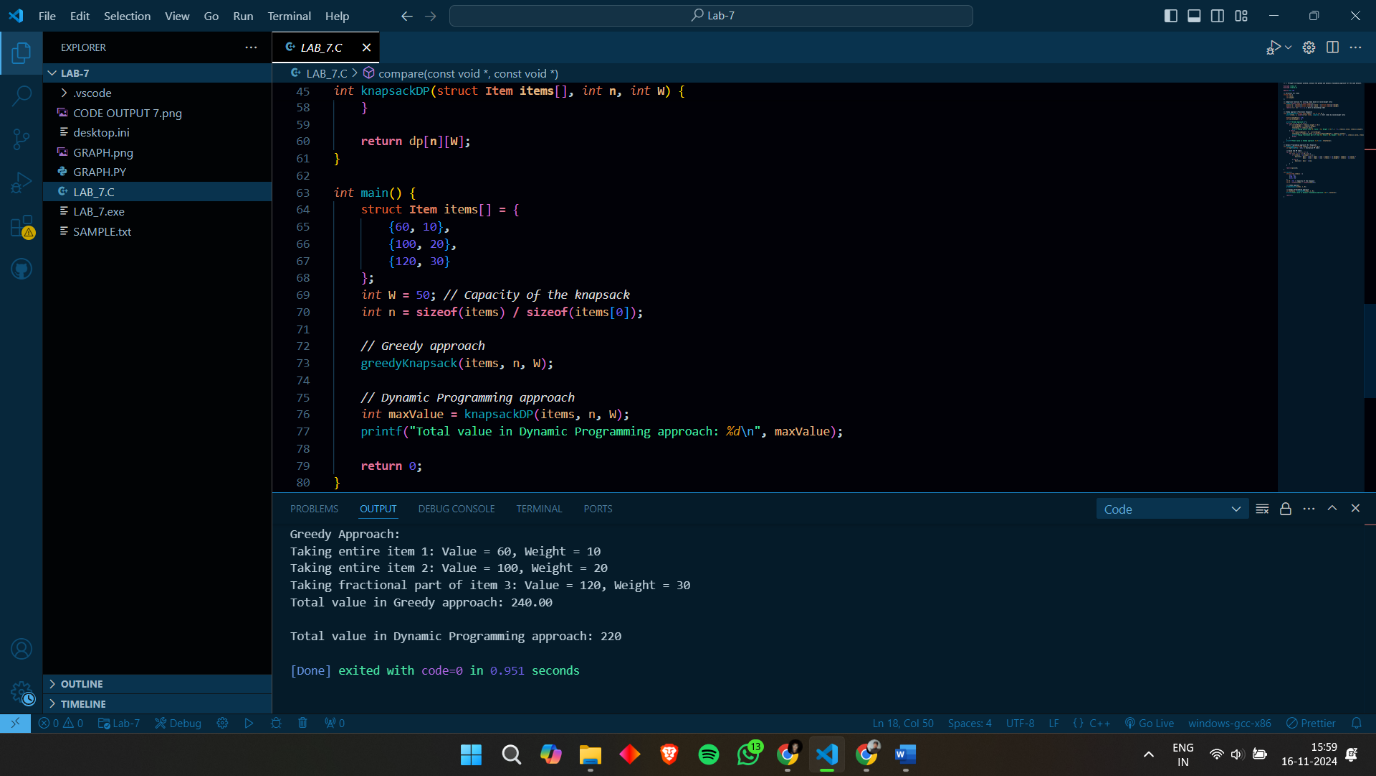
*Through the 0/1 Knapsack problem, analyze both greedy and dynamic programming approaches using the same dataset.*

* **Description**: In this lab, we implement the 0/1 Knapsack problem with both *greedy* and *dynamic programming* techniques. The performance comparison highlights the trade-offs in **accuracy** and **efficiency** between these approaches.

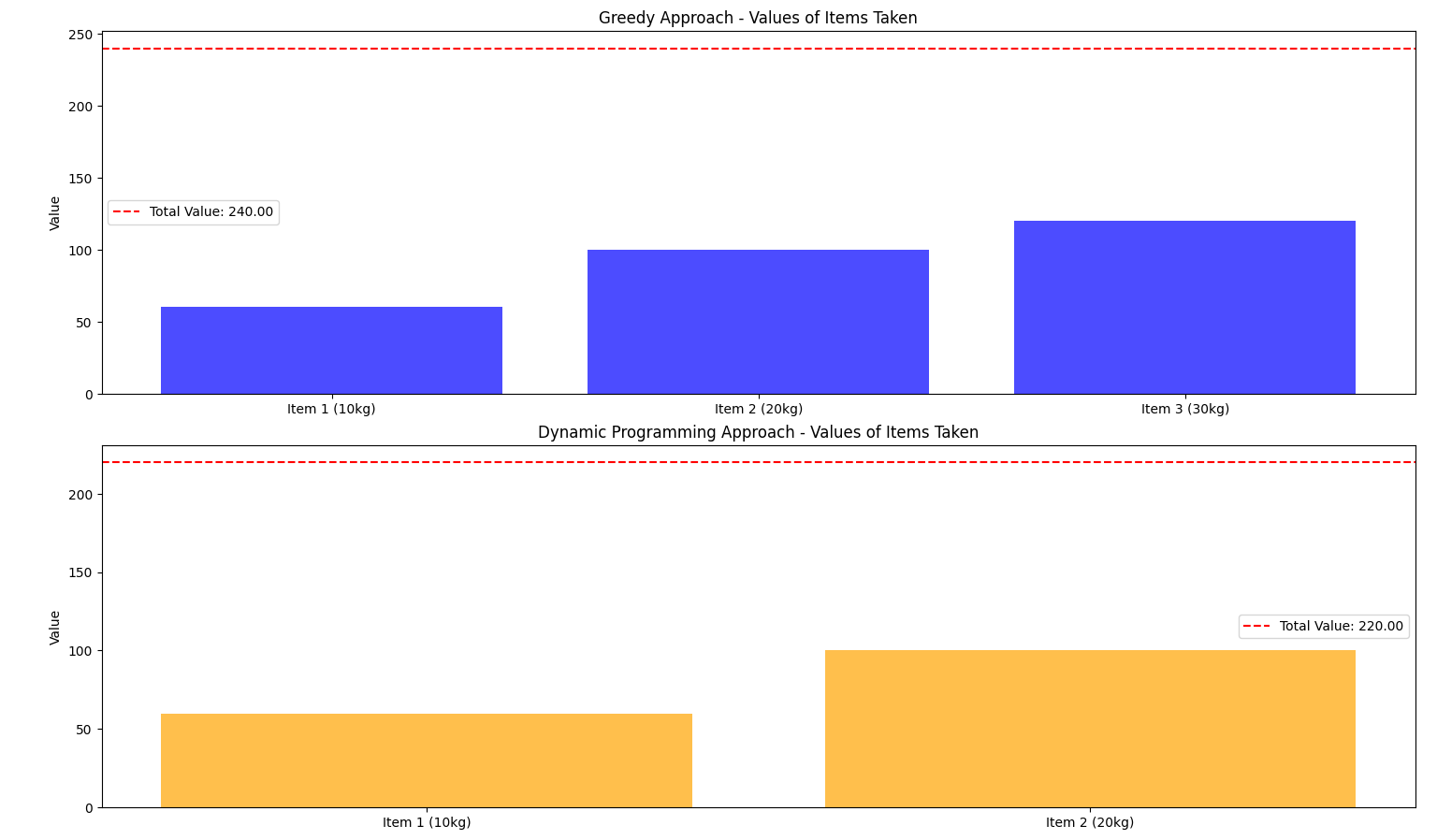
### CODE:

|  |
| --- |
| #include <stdio.h> |
| #include <stdlib.h> |
|  |
| #define MAX 100 |
|  |
| // Structure for items |
| struct Item { |
| int value; |
| int weight; |
| }; |
|  |
| // Comparison function for sorting items based on value/weight ratio |
| int compare(const void\* a, const void\* b) { |
| double r1 = (double)((struct Item\*)a)->value / ((struct Item\*)a)->weight; |
| double r2 = (double)((struct Item\*)b)->value / ((struct Item\*)b)->weight; |
| return (r1 > r2) ? -1 : 1; // Sort in descending order |
| } |
|  |
| // Greedy approach (Fractional Knapsack) |
| void greedyKnapsack(struct Item items[], int n, int W) { |
| qsort(items, n, sizeof(struct Item), compare); // Sort items by value/weight ratio |
|  |
| double totalValue = 0.0; |
| int currentWeight = 0; |
|  |
| printf("Greedy Approach:\n"); |
| for (int i = 0; i < n; i++) { |
| if (currentWeight + items[i].weight <= W) { |
| currentWeight += items[i].weight; |
| totalValue += items[i].value; |
| printf("Taking entire item %d: Value = %d, Weight = %d\n", i + 1, items[i].value, items[i].weight); |
| } else { |
| int remainingWeight = W - currentWeight; |
| totalValue += items[i].value \* ((double)remainingWeight / items[i].weight); |
| printf("Taking fractional part of item %d: Value = %d, Weight = %d\n", i + 1, items[i].value, items[i].weight); |
| break; |
| } |
| } |
| printf("Total value in Greedy approach: %.2f\n\n", totalValue); |
| } |
|  |
| // Dynamic Programming approach (0/1 Knapsack) |
| int knapsackDP(struct Item items[], int n, int W) { |
| int dp[MAX][MAX] = {0}; // Initialize DP table |
|  |
| // Build the DP table |
| for (int i = 1; i <= n; i++) { |
| for (int w = 0; w <= W; w++) { |
| if (items[i - 1].weight <= w) { |
| dp[i][w] = dp[i - 1][w] > (dp[i - 1][w - items[i - 1].weight] + items[i - 1].value) ? |
| dp[i - 1][w] : (dp[i - 1][w - items[i - 1].weight] + items[i - 1].value); |
| } else { |
| dp[i][w] = dp[i - 1][w]; |
| } |
| } |
| } |
|  |
| return dp[n][W]; |
| } |
|  |
| int main() { |
| struct Item items[] = { |
| {60, 10}, |
| {100, 20}, |
| {120, 30} |
| }; |
| int W = 50; // Capacity of the knapsack |
| int n = sizeof(items) / sizeof(items[0]); |
|  |
| // Greedy approach |
| greedyKnapsack(items, n, W); |
|  |
| // Dynamic Programming approach |
| int maxValue = knapsackDP(items, n, W); |
| printf("Total value in Dynamic Programming approach: %d\n", maxValue); |
|  |
| return 0; |
| } |

### OUTPUT:



### GRAPH:



## LAB-8: Subset Sum Problem 🧮

**Topic:**

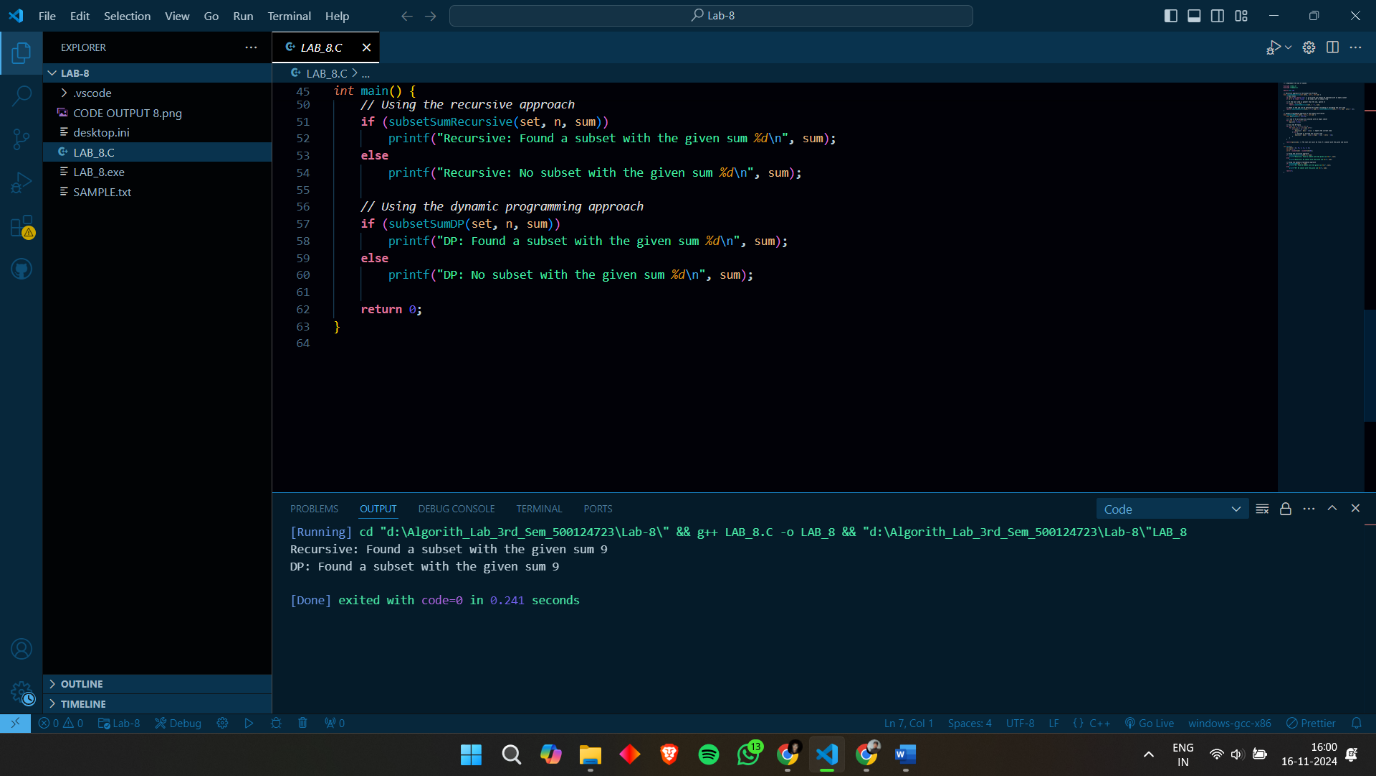
*Implement the subset sum problem to explore decision-making for finding subsets within a target sum.*

* **Description**: This lab focuses on the *subset sum problem*, an important decision-making problem where subsets of elements are selected to achieve a specified sum. The lab illustrates **recursive** solutions and **dynamic programming** approaches for optimization.

### CODE:

|  |
| --- |
| #include <stdio.h> |
| #include <stdbool.h> |
|  |
| #define MAX 100 |
|  |
| // Recursive approach to the Subset Sum Problem |
| bool subsetSumRecursive(int set[], int n, int sum) { |
| // Base Cases |
| if (sum == 0) return true; // A sum of 0 can always be achieved with an empty subset |
| if (n == 0) return false;  // No items left to choose from |
|  |
| // If the last item is greater than the sum, ignore it |
| if (set[n - 1] > sum) |
| return subsetSumRecursive(set, n - 1, sum); |
|  |
| // Check if the sum can be obtained by either including or excluding the last item |
| return subsetSumRecursive(set, n - 1, sum) || subsetSumRecursive(set, n - 1, sum - set[n - 1]); |
| } |
|  |
| // Dynamic Programming approach to the Subset Sum Problem |
| bool subsetSumDP(int set[], int n, int sum) { |
| bool dp[MAX][MAX] = {false}; |
|  |
| // A sum of 0 can always be achieved with an empty subset |
| for (int i = 0; i <= n; i++) |
| dp[i][0] = true; |
|  |
| // Fill the DP table |
| for (int i = 1; i <= n; i++) { |
| for (int j = 1; j <= sum; j++) { |
| if (set[i - 1] > j) { |
| dp[i][j] = dp[i - 1][j]; // Ignore the current item |
| } else { |
| // Include or exclude the current item |
| dp[i][j] = dp[i - 1][j] || dp[i - 1][j - set[i - 1]]; |
| } |
| } |
| } |
|  |
| return dp[n][sum]; // The last cell will be true if a subset with the given sum exists |
| } |
|  |
| int main() { |
| int set[] = {3, 34, 4, 12, 5, 2}; |
| int sum = 9; |
| int n = sizeof(set) / sizeof(set[0]); |
|  |
| // Using the recursive approach |
| if (subsetSumRecursive(set, n, sum)) |
| printf("Recursive: Found a subset with the given sum %d\n", sum); |
| else |
| printf("Recursive: No subset with the given sum %d\n", sum); |
|  |
| // Using the dynamic programming approach |
| if (subsetSumDP(set, n, sum)) |
| printf("DP: Found a subset with the given sum %d\n", sum); |
| else |
| printf("DP: No subset with the given sum %d\n", sum); |
|  |
| return 0; |
| } |

### OUTPUT:



## LAB-9: 0/1 Knapsack - Backtracking vs Branch & Bound 🔗

**Topic:**

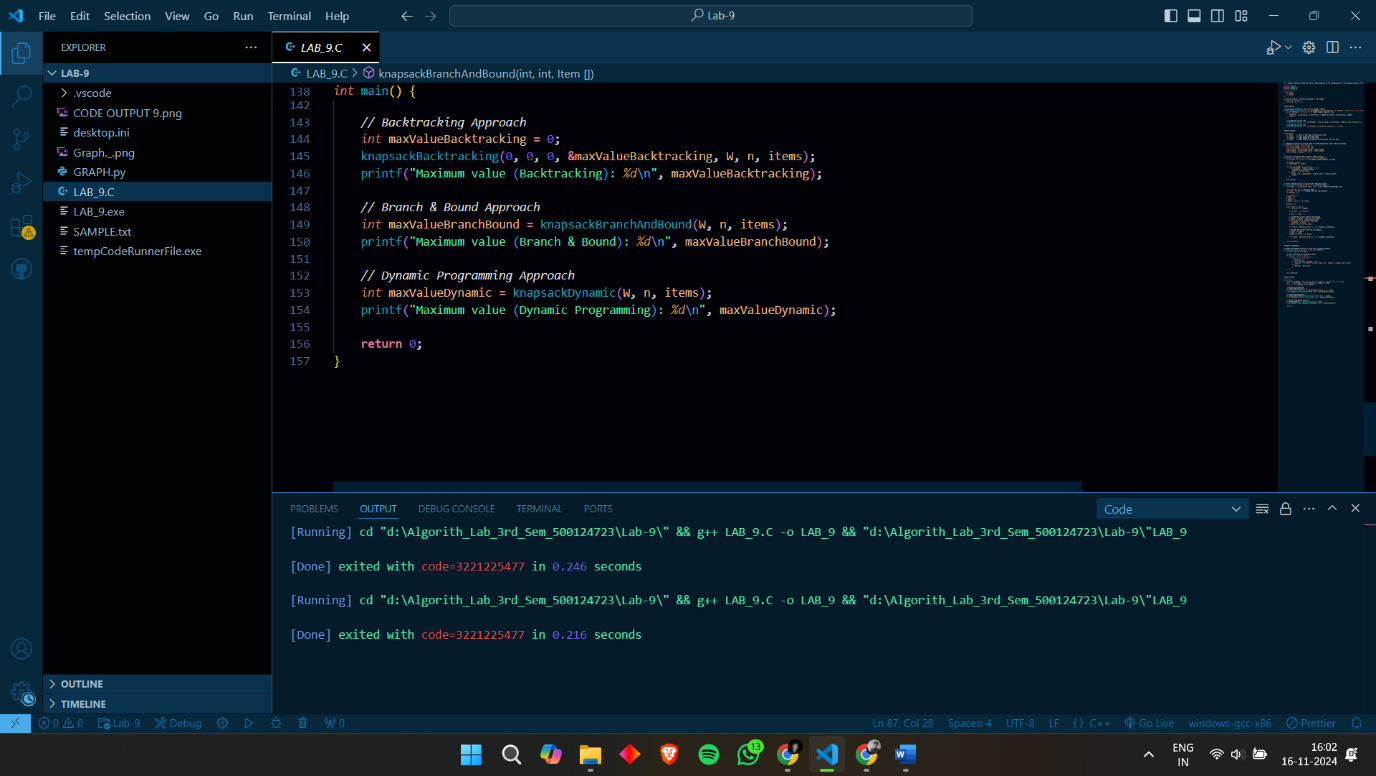
*Compare the Backtracking and Branch & Bound approaches for solving the 0/1 Knapsack problem, alongside a comparison with dynamic programming.*

* **Description**: This lab explores three techniques for solving the 0/1 Knapsack problem: **Backtracking**, **Branch & Bound**, and **dynamic programming**. The comparison helps illustrate the scenarios where each approach is more efficient or effective.

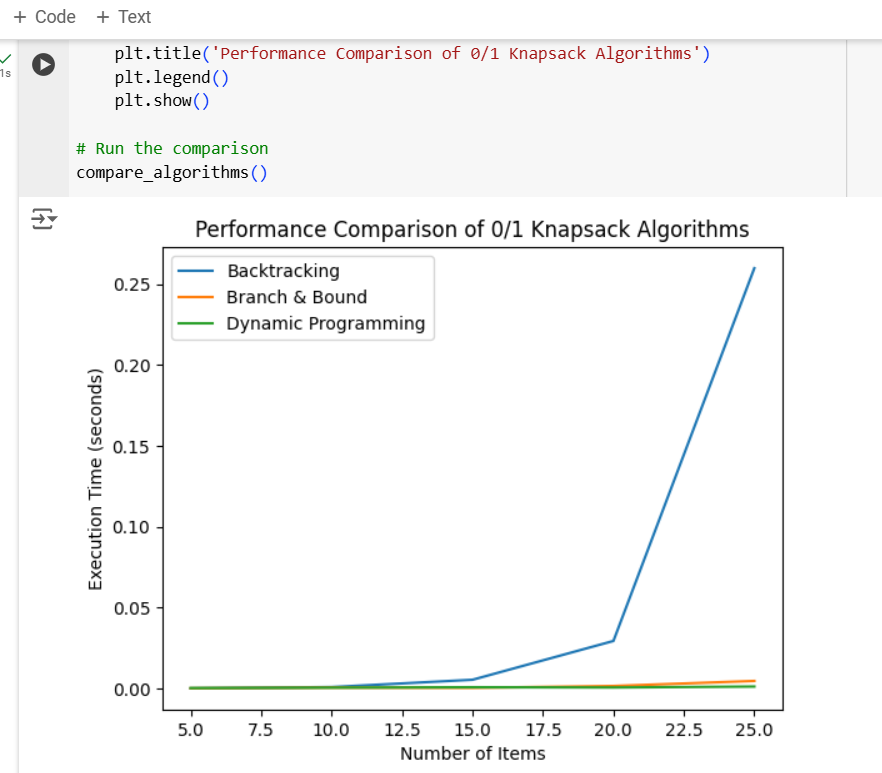
### CODE:

|  |
| --- |
| #include <stdio.h> |
| #include <stdbool.h> |
| #include <stdlib.h> |
|  |
| struct Item { |
| int weight; |
| int value; |
| }; |
|  |
| // Utility function to return the maximum of two integers |
| int max(int a, int b) { |
| return (a > b) ? a : b; |
| } |
|  |
| //Backtracking |
|  |
| // Backtracking function to solve the 0/1 Knapsack problem |
| void knapsackBacktracking(int i, int currentWeight, int currentValue, int \*maxValue, int W, int n, struct Item items[]) { |
| if (currentWeight > W) return; // If weight exceeds capacity, stop |
| if (i == n) { |
| \*maxValue = max(\*maxValue, currentValue); // Update max value if current value is better |
| return; |
| } |
|  |
| // Include the current item |
| knapsackBacktracking(i + 1, currentWeight + items[i].weight, currentValue + items[i].value, maxValue, W, n, items); |
|  |
| // Exclude the current item |
| knapsackBacktracking(i + 1, currentWeight, currentValue, maxValue, W, n, items); |
| } |
|  |
| //Branch & Bound |
|  |
| struct Node { |
| int level;    // Level of the node in the decision tree |
| int value;    // Total value up to this node |
| int weight;   // Total weight up to this node |
| int bound;    // Upper bound of the maximum value possible from this node |
| }; |
|  |
| // Comparator function for sorting items by value/weight ratio (used in Branch & Bound) |
| int cmp(const void \*a, const void \*b) { |
| struct Item \*item1 = (struct Item \*)a; |
| struct Item \*item2 = (struct Item \*)b; |
| double ratio1 = (double)item1->value / item1->weight; |
| double ratio2 = (double)item2->value / item2->weight; |
| return (ratio1 > ratio2) ? -1 : 1; |
| } |
|  |
| // Function to calculate upper bound for Branch & Bound |
| int bound(struct Node u, int n, int W, struct Item items[]) { |
| if (u.weight >= W) return 0; // If weight exceeds capacity, no bound |
|  |
| int bound = u.value; |
| int totalWeight = u.weight; |
|  |
| for (int i = u.level; i < n; i++) { |
| if (totalWeight + items[i].weight <= W) { |
| totalWeight += items[i].weight; |
| bound += items[i].value; |
| } else { |
| bound += (W - totalWeight) \* items[i].value / items[i].weight; |
| break; |
| } |
| } |
| return bound; |
| } |
|  |
| // Branch & Bound function to solve the 0/1 Knapsack problem |
| int knapsackBranchAndBound(int W, int n, struct Item items[]) { |
| qsort(items, n, sizeof(struct Item), cmp); // Sort items by value/weight ratio |
|  |
| struct Node Q[n \* 2]; // Queue of nodes |
| int front = 0, rear = 0; // Queue front and rear pointers |
| int maxValue = 0; |
|  |
| struct Node u, v; |
| u.level = 0; |
| u.value = 0; |
| u.weight = 0; |
| u.bound = bound(u, n, W, items); |
|  |
| Q[rear++] = u; |
|  |
| while (front != rear) { |
| u = Q[front++]; // Dequeue |
|  |
| if (u.level == n) continue; |
|  |
| v.level = u.level + 1; |
|  |
| // Include the current item in the knapsack |
| v.weight = u.weight + items[u.level].weight; |
| v.value = u.value + items[u.level].value; |
| if (v.weight <= W && v.value > maxValue) |
| maxValue = v.value; |
| v.bound = bound(v, n, W, items); |
|  |
| if (v.bound > maxValue) Q[rear++] = v; // Enqueue if promising |
|  |
| // Exclude the current item from the knapsack |
| v.weight = u.weight; |
| v.value = u.value; |
| v.bound = bound(v, n, W, items); |
|  |
| if (v.bound > maxValue) Q[rear++] = v; // Enqueue if promising |
| } |
|  |
| return maxValue; |
| } |
|  |
| //Dynamic Programming |
|  |
| // Dynamic Programming function to solve the 0/1 Knapsack problem |
| int knapsackDynamic(int W, int n, struct Item items[]) { |
| int dp[n+1][W+1]; // DP table |
|  |
| // 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 (items[i-1].weight <= w) { |
| dp[i][w] = max(items[i-1].value + dp[i-1][w - items[i-1].weight], dp[i-1][w]); |
| } else { |
| dp[i][w] = dp[i-1][w]; |
| } |
| } |
| } |
|  |
| return dp[n][W]; |
| } |
|  |
| //Main Function |
|  |
| int main() { |
| struct Item items[] = {{2, 40}, {3, 50}, {5, 100}, {7, 130}, {4, 60}, {6, 110}}; |
| int n = sizeof(items) / sizeof(items[0]); // Number of items |
| int W = 10; // Capacity of the knapsack |
|  |
| // Backtracking Approach |
| int maxValueBacktracking = 0; |
| knapsackBacktracking(0, 0, 0, &maxValueBacktracking, W, n, items); |
| printf("Maximum value (Backtracking): %d\n", maxValueBacktracking); |
|  |
| // Branch & Bound Approach |
| int maxValueBranchBound = knapsackBranchAndBound(W, n, items); |
| printf("Maximum value (Branch & Bound): %d\n", maxValueBranchBound); |
|  |
| // Dynamic Programming Approach |
| int maxValueDynamic = knapsackDynamic(W, n, items); |
| printf("Maximum value (Dynamic Programming): %d\n", maxValueDynamic); |
|  |
| return 0; |
| } |

### OUTPUT:



### GRAPH:



## LAB-10: String Matching Algorithms 🔍

**Topic:**

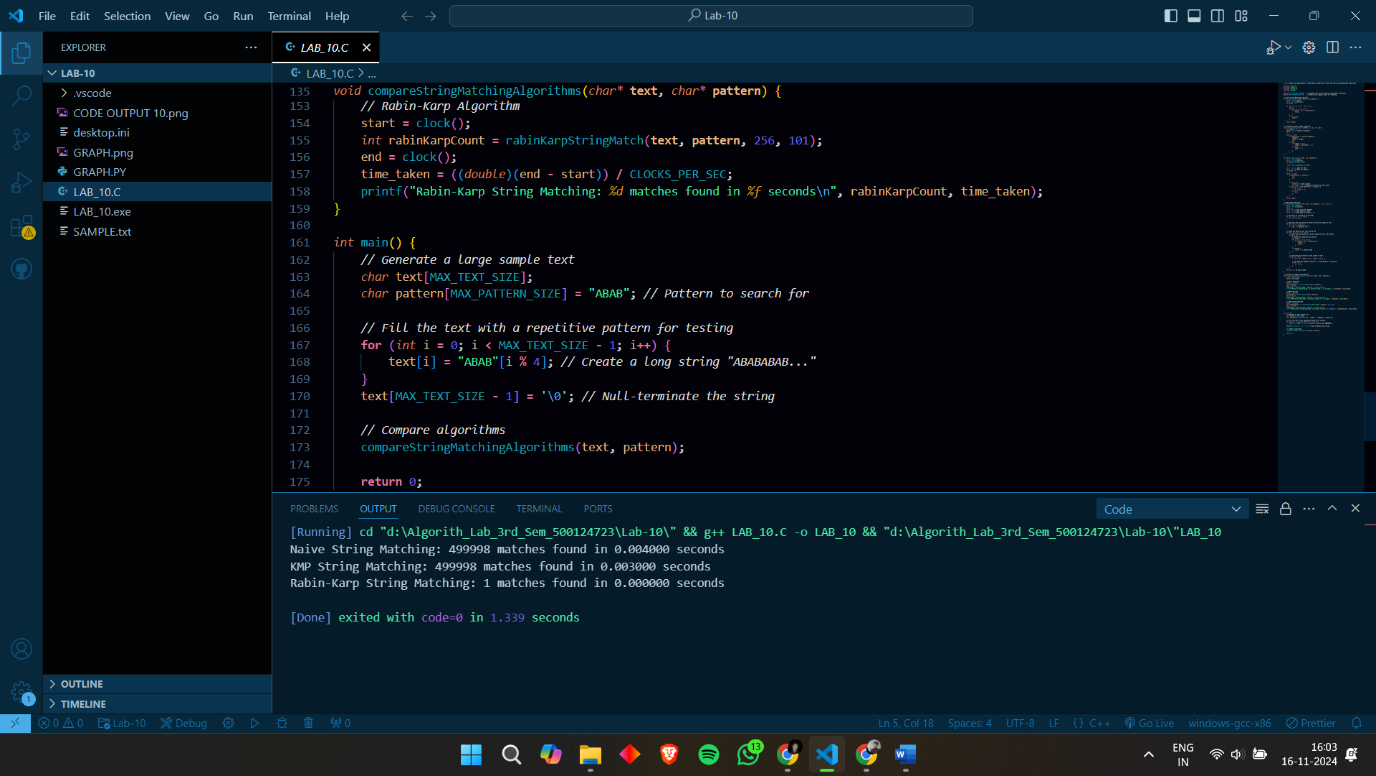
*Compare the performance of Rabin-Karp, Knuth-Morris-Pratt (KMP), and naive string-matching algorithms.*

* **Description**: In this lab, three string-matching algorithms are implemented: **Rabin-Karp**, **Knuth-Morris-Pratt (KMP)**, and the *naive approach*. The comparison of these algorithms provides insights into their **time complexity** and practical **use cases** for pattern matching.

### CODE:

|  |
| --- |
| #include <stdio.h> |
| #include <stdlib.h> |
| #include <string.h> |
| #include <time.h> |
|  |
| #define MAX\_TEXT\_SIZE 1000000  // Increased text size for better performance evaluation |
| #define MAX\_PATTERN\_SIZE 100    // Pattern size remains small for matching |
|  |
| // Naive String Matching Algorithm |
| int naiveStringMatch(char\* text, char\* pattern) { |
| int n = strlen(text); |
| int m = strlen(pattern); |
| int count = 0; |
|  |
| for (int i = 0; i <= n - m; i++) { |
| int j; |
| for (j = 0; j < m; j++) { |
| if (text[i + j] != pattern[j]) { |
| break; |
| } |
| } |
| if (j == m) { |
| count++; |
| } |
| } |
| return count; |
| } |
|  |
| // Knuth-Morris-Pratt (KMP) Algorithm |
| void computeLPSArray(char\* pattern, int m, int\* lps) { |
| int length = 0; |
| lps[0] = 0; // lps[0] is always 0 |
| int i = 1; |
|  |
| while (i < m) { |
| if (pattern[i] == pattern[length]) { |
| length++; |
| lps[i] = length; |
| i++; |
| } else { |
| if (length != 0) { |
| length = lps[length - 1]; |
| } else { |
| lps[i] = 0; |
| i++; |
| } |
| } |
| } |
| } |
|  |
| int KMPStringMatch(char\* text, char\* pattern) { |
| int n = strlen(text); |
| int m = strlen(pattern); |
| int lps[MAX\_PATTERN\_SIZE]; |
|  |
| computeLPSArray(pattern, m, lps); |
|  |
| int i = 0; // index for text |
| int j = 0; // index for pattern |
| int count = 0; |
|  |
| while (i < n) { |
| if (pattern[j] == text[i]) { |
| i++; |
| j++; |
| } |
|  |
| if (j == m) { |
| count++; // found a match |
| j = lps[j - 1]; // Continue to search for next match |
| } else if (i < n && pattern[j] != text[i]) { |
| if (j != 0) { |
| j = lps[j - 1]; |
| } else { |
| i++; |
| } |
| } |
| } |
| return count; |
| } |
|  |
| // Rabin-Karp Algorithm |
| int rabinKarpStringMatch(char\* text, char\* pattern, int d, int q) { |
| int n = strlen(text); |
| int m = strlen(pattern); |
| int i, j; |
| int p = 0; // hash value for pattern |
| int t = 0; // hash value for text |
| int h = 1; // The value of d^(m-1) % q |
|  |
| // The value of h would be "d^(m-1)%q" |
| for (i = 0; i < m - 1; i++) { |
| h = (h \* d) % q; |
| } |
|  |
| // Calculate the hash value of pattern and first window of text |
| for (i = 0; i < m; i++) { |
| p = (d \* p + pattern[i]) % q; |
| t = (d \* t + text[i]) % q; |
| } |
|  |
| // Slide the pattern over text one by one |
| for (i = 0; i <= n - m; i++) { |
| // Check the hash values of current window of text and pattern |
| if (p == t) { |
| // Check for characters one by one |
| int found = 1; |
| for (j = 0; j < m; j++) { |
| if (text[i + j] != pattern[j]) { |
| found = 0; |
| break; |
| } |
| } |
| if (found) { |
| return 1; // Match found |
| } |
| } |
|  |
| // Calculate hash value for next window of text |
| if (i < n - m) { |
| t = (d \* (t - text[i] \* h) + text[i + m]) % q; |
|  |
| // We might get negative value of t, converting it to positive |
| if (t < 0) { |
| t = t + q; |
| } |
| } |
| } |
| return 0; // No match found |
| } |
|  |
| // Function to compare performances |
| void compareStringMatchingAlgorithms(char\* text, char\* pattern) { |
| clock\_t start, end; |
| double time\_taken; |
|  |
| // Naive Algorithm |
| start = clock(); |
| int naiveCount = naiveStringMatch(text, pattern); |
| end = clock(); |
| time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC; |
| printf("Naive String Matching: %d matches found in %f seconds\n", naiveCount, time\_taken); |
|  |
| // KMP Algorithm |
| start = clock(); |
| int kmpCount = KMPStringMatch(text, pattern); |
| end = clock(); |
| time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC; |
| printf("KMP String Matching: %d matches found in %f seconds\n", kmpCount, time\_taken); |
|  |
| // Rabin-Karp Algorithm |
| start = clock(); |
| int rabinKarpCount = rabinKarpStringMatch(text, pattern, 256, 101); |
| end = clock(); |
| time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC; |
| printf("Rabin-Karp String Matching: %d matches found in %f seconds\n", rabinKarpCount, time\_taken); |
| } |
|  |
| int main() { |
| // Generate a large sample text |
| char text[MAX\_TEXT\_SIZE]; |
| char pattern[MAX\_PATTERN\_SIZE] = "ABAB"; // Pattern to search for |
|  |
| // Fill the text with a repetitive pattern for testing |
| for (int i = 0; i < MAX\_TEXT\_SIZE - 1; i++) { |
| text[i] = "ABAB"[i % 4]; // Create a long string "ABABABAB..." |
| } |
| text[MAX\_TEXT\_SIZE - 1] = '\0'; // Null-terminate the string |
|  |
| // Compare algorithms |
| compareStringMatchingAlgorithms(text, pattern); |
|  |
| return 0; |
| } |

### OUTPUT:



### GRAPH:

