**Data Structures Applications Lab (21EECF201) [0-0-2]**

**Term-work Report**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | 01 | | | | |  |  | | | | |
| **Student Name** | ARYAN KAMAT | | | | |  |  | | | | |
| **SRN** | 01FE21BEC190 | | | | **Roll Number** | | 416 | | **Division** | D | |
| **Code of ethics:**  I hereby declare that I am bound by ethics and have not copied any text/program/figure without acknowledging the content creators. I abide to the rule that upon plagiarized content all my marks will be made to zero.  Digital signature of the student | | | | | | | | | | | |
| **Apply Programming Skills**  **(5 marks)** | | **Identify Constraints and Implement**  **(10 marks)** | | **Integrate Modules**  **(3 Marks)** | | **Debugging and Tool usage**  **(2 marks)** | | **Remarks** | | | **Total**  **(20 Marks)** |
|  | |  | |  | |  | |  | | |  |
| **Problem Statement** | | | | | | | | | | | |
| Explain the operation of each algorithm type, take into account two examples of programmes for each algorithm type, and express the time complexity of each programme.   1. Iterative, 2. Recursive, 3. Back tracking, 4. Divide and conquer, 5. Dynamic programming, 2. Greedy, 7. Branch and Bound, 8. Brute force, 9. Randomized | | | | | | | | | | | |
| **Type of algorithm** | **Example No** | | **Which data structures are used?** | | | | | **What is the time complexity? O(n)** | | | |
| Iterative | **1** | | ARRAY | | | | | O(n^2) | | | |
| **2** | | ARRAY | | | | | O(mn) | | | |
| Recursive | **1** | | ARRAY | | | | | O(n) | | | |
| **2** | | ARRAY | | | | | O(mn) | | | |
| Back tracking | **1** | | ARRAY | | | | | O(n^3) | | | |
| **2** | | Linked list | | | | | O(9!) | | | |
| Divide and conquer | **1** | | ARRAY | | | | | O(log n) | | | |
| **2** | | ARRAY | | | | |  | | | |
| Dynamic programming | **1** | | ARRAY | | | | | O(n^3) | | | |
| **2** | | ARRAY | | | | | O(n) | | | |
| Greedy | **1** | | ARRAY | | | | | O(n) | | | |
| **2** | | Linked list | | | | | O(n^2) | | | |
| Branch and bound | **1** | | ARRAY | | | | | O(n\*k) | | | |
| **2** | | ARRAY | | | | | O(n) | | | |
| Brute force | **1** | | ARRAY | | | | | O(n) | | | |
| **2** | | ARRAY | | | | | O(n^2) | | | |
| Randomized | **1** | | ARRAY | | | | | O(n) | | | |
| **2** | | Linked list | | | | | O(n) | | | |

Were you able to solve this problem? If not what where the challenges?

Yes I was able to solve the problems

What assistance do you need to learn this term work better?

Had taken help from the lecutrers

What are the areas you think you should work on to be able to make this solution better?

Linked list, structures

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Iterative | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| Iterative algorithms are a type of algorithm that use looping constructs to repeat a set of instructions until a specific condition is met. Unlike recursive algorithms, which use function calls to repeatedly execute a set of instructions, iterative algorithms use loops, such as for loops, while loops, and do-while loops, to repeatedly execute the same set of instructions until the desired result is achieved.  The name "iterative algorithm" comes from the fact that these algorithms use iteration, which means repetition of a set of instructions, to perform a task. In other words, they repeatedly execute a set of instructions until a certain condition is met.  Iterative algorithms are used in search operations such as bubble sort, quick sort, merge sort and search operations. They are useful in conditions where a problem can be broken down and solved iteratively.  **The basic steps in an iterative algorithm are:**   * Initialize any necessary variables or data structures before entering the loop. * Set up a loop construct, such as a for loop, while loop, or do-while loop, that will repeat the desired set of instructions until a specific condition is met. * Within the loop, execute the set of instructions that need to be repeated. * Check the stopping condition at the end of each iteration of the loop. If the condition is met, exit the loop. Otherwise, continue to the next iteration. * After the loop has exited, return the result or perform any necessary debugging steps. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| **Program of Bubble sort using iterative algorithm:**  #include <stdio.h>  void bub\_srt(int arr[], int n) {  int i, j;  for (i = 0; i < n - 1; i++) {  for (j = 0; j < n - i - 1; j++) {  if (arr[j] > arr[j+1]) {  int temp = arr[j];  arr[j] = arr[j+1];  arr[j+1] = temp;  }  }  }  }  int main() {  int n, i;  printf("Enter the size of the array: ");  scanf("%d", &n);  int arr[n];  printf("Enter the elements of the array:\n");  for (i = 0; i < n; i++) {  scanf("%d", &arr[i]);  }  printf("Original array:\n");  for (i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  bub\_sort(arr, n);  printf("\nSorted array:\n");  for (i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Enter the size of the array: 5  Enter the elements of the array:  4 2 7 1 3 | | | | | | | |
| **Sample Output:** | | | | | | | |
| Original array:  4 2 7 1 3  Sorted array:  1 2 3 4 7 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of the bubble sort algorithm is O(n^2), where n is the number of elements in the array. | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| **Program to find transpose of a 2D-Matrix using iterative algorithm**  #include <stdio.h>  void transpose(int mat[][100], int m, int n) {  int trans[n][m];  for (int i = 0; i < n; i++) {  for (int j = 0; j < m; j++) {  trans[i][j] = mat[j][i];  }  }  printf("Transpose of the matrix:\n");  for (int i = 0; i < n; i++) {  for (int j = 0; j < m; j++) {  printf("%d ", trans[i][j]);  }  printf("\n");  }  }  int main() {  int mat[100][100], m, n;  printf("Enter number of rows and columns: ");  scanf("%d %d", &m, &n);  printf("Enter the matrix elements:\n");  for (int i = 0; i < m; i++) {  for (int j = 0; j < n; j++) {  scanf("%d", &mat[i][j]);  }  }  printf("Original matrix:\n");  for (int i = 0; i < m; i++) {  for (int j = 0; j < n; j++) {  printf("%d ", mat[i][j]);  }  printf("\n");  }  transpose(mat, m, n);  return 0;  } |
| **Sample Input:** |
| Enter number of rows and columns: 3 3  Enter the matrix elements:  1 2 3  4 5 6  7 8 9 |
| **Sample Output:** |
| Original matrix:  1 2 3  4 5 6  7 8 9  Transpose of the matrix:  1 4 7  2 5 8  3 6 9 |
| **Time complexity calculation:** |
| The time complexity of this code is O(mn), where m is the number of rows and n is the number of columns of the matrix. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Recursive | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| Recursion is a programming technique that involves a function calling itself in order to solve a problem. In other words, a recursive function is one that calls itself with different arguments until it reaches a base case, which is a condition that allows the function to stop calling itself and return a result.  The term "recursive" is used to describe this type of algorithm because it involves repeatedly calling the same function or procedure, which creates a recursive call stack  Applications of Recursive algorithms are Data structures, Searching and sorting operations like binary search, merge sort, quick sort etc.  The steps involved in a recursive algorithm are:   * Identify the base case: This is the condition that allows the function to stop calling itself and return a result. * Define the recursive case: This is the condition that determines when the function needs to call itself with a different set of arguments. * Call the function recursively: When the recursive case is met, the function calls itself with a new set of arguments. * Combine the results: Once the base case is reached, the function combines the results obtained from the recursive calls and returns the final result. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| **Program to determine whether a given string is Palindrome or not:**  #include <stdio.h>  #include <string.h>  int ispal(char str[], int start, int end)  {  If (start >= end)  {  return 1;  }  if (str[start] == str[end])  {  return ispal(str, start + 1, end - 1);  }  else  {  return 0;  }  int main() {  char str[100];  printf("Enter a string: ");  scanf("%s", str);  int len = strlen(str);  int result = ispal(str, 0, len - 1);  if (result == 1) {  printf("%s is a palindrome\n", str);  }  else {  printf("%s is not a palindrome\n", str);  }  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Enter a string: hello  Enter a string: madam | | | | | | | |
| **Sample Output:** | | | | | | | |
| hello is not a palindrome  madam is a palindrome | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| Time complexity of the program is O(n), where n is the length of the input string. | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| **Program to find Maximum element of an array:**  #include <stdio.h>  int findMax(int arr[], int n) {  // base case: if array contains only one element, return it  if (n == 1) {  return arr[0];  }  // recursive case: compare first element with maximum element of the rest of the array  int max = findMax(arr, n - 1);  if (arr[n-1] > max) {  return arr[n-1];  } else {  return max;  }  }  int main() {  int n, i;  printf("Enter the size of the array: ");  scanf("%d", &n);  int arr[n];  printf("Enter the elements of the array:\n");  for (i = 0; i < n; i++) {  scanf("%d", &arr[i]);  }  int max = findMax(arr, n);  printf("Maximum element is: %d", max);  return 0;  } |
| **Sample Input:** |
| Enter the size of the array: 5  Enter the elements of the array  3 4 6 9 5 |
| **Sample Output:** |
| Maximum element is: 9 |
| **Time complexity calculation:** |
| time complexity of this program is also O(n). |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Backtracking | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| Backtracking is a technique used to solve computational problems that involve trying out different solutions and undoing a certain sequence of moves if it is found to be incorrect. The algorithmic approach involves starting from an initial solution and trying to extend it step by step until a feasible solution is found. If a step leads to an unfeasible solution, then the algorithm backtracks to the previous step and tries a different solution from there.  The name "backtracking" refers to a specific behaviour of the algorithm. Backtracking algorithms solve problems by incrementally building a candidate solution and checking if it satisfies the problem constraints. If a solution candidate violates any constraint, the algorithm abandons or "backs up" and tries a different candidate solution.  Backtracking algorithms are used to solve various computational problems such as constraint satisfaction problems, Maze Solving Problems, and games such as Sudoku and chess.  **The steps in a backtracking algorithm are:**   * Define the problem in terms of the initial state, the goal state, and the set of constraints that must be satisfied. * Choose a possible solution to the problem and apply it to the current state. * Check if the new state satisfies the constraints. If it does, move to the next state and repeat step 2. * If the new state violates any of the constraints, undo the last move and choose a different solution from the remaining possibilities. * If all possibilities have been exhausted and no feasible solution has been found, backtrack to the previous state and choose a different solution from there. * Repeat steps 2-5 until a feasible solution is found or all possibilities have been exhausted. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  #include <stdlib.h>  int isSafe(int board[][N], int row, int col) {  int i, j;  // Check this row on left side  for (i = 0; i < col; i++) {  if (board[row][i])  return 0;  }  // Check upper diagonal on left side  for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {  if (board[i][j])  return 0;  }  // Check lower diagonal on left side  for (i = row, j = col; j >= 0 && i < N; i++, j--)  {  if (board[i][j])  return 0;  }  // If no conflict found, return 1  return 1;  }  int solveNQueens(int board[][N], int col)  {  // Base case: all queens are placed  if (col == N)  return 1;  int i;  // Try placing queen in each row of current column  for (i = 0; i < N; i++)  {  if (isSafe(board, i, col)) {  // Place the queen in board[i][col]  board[i][col] = 1;  // Recur to place rest of the queens  if (solveNQueens(board, col + 1))  return 1;  // If placing queen in board[i][col] doesn't lead to solution, remove it  board[i][col] = 0;  }  }  // If queen can't be placed in any row of current column, return 0  return 0;  }  void printSolution(int board[][N]) {  int i, j;  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  printf("%d ", board[i][j]);  }  printf("\n");  }  }  int main() {  int N;  printf("Enter the size of the chess board: ");  scanf("%d", &N);  int board[N][N];  printf("Enter the initial state of the chess board (0 for empty cell, 1 for queen):\n");  int i, j;  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  scanf("%d", &board[i][j]);  }  }  if (solveNQueens(board, 0) == 0) {  printf("Solution does not exist\n");  } else {  printf("Solution:\n");  printSolution(board);  }  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Enter the value of N: 4 | | | | | | | |
| **Sample Output:** | | | | | | | |
| Solution for N = 4:  - Q - -  - - - Q  Q - - -  - - Q -  - - Q -  Q - - -  - - - Q  - Q - - | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of the given code is O(N^3). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *// This program implements a backtracking algorithm for solving Sudoku puzzles.*  *#include <stdio.h>*  *#include <stdbool.h>*  *#define N 3*  *bool solveSudoku(int grid[N][N]) // A function to solve a Sudoku puzzle.*  *{*  *// Find an empty cell in the grid.*  *int row, col;*  *bool isBlankFound = false;*  *for (row = 0; row < N; row++) {*  *for (col = 0; col < N; col++) {*  *if (grid[row][col] == 0) {*  *isBlankFound = true;*  *break;*  *}*  *}*  *if (isBlankFound) {*  *break;*  *}*  *}*  *// If no empty cell is found, the Sudoku is solved.*  *if (!isBlankFound) {*  *return true;*  *}*  *// Try numbers from 1 to 9 in the current cell.*  *for (int num = 1; num <= 9; num++) {*  *// Check if the number is safe to place in the current cell.*  *bool isSafe = true;*  *// Check the row and column.*  *for (int i = 0; i < N; i++) {*  *if (grid[row][i] == num || grid[i][col] == num) {*  *isSafe = false;*  *break;*  *}*  *}*  *// Check the 3x3 grid.*  *int startRow = row - row % 3;*  *int startCol = col - col % 3;*  *for (int i = 0; i < 3; i++) {*  *for (int j = 0; j < 3; j++) {*  *if (grid[startRow + i][startCol + j] == num) {*  *isSafe = false;*  *break;*  *}*  *}*  *if (!isSafe) {*  *break;*  *}*  *}*  *// If the number is safe, place it in the current cell and recursively solve the Sudoku.*  *if (isSafe) {*  *grid[row][col] = num;*  *if (solveSudoku(grid)) {*  *return true;*  *}*  *// If the current placement doesn't lead to a solution, backtrack.*  *grid[row][col] = 0;*  *}*  *}*  *// If no number can be placed in the current cell, backtrack.*  *return false;*  *}*  *// A function to print the Sudoku grid.*  *void printGrid(int grid[N][N]) {*  *for (int row = 0; row < N; row++) {*  *for (int col = 0; col < N; col++) {*  *printf("%2d ", grid[row][col]);*  *}*  *printf("\n");*  *}*  *}*  *// The main function.*  *int main() {*  *int grid[N][N];*  *printf("Enter the Sudoku grid (3\*3):\n");*  *for (int row = 0; row < N; row++) {*  *for (int col = 0; col < N; col++) {*  *scanf("%d", &grid[row][col]); // Read each cell of the grid from the user*  *}*  *}*  *// Solve the Sudoku puzzle.*  *if (solveSudoku(grid)) {*  *printf("Sudoku Solution:\n");*  *printGrid(grid);*  *} else {*  *printf("No solution exists for the given Sudoku.\n");*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| Enter the Sudoku grid (3\*3):  2  5  2  5  2  5  2  3  5 |
| **Sample Output:** |
| **Sudoku Solution:**  **2 5 2**  **5 2 5**  **2 3 5** |
| **Time complexity calculation:** |

The time complexity of the code is O(9!), which is a very large number. This is because the algorithm must try all possible combinations of numbers in the empty cells of the Sudoku grid. For a 3x3 Sudoku puzzle, there are 9! possible combinations of numbers. For a 9x9 Sudoku puzzle, there are 9! \* 9! possible combinations of numbers.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Divide and conquer | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| Divide and conquer is a problem-solving strategy in which a problem is divided into smaller subproblems that are solved recursively. The solutions to these subproblems are then combined to obtain the final solution to the original problem. This approach is often used to solve complex problems that would be difficult or impossible to solve directly.  Divide and conquer algorithm is used in sorting operations like merge sort and quick sort. It is also used in binary search. Strassen's algorithm is a matrix multiplication algorithm that uses the divide and conquer approach  Here are the steps involved:   * Divide the problem into smaller subproblems: The first step is to break the original problem into smaller subproblems that are similar to the original problem but are simpler to solve. * Solve the subproblems: The next step is to solve the subproblems recursively. This can be done by applying the same divide and conquer algorithm to each subproblem until the subproblems are small enough to be solved directly. * Combine the solutions: The final step is to combine the solutions to the subproblems to obtain the solution to the original problem. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  int binary\_search(int arr[], int element, int low, int high) {  if (high >= low) {  int mid = low + (high - low) / 2;  if (arr[mid] == element) {  return mid;  }  if (arr[mid] > element) {  return binary\_search(arr, element, low, mid - 1);  }  return binary\_search(arr, element, mid + 1, high);  }  return -1;  }  int main() {  int siz, element, i;  printf("Enter the size of the array: ");  scanf("%d", &siz);  int arr[siz];  printf("Enter the elements of the array:\n");  for (i = 0; i < siz; i++) {  scanf("%d", &arr[i]);  }  printf("Enter the element to be searched: ");  scanf("%d", &element);  int result = binary\_search(arr, element, 0, siz - 1);  if (result == -1) {  printf("Element not found in the array.");  } else {  printf("Element found at index %d.", result);  }  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Enter the size of the array: 5  Enter the elements of the array:  2  4  6  8  10  Enter the element to be searched: 6 | | | | | | | |
| **Sample Output:** | | | | | | | |
| Element found at index 2. | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| **The time complexity of binary search algorithm is O(log n),** | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  void swap(int arr[], int i, int j)  {  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  int split(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, j);  }  }  swap(arr, i + 1, high);  return i + 1;  }  void Quick\_Sort(int arr[], int low, int high)  {  if (low < high)  {  int pi = split(arr, low, high);  Quick\_Sort(arr, low, pi - 1);  Quick\_Sort(arr, pi + 1, high);  }  }  int main()  {  int n, i;  printf("Enter the size of array:");  scanf("%d", &n);  int arr[n];  printf("Enter the elements of the array:\n");  for (i = 0; i < n; i++)  {  scanf("%d", &arr[i]);  }  printf("Original array:\n");  for (i = 0; i < n; i++)  {  printf("%d ", arr[i]);  }  Quick\_Sort(arr, 0, n - 1);  printf("\nSorted array:\n");  for (i = 0; i < n; i++)  {  printf("%d ", arr[i]);  }  return 0;  } |
| **Sample Input:** |
| Enter the size of the array: 5  Enter the elements of the array:  9 1 5 2 8 |
| **Sample Output:** |
| Original array:  9 1 5 2 8  Sorted array:  1 2 5 8 9 |
| **Time complexity calculation:** |
| The time complexity of the Quick Sort algorithm is On log n) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Dynamic Programming | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| Dynamic programming is a problem-solving technique that involves breaking down a problem into smaller subproblems and solving each subproblem only once. The technique is used to solve optimization problems, where we seek to minimize or maximize a given objective function. Dynamic programming can be used to solve problems that have an optimal substructure and overlapping subproblems.  **Applications of dynamic programming include**:   * Finding the shortest path between two nodes in a graph * Solving the knapsack problem * Computing the nth Fibonacci number * Finding the longest common subsequence between two strings * Solving the matrix chain multiplication problem * Computing the edit distance between two strings   **The steps involved in dynamic programming algorithm are:**  Identify the problem and define a recursive solution: Identify the problem that needs to be solved and define a recursive solution. Break the problem down into smaller subproblems that have a similar structure.  Define the base case: Define the base case of the recursive solution. This is the smallest subproblem that can be solved directly.  Store the results: Create a data structure to store the results of subproblems that have already been solved. This can be done using a table or an array.  Define the recursive formula: Define the recursive formula that will be used to solve the problem. This formula should use the results of subproblems that have already been solved.  Solve the problem: Use the recursive formula to solve the problem. Start with the base case and work up to the solution of the original problem.  Optimize the solution: Analyse the algorithm and optimize it if necessary. This can be done by identifying duplicate calculations and avoiding them by using memorization. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| **Program for Dynamic programming using the flyod-warshall’s algorithm which calculates shortest distance between vertices**  #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  void floydWarshall(int \*\*graph, int n) {  int \*\*dist = (int \*\*) malloc(n \* sizeof(int \*));  for (int i = 0; i < n; i++) {  dist[i] = (int \*) malloc(n \* sizeof(int));  for (int j = 0; j < n; j++) {  dist[i][j] = graph[i][j];  }  }  for (int k = 0; k < n; k++) {  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if (dist[i][k] != INT\_MAX && dist[k][j] != INT\_MAX && dist[i][k] + dist[k][j] < dist[i][j]) {  dist[i][j] = dist[i][k] + dist[k][j];  }  }  }  }  printf("Shortest distances between all pairs of vertices:\n");  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if (dist[i][j] == INT\_MAX) {  printf("INF ");  } else {  printf("%d ", dist[i][j]);  }  }  printf("\n");  }  for (int i = 0; i < n; i++) {  free(dist[i]);  }  free(dist);  }  int main() {  int n;  printf("Enter the number of vertices in the graph: ");  scanf("%d", &n);  int \*\*graph = (int \*\*) malloc(n \* sizeof(int \*));  printf("Enter the adjacency matrix of the graph:\n");  for (int i = 0; i < n; i++) {  graph[i] = (int \*) malloc(n \* sizeof(int));  for (int j = 0; j < n; j++) {  scanf("%d", &graph[i][j]);  }  }  floydWarshall(graph, n);  for (int i = 0; i < n; i++) {  free(graph[i]);  }  free(graph);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Enter the number of vertices: 5  Enter the adjacency matrix:  0 3 8 INF -4  INF 0 INF 1 7  INF 4 0 INF INF  2 INF -5 0 INF  INF INF INF 6 0 | | | | | | | |
| **Sample Output:** | | | | | | | |
| Shortest distances between all pairs of vertices:  0 5 8 9  INF 0 3 4  INF INF 0 1  INF INF INF 0 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of theprogram is O(n^3). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  // Function to calculate the nth Fibonacci number using dynamic programming  int fib(int n) {  // Declare an array to store previously calculated Fibonacci numbers  int ar[n + 1];  // Set the first two elements of the array to 0 and 1 respectively  ar[0] = 0;  ar[1] = 1;  // Loop through the array and calculate each Fibonacci number using the previous two numbers  for (int i = 2; i <= n; i++) {  ar[i] = ar[i - 1] + ar[i - 2];  }  // Return the nth Fibonacci number  return ar[n];  }  // Main function  int main() {  int n;  // Prompt the user to enter the position of the Fibonacci number they want to calculate  printf("Enter n: ");  scanf("%d", &n);  // Call the fib function to calculate the nth Fibonacci number and print the result  printf("Fibonacci number at position %d is: %d", n, fib(n));  return 0;  } |
| **Sample Input:** |
| Enter n: 7 |
| **Sample Output:** |
| Fibonacci number at position 7 is: 13 |
| **Time complexity calculation:** |

time complexity of this program is O(n).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Greedy | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. It works by making locally optimal choices at each step with the hope of finding a global optimum.  The general steps involved in a greedy algorithm are:   * Define the problem and the goal of the algorithm. * Determine the optimal solution for a small instance of the problem. * Choose the best possible option at the current step. * Recursively apply the greedy algorithm to the remaining sub-problems until the goal is achieved.   Applications of the greedy algorithm include:  Knapsack problem: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is maximized.  Huffman coding: A lossless data compression algorithm that uses variable-length codes for encoding a source symbol where the code assigned to each symbol is based on the probability of occurrence of that symbol.  Prim's algorithm: A minimum spanning tree algorithm that finds a subset of the edges that form a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| **Program to determine the minimum number of coins required to make up a given amount of money using the greedy algorithm**  #include <stdio.h>  #include <stdlib.h>  void Coin\_Change(int coins[], int n, int amount)  {  int Countcoin = 0;  printf("Coins used: ");  for (int i = n - 1; i >= 0; i--) // iterate through the coin denominations in descending order  {    while (amount >= coins[i])  {  amount -= coins[i];  Countcoin++; // add the maximum number of coins of that denomination to the solution  printf("%d ", coins[i]);  }  }  printf("\nMinimum number of coins required: %d\n", Countcoin);  }  int main()  {  int n, amount;  printf("Enter the number of coins: ");  scanf("%d", &n);  int coins[n];  printf("Enter the coin denominations in ascending order: ");  for (int i = 0; i < n; i++)  {  scanf("%d", &coins[i]);  }  printf("Enter the total amount to make up: ");  scanf("%d", &amount);  Coin\_Change(coins, n, amount);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Enter the number of coins: 4  Enter the coin denominations in ascending order: 1 3 5 10  Enter the total amount to make up: 16 | | | | | | | |
| **Sample Output:** | | | | | | | |
| Coins used: 10 5 1  Minimum number of coins required: 3 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of the code is O(n) | | | | | | | |

|  |
| --- |
| ***Code for example 2:*** |
| #include <stdio.h>  #include <stdlib.h>  struct node {  int value;  struct node \*next;  };  struct node \*head = NULL;  void add\_node(int value) {  struct node \*new\_node = malloc(sizeof(struct node));  new\_node->value = value;  new\_node->next = head;  head = new\_node;  }  void print\_list() {  struct node \*current = head;  while (current != NULL) {  printf("%d ", current->value);  current = current->next;  }  printf("\n");  }  int main() {  int n, value;  printf("Enter the number of values: ");  scanf("%d", &n);  printf("Enter the values:\n");  for (int i = 0; i < n; i++) {  scanf("%d", &value);  add\_node(value);  }  printf("Original list: ");  print\_list();  // Greedy algorithm  struct node \*current = head;  struct node \*prev = NULL;  while (current != NULL) {  // Find the largest denomination that is smaller than the current value  struct node \*largest\_smaller = NULL;  for (struct node \*node = head; node != NULL; node = node->next) {  if (node->value < current->value && (largest\_smaller == NULL || node->value > largest\_smaller->value)) {  largest\_smaller = node;  }  }  // Remove the largest denomination from the list  if (largest\_smaller != NULL) {  if (prev == NULL) {  head = largest\_smaller->next;  } else {  prev->next = largest\_smaller->next;  }  free(largest\_smaller);  }  // Move to the next node  prev = current;  current = current->next;  }  printf("Modified list: ");  print\_list();  return 0;  } |
| **Sample Input:** |
| Enter the number of values: 5  Enter the values:  1  2  3  4  5 |
| **Sample Output:** |
| Original list: 5 4 3 2 1 |
| **Time complexity calculation:** |
| The time complexity of the code is O(n^2), where n is the number of nodes in the linked list. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Branch and bound | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| Branch and bound is an algorithmic technique used in optimization problems to efficiently search through a large solution space and find the best solution. It is used to solve combinatorial optimization problems, which are problems where the goal is to find the best combination of decisions among a set of alternatives.  The name "Branch and bound" refers to the two main steps involved in the algorithm. The "branch" step involves dividing the problem into smaller subproblems, and the "bound" step involves computing lower and upper bounds on the solutions of those subproblems   * The steps involved in the Branch and bound algorithm are: * Initialization: Set up the problem and initialize the algorithm. * Branching: Divide the problem into smaller subproblems. * Bound computation: Compute lower and upper bounds on the solutions of the subproblems. * Pruning: Eliminate subproblems that cannot lead to a better solution than the current best solution. * Update: Update the current best solution based on the solutions of the remaining subproblems. * Termination: Stop the algorithm when all subproblems have been explored.   The applications of Branch and bound algorithm include various optimization problems such as integer linear programming, traveling salesman problem, graph coloring problem, knapsack problem, and many others. It is also used in game theory to solve games with perfect information, such as chess or checkers. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *#include <stdio.h>*  *#include <stdlib.h>*  *struct Item {*  *int weight;*  *int value;*  *};*  *int max(int a, int b) {*  *return (a > b) ? a : b;*  *}*  *int knapsack(int capacity, struct Item items[], int n) {*  *// Create a dynamic programming table*  *int\*\* dp = (int\*\*)malloc((n + 1) \* sizeof(int\*));*  *for (int i = 0; i <= n; i++) {*  *dp[i] = (int\*)malloc((capacity + 1) \* sizeof(int));*  *}*  *// Fill the table with computed values*  *for (int i = 0; i <= n; i++) {*  *for (int w = 0; w <= capacity; w++) {*  *if (i == 0 || w == 0) {*  *// Base case: no items or no capacity left*  *dp[i][w] = 0;*  *} else if (items[i - 1].weight <= w) {*  *// If the current item can be included, compute the maximum value*  *dp[i][w] = max(items[i - 1].value + dp[i - 1][w - items[i - 1].weight], dp[i - 1][w]);*  *} else {*  *// If the current item cannot be included, take the value from the previous row*  *dp[i][w] = dp[i - 1][w];*  *}*  *}*  *}*  *int result = dp[n][capacity];*  *// Free the memory allocated for the dynamic programming table*  *for (int i = 0; i <= n; i++) {*  *free(dp[i]);*  *}*  *free(dp);*  *return result;*  *}*  *int main() {*  *int n, capacity;*  *printf("Enter the number of items: ");*  *scanf("%d", &n);*  *// Create an array to store the items*  *struct Item\* items = (struct Item\*)malloc(n \* sizeof(struct Item));*  *printf("Enter the weight and value of each item:\n");*  *for (int i = 0; i < n; i++) {*  *printf("Item %d: ", i + 1);* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter the number of items: 3*  *Enter the weight and value of each item:*  *Item 1: 1*  *2*  *Item 2: 3*  *4*  *Item 3: 5*  *6*  *Enter the knapsack capacity: 10* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Maximum value in the knapsack: 12* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of this code is **O(n \* k)**, where 'n' is the number of items and 'k' is the knapsack capacity. | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| //Here's an example program in C that implements the branch and bound algorithm for solving the Traveling Salesman Problem (TSP):  #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #define N 10 // Maximum number of cities  int graph[N][N]; // Adjacency matrix representing the distance between cities  int visited[N]; // Array to track visited cities  int minCost = INT\_MAX; // Variable to store the minimum cost  void tsp(int currentCity, int n, int cost, int count) {  if (count == n && graph[currentCity][0] != 0) {  // If all cities have been visited and there is a path back to the starting city  cost += graph[currentCity][0]; // Add the cost of returning to the starting city  if (cost < minCost) {  minCost = cost; // Update the minimum cost  }  return;  }  // Traverse all unvisited cities  for (int i = 0; i < n; i++) {  if (visited[i] == 0 && graph[currentCity][i] != 0) {  visited[i] = 1; // Mark the city as visited  tsp(i, n, cost + graph[currentCity][i], count + 1); // Recursively visit the next city  visited[i] = 0; // Mark the city as unvisited for backtracking  }  }  }  int main() {  int n; // Number of cities  printf("Enter the number of cities: ");  scanf("%d", &n);  printf("Enter the adjacency matrix representing the distances between cities:\n");  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  scanf("%d", &graph[i][j]);  }  }  visited[0] = 1; // Mark the starting city as visited  tsp(0, n, 0, 1); // Start the branch and bound algorithm from the first city  printf("Minimum cost for the Traveling Salesman Problem: %d\n", minCost);  return 0;  } |
| Enter the number of cities: 2  Enter the adjacency matrix representing the distances between cities:  1  2  3  4 |
| **Sample Output:** |
| Minimum cost for the Traveling Salesman Problem: 5 |
| **Time complexity calculation:** |
| The time complexity of the program is O(n!) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Brute force | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| Brute force algorithm is a straightforward approach to solving problems by trying every possible solution sequentially and selecting the best one. It is also known as exhaustive search algorithm or generate-and-test algorithm.  **The general steps in brute force algorithm are:**   * Generate all possible solutions. * Test each solution by evaluating its suitability or fitness. * Select the best solution based on the evaluation. * Output the selected solution.   **Applications of brute force algorithm:**   * Selection Sort * Bubble Sort * Sequential Search * Brute Force String Match * Travelling Sales Problem (Can be solved using other techniques too) * Knapsack Problem (Can be solved using other techniques too) | | | | | | | |
| **Code for example 1:** | | | | | | | |
| **Program of sequential search of an element in an array to find its position using the brute force algorithm:**  #include <stdio.h>  int sequential\_search(int arr[], int n, int x)  {  for (int i = 0; i < n; i++)  {  if (arr[i] == x)  {  return i; // return the index of the element if found  }  }  return -1; // return -1 if the element is not found  }  int main()  {  int n, x;  printf("Enter the size of the array: ");  scanf("%d", &n);  int arr[n];  printf("Enter the elements of the array:\n");  for (int i = 0; i < n; i++)  {  scanf("%d", &arr[i]);  }  printf("Enter the element to be searched: ");  scanf("%d", &x);  int result = sequential\_search(arr, n, x);  if (result == -1)  {  printf("Element not found\n");  }  else  {  printf("Element found at index %d\n", result);  }  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Enter the size of the array: 5  Enter the elements of the array:  1 2 3 4 5  Enter the element to be searched: 3 | | | | | | | |
| **Sample Output:** | | | | | | | |
| Element found at index 2 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of the code is O(n) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| **Program of selection sort using brute force algorithm:**  #include <stdio.h>  void selectionSort(int arr[], int n) {  int i, j, minIndex, temp;  // iterate through the array  for (i = 0; i < n - 1; i++)  {  // find the index of the minimum element in the unsorted part of the array  minIndex = i;  for (j = i + 1; j < n; j++)  {  if (arr[j] < arr[minIndex])  {  minIndex = j;  }  }  // swap the minimum element with the first element of the unsorted part of the array  temp = arr[minIndex];  arr[minIndex] = arr[i];  arr[i] = temp;  }  }  int main() //  {  int n, i;  printf("Enter the size of the array: ");  scanf("%d", &n);  int arr[n];  printf("Enter the elements of the array:\n");  for (i = 0; i < n; i++)  {  scanf("%d", &arr[i]);  }  selectionSort(arr, n);  printf("Sorted array:\n");  for (i = 0; i < n; i++)  {  printf("%d ", arr[i]);  }  return 0;  } |
| **Sample Input:** |
| Enter the size of the array: 5  Enter the elements of the array:  4 8 3 2 1 |
| **Sample Output:** |
| Sorted array:  1 2 3 4 8 |
| **Time complexity calculation:** |
| The time complexity of the program is O(n^2) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Randomized | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| A randomized algorithm is an algorithm that uses a random number or random choices during its computation to improve its performance or efficiency. The randomness can be used to find a solution to a problem faster or to approximate the solution with a high probability of correctness. The name comes from the fact that randomness is used to make the algorithm less predictable and more efficient.  Randomized algorithms have a wide range of applications in computer science and other fields. Some examples include:   * Monte Carlo simulations in physics and engineering * Cryptography and security protocols * Machine learning and data analysis * Graph algorithms and network analysis   The steps to use a randomized algorithm depend on the specific problem being solved, but typically involve the following:   * Identify the problem and determine if a randomized algorithm can be used to solve it more efficiently than a deterministic algorithm. * Choose a random number generator or other source of randomness to be used in the algorithm. * Develop the algorithm, incorporating the random elements as necessary. * Analyse the performance of the algorithm and determine if it meets the desired requirements. * Implement the algorithm and test it using sample data. * Fine-tune the algorithm as necessary to improve its performance or efficiency. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| **Program for Randomized algorithm that will shuffle the array elements:**  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100  int main() {  int arr[MAX\_SIZE];  int n, i, j, temp;  // Read input size  printf("Enter the size of the array (<= %d): ", MAX\_SIZE);  scanf("%d", &n);  // Read input array elements  printf("Enter the elements of the array: ");  for (i = 0; i < n; i++) {  scanf("%d", &arr[i]);  }  // Seed the random number generator with current time  srand(time(NULL));  // Shuffle the array using Fisher-Yates algorithm  for (i = n - 1; i >= 1; i--) {  // Generate a random index between 0 and i (inclusive)  j = rand() % (i + 1);  // Swap the elements at index i and j  temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  // Print the shuffled array  printf("Shuffled array: ");  for (i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Enter the size of the array (<= 100): 6  Enter the elements of the array: 2 4 6 8 9 2 | | | | | | | |
| **Sample Output:** | | | | | | | |
| Shuffled array: 2 2 9 4 6 8 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of the code is O(n), where n is the size of the array. This is because the algorithm iterates over the array n-1 times. | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  struct Node {  int data;  struct Node\* next;  };  // Function to create a new node  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to insert a node at the end of the linked list  void insertNode(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  } else {  struct Node\* current = \*head;  while (current->next != NULL) {  current = current->next;  }  current->next = newNode;  }  }  // Function to swap the values of two nodes  void swap(struct Node\* a, struct Node\* b) {  int temp = a->data;  a->data = b->data;  b->data = temp;  }  // Function to shuffle a linked list using Fisher-Yates algorithm  void shuffleLinkedList(struct Node\* head) {  // Count the number of nodes in the linked list  int count = 0;  struct Node\* current = head;  while (current != NULL) {  count++;  current = current->next;  }  // Initialize random seed  srand(time(NULL));  // Traverse the linked list and swap each node with a random node  for (int i = count - 1; i > 0; i--) {  int j = rand() % (i + 1); // Generate a random index between 0 and i (inclusive)  // Find the ith and jth nodes in the linked list  struct Node\* nodeI = head;  struct Node\* nodeJ = head;  for (int k = 0; k < i; k++) {  nodeI = nodeI->next;  }  for (int k = 0; k < j; k++) {  nodeJ = nodeJ->next;  }  // Swap the values of the ith and jth nodes  swap(nodeI, nodeJ);  }  }  // Function to print the linked list  void printLinkedList(struct Node\* head) {  struct Node\* current = head;  while (current != NULL) {  printf("%d ", current->data);  current = current->next;  }  printf("\n");  }  int main() {  struct Node\* head = NULL;  int size;  printf("Enter the size of the linked list: ");  scanf("%d", &size);  printf("Enter the elements of the linked list:\n");  for (int i = 0; i < size; i++) {  int data;  scanf("%d", &data);  insertNode(&head, data);  }  printf("Original linked list: ");  printLinkedList(head);  shuffleLinkedList(head);  printf("Shuffled linked list: ");  printLinkedList(head);  // TODO: Free the memory allocated for the linked list  return 0;  } |
| **Sample Input:** |
| Enter the size of the linked list: 5  Enter the elements of the linked list:  1  2  3  4  5 |
| **Sample Output:** |
| Original linked list: 1 2 3 4 5  Shuffled linked list: 5 3 1 4 2 |
| **Time complexity calculation:** |
| *The overall time complexity of the code is* ***O(n)*** *where 'n' is the size of the linked list.* |