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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *01* | | | | |  |  | | | | |
| **Student Name** | Chandan Raikar | | | | |  |  | | | | |
| **SRN** | 01FE21BEC038 | | | | **Roll Number** | | 32 | | **Division** | A | |
| **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** | | User-defined | | | | | O(1) | | | |
| Recursive | **1** | | Linked list | | | | | O(n) | | | |
| **2** | | Array | | | | | O(n^2) | | | |
| Back tracking | **1** | | 2D=Array | | | | | O(N!) | | | |
| **2** | | Stack | | | | | O(n) | | | |
| Divide and conquer | **1** | | Array | | | | | O(log n) | | | |
| **2** | | Array | | | | | O(nlog n) | | | |
| Dynamic programming | **1** | | 2D-Array | | | | | O(nk) | | | |
| **2** | | Array | | | | | O(a\*n) | | | |
| Greedy | **1** | | Array | | | | | O(n log n) | | | |
| **2** | | Array | | | | | O(n log n) | | | |
| Branch and bound | **1** | | 2D-Array | | | | | O((n-1)!)^2 | | | |
| **2** | | 2D-Array | | | | | O(m^V) | | | |
| Brute force | **1** | | Array | | | | | O(n) | | | |
| **2** | | Array | | | | | O(mn) | | | |
| Randomized | **1** | | Array | | | | | O(n) | | | |
| **2** | | Arrray | | | | | O(N^2) | | | |

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

yes

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

*none*

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

*Mathematic and logical reasoning*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Iterative (The given number is prime or not)* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| Sieve of Eratosthenes algorithm, a popular and efficient algorithm for checking whether a number is prime or not. This program uses an array data structure to store the prime numbers.  1. Start the program.  2. Prompt the user to enter a positive integer.  3. Read the input value from the user and store it in the variable num.  4. If num is less than or equal to 1, print "num is not a prime number" and exit the program.  5. Set the variable isPrime to 1.  6. For i from 2 to num/2 do steps 7-8.  7. Check if num is evenly divisible by i (i.e., num % i == 0).  8. If num is evenly divisible by i, set isPrime to 0 and break out of the loop.  9. If isPrime is still equal to 1, print "num is a prime number".  10. If isPrime is equal to 0, print "num is not a prime number".  11. End the program. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h> //header files –standard input output  #include <stdlib.h> //header files – standard library  #include <math.h> //header files – various mathematical functions  int is\_prime(int n);    int main()  {  int n;    printf("Enter a positive integer: ");  scanf("%d", &n);    if(is\_prime(n))  printf("%d is a prime number.\n", n);  else  printf("%d is not a prime number.\n", n);    return 0;  }    int is\_prime(int n)  {  if(n < 2)  return 0;    int i, j, sqrt\_n = sqrt(n);  int \*sieve = (int \*) calloc(n + 1, sizeof(int));    for(i = 2; i <= sqrt\_n; i++)  {  if(sieve[i] == 0)  {  for(j = i \* i; j <= n; j += i)  {  sieve[j] = 1;  }  }  }    int prime = !sieve[n];    free(sieve);  return prime;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter a positive integer:5* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *5 is a prime number.* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| the overall time complexity of the program is dominated by the loop, which has a time complexity of O(n). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//To calculate perimeter and area*  *#include <stdio.h>*  *typedef struct {*  *double length;*  *double width;*  *}*  *Land;*  *double calculateArea(Land land) {*  *return land.length \* land.width;*  *}*  *double calculatePerimeter(Land land) {*  *return 2 \* (land.length + land.width);*  *}*  *int main() {*  *Land myLand;*  *printf("Enter the length of the land: ");*  *scanf("%lf", &myLand.length);*  *printf("Enter the width of the land: ");*  *scanf("%lf", &myLand.width);*  *double area = calculateArea(myLand);*  *double perimeter = calculatePerimeter(myLand);*  *printf("The area of the land is %.2lf sq. units\n", area);*  *printf("The perimeter of the land is %.2lf units\n", perimeter);*    *return 0;*  *}* |
| **Sample Input:** |
| *Enter the length of the land: 24*  *Enter the width of the land: 26* |
| **Sample Output:** |
| *The area of the land is 600.00 sq. units*  *The perimeter of the land is 100.00 units* |
| **Time complexity calculation:** |
| The time complexity of this program is O(1) because it performs a fixed number of operations, regardless of the size of the land plot. The program reads in two values and then performs some arithmetic operations to calculate the area and perimeter. These operations take constant time and are not dependent on the input size. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Recursive (To find a factorial of given number)* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| C program that uses a recursive function and a linked list data structure to find the factorial of a number  Algorithm:   1. Start with the main function and input a number to find its factorial. 2. Define a linked list data structure with a node containing two members: an integer data and a pointer to the next node. 3. Define the **addNode** function to add a new node to the linked list. 4. Define the **factorial** function to find the factorial of a number recursively. It takes two arguments: the number to find the factorial of and a pointer to the head of the linked list. If the input number is 0, return 1. Otherwise, calculate the factorial by multiplying the input number by the factorial of one less than the input number, and add the result to the linked list using the **addNode** function. Return the factorial. 5. Define the **printList** function to print the linked list. 6. Call the **factorial** function from the main function, passing in the input number and a pointer to the head of the linked list. 7. Print the final factorial and the list of intermediate results using the **printf** and **printList** functions. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *#include <stdio.h>*  *#include <stdlib.h>*  *// Define a linked list node structure*  *struct Node {*  *int data;*  *struct Node\* next;*  *};*  *// Define a function to add a new node to the linked list*  *void addNode(struct Node\*\* head\_ref, int data) {*  *struct Node\* new\_node = (struct Node\*) malloc(sizeof(struct Node));*  *new\_node->data = data;*  *new\_node->next = (\*head\_ref);*  *(\*head\_ref) = new\_node;*  *}*  *// Define a recursive function to find the factorial*  *int factorial(int n, struct Node\*\* head\_ref) {*  *if (n == 0) {*  *return 1;*  *} else {*  *int result = n \* factorial(n - 1, head\_ref);*  *addNode(head\_ref, result);*  *return result;*  *}*  *}*  *// Define a function to print the linked list*  *void printList(struct Node\* node) {*  *while (node != NULL) {*  *printf("%d ", node->data);*  *node = node->next;*  *}*  *printf("\n");*  *}*  *// Define the main function*  *int main() {*  *int n;*  *printf("Enter a number to find its factorial: ");*  *scanf("%d", &n);*  *// Initialize the linked list*  *struct Node\* head = NULL;*  *// Call the factorial function*  *int result = factorial(n, &head);*  *// Print the factorial and the list of intermediate results*  *printf("%d! = %d\n", n, result);*  *printf("Intermediate results: ");*  *printList(head);*  *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter a number to find its factorial: 5* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *5! = 120*  *Intermediate results: 120 24 6 2 1* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| since it loops through each node in the linked list and prints its data. Therefore, the overall time complexity of the program is O(n). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//Programme to find n no of fibonacci numbers*  *#include<stdio.h>*  *int fibonacci(int n) {*  *int fib[n+1];*  *fib[0] = 0;*  *fib[1] = 1;*  *for(int i=2; i<=n; i++) {*  *fib[i] = fib[i-1] + fib[i-2];*  *}*  *return fib[n];*  *}*  *int main() {*  *int n;*  *printf("Enter the number of terms: ");*  *scanf("%d", &n);*  *printf("Fibonacci Series:\n");*  *for(int i=0; i<n; i++) {*  *printf("%d ", fibonacci(i));*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the number of terms: 9* |
| **Sample Output:** |
| *Fibonacci Series:*  *0 1 1 2 3 5 8 13 21* |
| **Time complexity calculation:** |
| The time complexity of the fibonacci function is O(n) since it uses a for loop to calculate the Fibonacci series up to the nth term. The overall time complexity of the program is O(n^2) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Back tracking (N-Queens problem)* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *N-Queens problem*  *Here ‘N’ is pre-defined as 6*  Here is the algorithm of the program:   1. Define the size of the board 2. Define a function isSafe() to check if it's safe to place a queen in a given row and column 3. Define a function solveNQUtil() which will recursively place queens on the board and return true if it's possible to place all the queens on the board 4. Define a function printSolution() to print the solution 5. Define a function solveNQ() which will create an empty board, call solveNQUtil() to place the queens on the board, and print the solution if it exists 6. In the main() function, call the solveNQ() function to solve the N-Queens problem | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *#include <stdio.h>*  *#include <stdbool.h>*  *#define N 6*  *bool isSafe(int board[N][N], int row, int col) {*  *int i, j;*  *// Check for queens in the same row*  *for (i = 0; i < col; i++) {*  *if (board[row][i])*  *return false;*  *}*  *// Check for queens in the upper left diagonal*  *for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {*  *if (board[i][j])*  *return false;*  *}*  *// Check for queens in the lower left diagonal*  *for (i = row, j = col; j >= 0 && i < N; i++, j--) {*  *if (board[i][j])*  *return false;*  *}*  *return true;*  *}*  *bool solveNQUtil(int board[N][N], int col) {*  *if (col == N)*  *return true;*  *for (int i = 0; i < N; i++) {*  *if (isSafe(board, i, col)) {*  *board[i][col] = 1;*  *if (solveNQUtil(board, col + 1))*  *return true;*  *board[i][col] = 0;*  *}*  *}*  *return false;*  *}*  *void printSolution(int board[N][N]) {*  *for (int i = 0; i < N; i++) {*  *for (int j = 0; j < N; j++)*  *printf(" %d ", board[i][j]);*  *printf("\n");*  *}*  *}*  *bool solveNQ() {*  *int board[N][N] = {{0}};*  *if (solveNQUtil(board, 0) == false) {*  *printf("Solution does not exist");*  *return false;*  *}*  *printSolution(board);*  *return true;*  *}*  *int main()*  *{*  *solveNQ();*  *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Pre-defined value = 6* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *0 0 0 1 0 0*  *1 0 0 0 0 0*  *0 0 0 0 1 0*  *0 1 0 0 0 0*  *0 0 0 0 0 1*  *0 0 1 0 0 0* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of this program is O(N!) because there are N possible positions for the first queen, N-2 possible positions for the second queen, N-4 possible positions for the third queen, and so on. This gives us a total of N! possible configurations. The program uses backtracking to eliminate invalid configurations, so the actual number of configurations that need to be checked is much less than N!, but it's still an upper bound on the time complexity. | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//Programme to remove invalid parentheses*  *#include <stdio.h>*  *#include <stdlib.h>*  *#include <string.h>*  *// Stack structure for storing parentheses*  *struct Stack {*  *int top;*  *char\* items;*  *};*  *// Function to initialize stack*  *struct Stack\* createStack(int capacity) {*  *struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));*  *stack->top = -1;*  *stack->items = (char\*)malloc(capacity \* sizeof(char));*  *return stack;*  *}*  *// Function to check if a character is a valid parenthesis*  *int isParenthesis(char ch) {*  *return (ch == '(' || ch == ')');*  *}*  *// Function to check if a pair of parentheses is valid*  *int isValidPair(char ch1, char ch2) {*  *return (ch1 == '(' && ch2 == ')');*  *}*  *// Function to remove invalid parentheses from a string*  *char\* removeInvalidParentheses(char\* str) {*  *int len = strlen(str);*  *// Create a stack for storing parentheses*  *struct Stack\* stack = createStack(len);*  *// Loop through the string*  *for (int i = 0; i < len; i++) {*  *char ch = str[i];*  *// If the character is not a parenthesis, ignore it*  *if (!isParenthesis(ch)) {*  *continue;*  *}*  *// If the character is an opening parenthesis, push it onto the stack*  *if (ch == '(') {*  *stack->items[++stack->top] = ch;*  *}*  *// If the character is a closing parenthesis*  *else if (ch == ')') {*  *// If the stack is empty or the top element is not a valid pair, push the closing parenthesis onto the stack*  *if (stack->top == -1 || !isValidPair(stack->items[stack->top], ch)) {*  *stack->items[++stack->top] = ch;*  *}*  *// If the top element is a valid pair, pop the top element*  *else {*  *stack->top--;*  *}*  *}*  *}*  *// Create a new string for storing the result*  *char\* result = (char\*)malloc((stack->top + 2) \* sizeof(char));*  *int j = 0;*  *// Loop through the stack and copy the remaining parentheses to the result string*  *for (int i = 0; i <= stack->top; i++) {*  *if (stack->items[i] == '(' || stack->items[i] == ')') {*  *result[j++] = stack->items[i];*  *}*  *}*  *result[j] = '\0';*  *// Free the stack memory*  *free(stack->items);*  *free(stack);*  *// Return the result string*  *return result;*  *}*  *// Main function*  *int main() {*  *char str[100];*  *// Read input string from user*  *printf("Enter a string: ");*  *scanf("%[^\n]", str);*  *// Remove invalid parentheses and print the result*  *char\* result = removeInvalidParentheses(str);*  *printf("Result: %s\n", result);*  *// Free the result memory*  *free(result);*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter a string: (((KLE)((Tech))* |
| **Sample Output:** |
| *Result: ((* |
| **Time complexity calculation:** |
| Overall, the time complexity of the program is O(n) + O(n) + O(n) + n\*O(1) + O(n) + O(1) = O(n), where n is the length of the input string. Therefore, the time complexity of the program is linear in the length of the input string. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Divide and conquer (Binary Search)* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *It gives the position number of the given number in the array*   1. Begin 2. Input the size of the array n 3. Create an array of size n 4. Input n elements into the array 5. Input the element to search x 6. Set the left index l to 0 and the right index r to n-1 7. While r >= l do steps 8-14 8. Set the mid index m to (l + r) / 2 9. If the element at index m is equal to x, return m 10. If the element at index m is greater than x, set r to m-1 11. If the element at index m is less than x, set l to m+1 12. End of while loop 13. If the element is not found, return -1 14. End | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *#include <stdio.h>*  *// binary search function*  *int binarySearch(int arr[], int l, int r, int x) {*  *if (r >= l) {*  *int mid = l + (r - l) / 2;*  *if (arr[mid] == x) {*  *return mid;*  *}*  *if (arr[mid] > x) {*  *return binarySearch(arr, l, mid - 1, x);*  *}*  *return binarySearch(arr, mid + 1, r, x);*  *}*  *return -1;*  *}*  *int main() {*  *int n, x, result;*  *printf("Enter the size of the array: ");*  *scanf("%d", &n);*    *int arr[n];*    *printf("Enter the elements of the array: ");*  *for(int i=0; i<n; i++){*  *scanf("%d", &arr[i]);*  *}*    *printf("Enter the element to search: ");*  *scanf("%d", &x);*    *result = binarySearch(arr, 0, n - 1, x);*  *if (result == -1) {*  *printf("Element is not present in array");*  *} else {*  *printf("Element is present at index %d", 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 search: 1* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Element is present at index 0* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of the binary search algorithm implemented in the above code is O(log n), where n is the number of elements in the array. | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//Merge sort (This program arranges the number in ascending order)*  *#include <stdio.h>*  *#include <stdlib.h>*  *void merge(int arr[], int left, int middle, int right) {*  *int i, j, k;*  *int n1 = middle - left + 1;*  *int n2 = right - middle;*  *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[middle + 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++;*  *}*  *}*  *void mergeSort(int arr[], int left, int right) {*  *if (left < right) {*  *int middle = left + (right - left) / 2;*    *mergeSort(arr, left, middle);*  *mergeSort(arr, middle + 1, right);*    *merge(arr, left, middle, right);*  *}*  *}*  *int main() {*  *int n;*  *printf("Enter the number of elements in the array: ");*  *scanf("%d", &n);*    *int arr[n];*  *printf("Enter the elements of the array: ");*  *for (int i = 0; i < n; i++) {*  *scanf("%d", &arr[i]);*  *}*    *mergeSort(arr, 0, n - 1);*    *printf("Sorted array: ");*  *for (int i = 0; i < n; i++)*  *printf("%d ", arr[i]);*  *printf("\n");*    *return 0;*  *}* |
| **Sample Input:** |
| *Enter the number of elements in the array: 6*  *Enter the elements of the array: 8 5 6 7 4 9* |
| **Sample Output:** |
| *Sorted array: 4 5 6 7 8 9* |
| **Time complexity calculation:** |
| The time complexity of Merge Sort is O(nlogn), where n is the number of elements in the array. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Dynamic programming (To find binomial co-efficient)* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *This code helps to find the binomial co-efficients by using Dynamic programming algorithm*  *1. Start*  *2. Read the values of n and k*  *3. Declare a 2D array C of size (n+1) by (k+1)*  *4. For i = 0 to n do*  *5. For j = 0 to min(i, k) do*  *6. If j = 0 or j = i then*  *7. Set C[i][j] = 1*  *8. Else*  *9. Set C[i][j] = C[i-1][j-1] + C[i-1][j]*  *10. End For*  *11. End For*  *12. Return C[n][k]*  *13. End* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *#include <stdio.h>*  *// Function to calculate the binomial coefficient*  *int binomial\_coefficient(int n, int k) {*  *int i, j;*  *int C[n+1][k+1];*  *// Calculate the binomial coefficient using dynamic programming*  *for (i = 0; i <= n; i++) {*  *for (j = 0; j <= k && j <= i; j++) {*  *if (j == 0 || j == i)*  *C[i][j] = 1;*  *else*  *C[i][j] = C[i-1][j-1] + C[i-1][j];*  *}*  *}*    *return C[n][k];*  *}*  *int main() {*  *int n, k;*    *printf("Enter the values of n and k: ");*  *scanf("%d %d", &n, &k);*    *printf("The binomial coefficient C(%d,%d) is %d\n", n, k, binomial\_coefficient(n, k));*    *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter the values of n and k: 5 2* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *The binomial coefficient C(5,2) is 10* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of the above program is O(n*k), where n and k are the two inputs to the program.* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//This program is to know the amount for which the client want to make change*  *#include <stdio.h>*  *#include <limits.h>*  *int coinChange(int coins[], int n, int amount) {*  *int table[amount+1];*  *table[0] = 0;*  *for (int i = 1; i <= amount; i++) {*  *table[i] = INT\_MAX;*  *}*  *for (int i = 0; i < n; i++) {*  *for (int j = coins[i]; j <= amount; j++) {*  *if (table[j-coins[i]] != INT\_MAX && table[j-coins[i]] + 1 < table[j]) {*  *table[j] = table[j-coins[i]] + 1;*  *}*  *}*  *}*  *return table[amount] == INT\_MAX ? -1 : table[amount];*  *}*  *int main() {*  *int coins[100], n, amount;*  *printf("Enter the number of coins: ");*  *scanf("%d", &n);*  *printf("Enter the coin denominations:\n");*  *for (int i = 0; i < n; i++) {*  *scanf("%d", &coins[i]);*  *}*  *printf("Enter the amount for which you want to make change: ");*  *scanf("%d", &amount);*  *int minCoins = coinChange(coins, n, amount);*  *if (minCoins == -1) {*  *printf("It is not possible to make change for the given amount using the given coins.\n");*  *} else {*  *printf("The minimum number of coins required to make %d rupees is %d\n", amount, minCoins);*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the number of coins: 4*  *Enter the coin denominations:*  *1 2 4 5*  *Enter the amount for which you want to make change: 26* |
| **Sample Output:** |
| *The minimum number of coins required to make 26 rupees is 6* |
| **Time complexity calculation:** |
| Time complexity: O(amount\*n), where nis the number of coins.  In above time complexity ‘a’ is referred as amount |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Greedy (Program for Activity selection)* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *This program helps to select the activity in the given no of activities*  *Algorithm*   1. Sort the activities based on their end times in ascending order. 2. Select the first activity in the sorted list and print it as part of the solution. 3. For each subsequent activity in the sorted list: a. If the start time of the activity is greater than or equal to the end time of the previously selected activity, select the current activity and print it as part of the solution. b. Otherwise, discard the current activity. 4. Repeat step 3 until all activities in the sorted list have been considered. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *#include <stdio.h>*  *#include <stdlib.h>*  *typedef struct Interval {*  *int start;*  *int end;*  *} Interval;*  *int compareIntervals(const void\* a, const void\* b) {*  *Interval\* intervalA = (Interval\*)a;*  *Interval\* intervalB = (Interval\*)b;*  *return (intervalA->end - intervalB->end);*  *}*  *void printMaxActivities(Interval intervals[], int n) {*  *qsort(intervals, n, sizeof(Interval), compareIntervals);*  *int i = 0;*  *printf("Selected Activities: ");*  *printf("(%d, %d) ", intervals[i].start, intervals[i].end);*  *for (int j = 1; j < n; j++) {*  *if (intervals[j].start >= intervals[i].end) {*  *printf("(%d, %d) ", intervals[j].start, intervals[j].end);*  *i = j;*  *}*  *}*  *printf("\n");*  *}*  *int main() {*  *int n;*  *printf("Enter the number of activities: ");*  *scanf("%d", &n);*  *Interval\* intervals = (Interval\*)malloc(n \* sizeof(Interval));*  *printf("Enter the start and end times of the activities:\n");*  *for (int i = 0; i < n; i++) {*  *scanf("%d %d", &intervals[i].start, &intervals[i].end);*  *}*  *printMaxActivities(intervals, n);*  *free(intervals);*  *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter the number of activities: 5*  *Enter the start and end times of the activities:*  *1 3*  *2 6*  *4 8*  *9 4*  *7 9* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Selected Activities: (1, 3) (9, 4) (4, 8)* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of the above program is dominated by the time complexity of the sorting algorithm, which is O(n log n), where n is the number of activities. | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//Program to sequence the job from the given data deadline and profit*  *#include <stdio.h>*  *#include <stdlib.h>*  *// Define a job structure to store each job's ID, deadline, and profit*  *struct job {*  *int id;*  *int deadline;*  *int profit;*  *};*  *// Compare function for sorting jobs in decreasing order of profit*  *int cmpfunc(const void\* a, const void\* b) {*  *return ((struct job\*)b)->profit - ((struct job\*)a)->profit;*  *}*  *// Function to find the maximum profit by scheduling jobs in decreasing order of profit*  *void findMaxProfit(struct job\* jobs, int n) {*  *int i, j;*  *int max\_deadline = 0;*  *int max\_profit = 0;*  *struct job\* schedule = (struct job\*)malloc(n \* sizeof(struct job)); // Initialize a schedule to store the selected jobs*  *qsort(jobs, n, sizeof(struct job), cmpfunc); // Sort jobs in decreasing order of profit*  *// Find the maximum deadline among all jobs*  *for (i = 0; i < n; i++) {*  *if (jobs[i].deadline > max\_deadline) {*  *max\_deadline = jobs[i].deadline;*  *}*  *}*  *// Initialize the schedule with placeholder jobs*  *for (i = 0; i < max\_deadline; i++) {*  *schedule[i].id = -1;*  *schedule[i].deadline = -1;*  *schedule[i].profit = -1;*  *}*  *// Schedule jobs in decreasing order of profit*  *for (i = 0; i < n; i++) {*  *for (j = jobs[i].deadline - 1; j >= 0; j--) {*  *if (schedule[j].id == -1) {*  *schedule[j] = jobs[i];*  *max\_profit += jobs[i].profit;*  *break;*  *}*  *}*  *}*  *// Print the schedule and maximum profit*  *printf("Schedule:\n");*  *for (i = 0; i < max\_deadline; i++) {*  *if (schedule[i].id != -1) {*  *printf("Job %d (deadline: %d, profit: %d)\n", schedule[i].id, schedule[i].deadline, schedule[i].profit);*  *}*  *}*  *printf("Maximum profit: %d\n", max\_profit);*    *free(schedule); // Free memory allocated to the schedule*  *}*  *int main() {*  *int n, i;*  *struct job\* jobs;*    *printf("Enter the number of jobs: ");*  *scanf("%d", &n);*    *jobs = (struct job\*)malloc(n \* sizeof(struct job)); // Allocate memory for the jobs*    *// Read job details from user input*  *for (i = 0; i < n; i++) {*  *printf("Enter the deadline and profit of job %d: ", i + 1);*  *scanf("%d %d", &jobs[i].deadline, &jobs[i].profit);*  *jobs[i].id = i + 1;*  *}*    *findMaxProfit(jobs, n); // Find the maximum profit by scheduling jobs*    *free(jobs); // Free memory allocated to the jobs*    *return 0;*  *}* |
| **Sample Input:** |
| *Enter the number of jobs: 3*  *Enter the deadline and profit of job 1: 2 50*  *Enter the deadline and profit of job 2: 4 70*  *Enter the deadline and profit of job 3: 6 90* |
| **Sample Output:** |
| *Schedule:*  *Job 1 (deadline: 2, profit: 50)*  *Job 2 (deadline: 4, profit: 70)*  *Job 3 (deadline: 6, profit: 90)*  *Maximum profit: 210* |
| **Time complexity calculation:** |
| Therefore, the overall time complexity of the job sequencing problem algorithm can be O(n^2) or O(n log n), depending on the method used to find the latest available slot for each job. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Branch and Bound (Program to solve the travelling salesman problem)* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| The Traveling Salesman Problem (TSP) is a well-known optimization problem that involves finding the shortest possible route that visits a given set of cities and returns to the starting city. Branch and Bound algorithm can be used to solve this problem efficiently.   1. Read the number of cities and the adjacency matrix from the user. 2. Call the TSP function with the starting city, current cost, and level 1. 3. In the TSP function, mark the current city as visited and add it to the current path. 4. If all cities have been visited, check if the current cost plus the cost of returning to the starting city is less than the minimum cost. If so, update the minimum cost path and cost. 5. If there are unvisited cities, compute the lower bound for each unvisited city and add it to the priority queue. If the bound is less than the minimum cost, explore | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#include <limits.h>*  *#define MAX\_N 10 // Maximum number of cities*  *int n; // Number of cities*  *int graph[MAX\_N][MAX\_N]; // Adjacency matrix*  *int visited[MAX\_N]; // Array to keep track of visited cities*  *int path[MAX\_N + 1]; // Array to store the current path*  *int min\_path[MAX\_N + 1]; // Array to store the minimum cost path*  *int min\_cost = INT\_MAX; // Minimum cost*  *int num\_nodes = 0; // Number of nodes explored*  *void TSP(int curr\_city, int curr\_cost, int level)*  *{*  *int i, j, next\_city, next\_cost, bound;*  *visited[curr\_city] = 1;*  *path[level] = curr\_city;*  *if (level == n) {*  *num\_nodes++;*  *path[level + 1] = path[1]; // Return to the starting city*  *if (curr\_cost + graph[curr\_city][0] < min\_cost) {*  *// Update the minimum cost path*  *min\_cost = curr\_cost + graph[curr\_city][0];*  *for (i = 1; i <= n + 1; i++) {*  *min\_path[i] = path[i];*  *}*  *}*  *} else {*  *// Compute the lower bound for each unvisited city*  *for (i = 0; i < n; i++) {*  *if (!visited[i]) {*  *next\_city = i;*  *next\_cost = curr\_cost + graph[curr\_city][next\_city];*  *bound = next\_cost;*  *for (j = 0; j < n; j++) {*  *if (!visited[j]) {*  *bound += graph[next\_city][j];*  *}*  *}*  *if (bound < min\_cost) {*  *TSP(next\_city, next\_cost, level + 1);*  *}*  *}*  *}*  *}*    *visited[curr\_city] = 0;*  *}*  *int main()*  *{*  *int i, j;*  *printf("Enter the number of cities (maximum %d): ", MAX\_N);*  *scanf("%d", &n);*  *printf("Enter the adjacency matrix:\n");*  *for (i = 0; i < n; i++) {*  *for (j = 0; j < n; j++) {*  *scanf("%d", &graph[i][j]);*  *}*  *}*  *// Start the algorithm from the first city*  *TSP(0, 0, 1);*  *// Print the minimum cost path*  *printf("Minimum cost path: ");*  *for (i = 1; i <= n + 1; i++) {*  *printf("%d ", min\_path[i]);*  *}*  *printf("\nMinimum cost: %d\n", min\_cost);*  *printf("Number of nodes explored: %d\n", num\_nodes);*    *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter the number of cities (maximum 10): 3*  *Enter the adjacency matrix:*  *1 2 3*  *4 5 6*  *7 8 9* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Minimum cost path: 0 1 2 0*  *Minimum cost: 15*  *Number of nodes explored: 1* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of the Branch and Bound algorithm for the Traveling Salesman Problem is O((n-1)!)^2. This is because the algorithm generates all possible permutations of n-1 cities and then for each permutation, it computes the cost of adding the remaining city to the path. The number of permutations is (n-1)!, and for each permutation, the cost is computed by examining each of the (n-1) edges in the path, resulting in a total time complexity of O((n-1)! \* (n-1)). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//This program helps to assign a color to each vertex such that no two adjacent vertices have the same colour*  *#include<stdio.h>*  *#include<stdlib.h>*  *#define V 3*  *int g[V][V]; // adjacency matrix to store the graph*  *int color[V]; // array to store the color assigned to each vertex*  *int m; // number of colors*  *// function to check if it is safe to color the vertex with the given color*  *int isSafe(int v, int c)*  *{*  *for(int i=0; i<V; i++)*  *{*  *if(g[v][i] && color[i] == c)*  *{*  *return 0;*  *}*  *}*  *return 1;*  *}*  *// function to assign colors to vertices using the Branch and Bound algorithm*  *void graphColor(int v) {*  *if(v == V)*  *{ // all vertices have been colored*  *for(int i=0; i<V; i++)*  *{*  *printf("%d ", color[i]);*  *}*  *printf("\n");*  *return;*  *}*  *// assign colors to the current vertex using the minimum number of colors possible*  *for(int c=1; c<=m; c++)*  *{*  *if(isSafe(v, c))*  *{*  *color[v] = c;*  *graphColor(v+1);*  *color[v] = 0; // backtrack*  *}*  *}*  *}*  *int main()*  *{*  *printf("Enter the adjacency matrix for the graph:\n");*  *for(int i=0; i<V; i++) {*  *for(int j=0; j<V; j++) {*  *scanf("%d", &g[i][j]);*  *}*  *}*  *printf("Enter the number of colors: ");*  *scanf("%d", &m);*  *graphColor(0);*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the adjacency matrix for the graph:*  *1 2 3*  *4 5 6*  *7 8 9*  *Enter the number of color: 3* |
| **Sample Output:** |
| *1 2 3*  *1 3 2*  *2 1 3*  *2 3 1*  *3 1 2*  *3 2 1* |
| **Time complexity calculation:** |
| The time complexity of this algorithm is O(m^V), where m is the number of colors and V is the number of vertices in the graph. This is because we need to try all possible color combinations for each vertex in the worst case. However, in practice, the time complexity can be reduced by using heuristics to choose the vertex with the minimum degree first, which is likely to have fewer color options. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Brute force (Program to search of element in the array)* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| Program using brute force algorithm to search for an element in an array  1. Initialize a one-dimensional integer array of size n to store the input elements.  2. Take input from the user for the number of elements n and the values of the elements.  3. Take input from the user for the element to be searched x.  4. Initialize an integer variable index to -1.  5. Use a loop to traverse the array from index 0 to n-1:  a. If the current element in the array is equal to x, set index to the index of that element and break out of the loop.  6. If index is still -1 after the loop, the element was not found in the array.  7. Otherwise, print the index of the element where it was found in the array. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *#include <stdio.h>*  *#include <stdlib.h>*  *// Function to search for an element using brute force algorithm*  *int search(int arr[], int n, int x)*  *{*  *int i;*  *for (i = 0; i < n; i++)*  *{*  *if (arr[i] == x)*  *return i;*  *}*  *return -1;*  *}*  *// Driver function*  *int main()*  *{*  *int n, x, i;*  *printf("Enter the number of elements in 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("Enter the element to be searched: ");*  *scanf("%d", &x);*  *// Call the search function*  *int index = search(arr, n, x);*  *// Print the result*  *if (index == -1)*  *printf("Element not found\n");*  *else*  *printf("Element found at index %d\n", index);*  *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter the number of elements in the array: 5*  *Enter the elements of the array:*  *4*  *2*  *8*  *9*  *7*  *Enter the element to be searched: 2* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Element found at index 1* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of this algorithm is O(n), where n is the number of elements in the array. This is because in the worst case scenario, we may have to check every element in the array to find the desired element. | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//Program to find the position of the given pattern*  *#include <stdio.h>*  *#include <string.h>*    *#define MAX\_TEXT\_SIZE 1000*  *#define MAX\_PATTERN\_SIZE 100*    *int brute\_force\_search(char text[], char pattern[]);*    *int main()*  *{*  *char text[MAX\_TEXT\_SIZE];*  *char pattern[MAX\_PATTERN\_SIZE];*  *int position;*    *printf("Enter the text: ");*  *fgets(text, MAX\_TEXT\_SIZE, stdin);*    *printf("Enter the pattern: ");*  *fgets(pattern, MAX\_PATTERN\_SIZE, stdin);*    *// remove newline characters from input strings*  *text[strcspn(text, "\n")] = '\0';*  *pattern[strcspn(pattern, "\n")] = '\0';*    *position = brute\_force\_search(text, pattern);*    *if (position == -1)*  *printf("Pattern not found\n");*  *else*  *printf("Pattern found at position %d\n", position);*    *return 0;*  *}*    *int brute\_force\_search(char text[], char pattern[])*  *{*  *int text\_length = strlen(text);*  *int pattern\_length = strlen(pattern);*  *int i, j;*    *for (i = 0; i <= text\_length - pattern\_length; i++) {*  *for (j = 0; j < pattern\_length; j++) {*  *if (text[i + j] != pattern[j])*  *break;*  *}*  *if (j == pattern\_length)*  *return i;*  *}*  *return -1;*  *}* |
| **Sample Input:** |
| *Enter the text: KLE TECH*  *Enter the pattern: TEC* |
| **Sample Output:** |
| *Pattern found at position 4* |
| **Time complexity calculation:** |
| The time complexity of the Brute Force algorithm for string matching used in the above code is O(mn), where m is the length of the pattern and n is the length of the text. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Randomized (Program to give random output treatment and control count)* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *This program helps to give random output numbers, the number of patients getting treatment and in control count*  **Algorithm:**   1. Seed the random number generator with the current time. 2. Get the number of trials from the user. 3. Initialize two counters for treatment and control groups to 0. 4. For each trial, generate a random value (0 or 1). 5. If the random value is 0, increment the treatment counter. Otherwise, increment the control counter. 6. Print the final counts for treatment and control groups. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#include <time.h>*  *// Function to generate random numbers*  *int randnum(int a, int b) {*  *return rand() % (b-a+1) + a;*  *}*  *// Function to perform the randomized controlled trial*  *void randomized\_trial(int num\_trials)*  *{*  *int treatment\_count = 0;*  *int control\_count = 0;*  *for (int i = 0; i < num\_trials; i++)*  *{*  *int rand\_val = randnum(0, 1);*  *if (rand\_val == 0) {*  *treatment\_count++;*  *} else {*  *control\_count++;*  *}*  *}*  *printf("Treatment count: %d\n", treatment\_count);*  *printf("Control count: %d\n", control\_count);*  *}*  *int main()*  *{*  *srand(time(0)); // Seed the random number generator with current time*  *int num\_trials;*  *printf("Enter number of trials: ");*  *scanf("%d", &num\_trials);*  *randomized\_trial(num\_trials);*  *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter number of trials: 5000* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Treatment count: 2537*  *Control count: 2463* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of this program is O(n), where n is the number of trials. This is because we are performing a single loop with n iterations, and each iteration involves generating a random number and incrementing one of two counters. The time complexity of generating a random number using **rand()** is O(1), so it does not contribute to the overall time complexity. Therefore, the dominant operation in the program is the loop with n iterations, giving a time complexity of O(n). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *//This Program helps give a shuffled deck of cards*  *#include <stdio.h>*  *#include <stdlib.h>*  *#include <time.h>*  *int main()*  *{*  *int deck\_size, i, j, temp;*  *// Ask the user for the size of the deck*  *printf("Enter the size of the deck: ");*  *scanf("%d", &deck\_size);*  *int deck[deck\_size];*  *// Initialize the deck with the standard order of cards*  *for (i = 0; i < deck\_size; i++)*  *{*  *deck[i] = i;*  *}*  *// Seed the random number generator*  *srand(time(NULL));*  *// Shuffle the deck using the Fisher-Yates shuffle algorithm*  *for (i = deck\_size - 1; i > 0; i--)*  *{*  *j = rand() % (i + 1);*  *temp = deck[i];*  *deck[i] = deck[j];*  *deck[j] = temp;*  *}*  *// Print the shuffled deck*  *printf("Shuffled deck:\n");*  *for (i = 0; i < deck\_size; i++)*  *{*  *printf("%d ", deck[i]);*  *}*  *printf("\n");*  *// Calculate the time complexity of the algorithm*  *// The outer loop runs N times, where N is the size of the deck*  *// The inner loop runs on average N/2 times*  *// Therefore, the time complexity is O(N^2)*  *printf("Time complexity: O(N^2)\n");*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the size of the deck: 52* |
| **Sample Output:** |
| *Shuffled deck:*  *30 4 41 26 1 48 16 43 15 50 20 40 36 29 42 19 37 8 9 14 51 44 5 22 24 13 18 2 3 25 7 45 11 23 17 32 12 49 27 39 33 28 0 21 38 34 35 47 10 6 31 46* |
| **Time complexity calculation:** |
| The outer loop runs N times, where N is the size of the deck. The inner loop runs on average N/2 times, since we are choosing a random index between 0 and i for each iteration. Therefore, the time complexity of the algorithm is O(N^2). |