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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *01* | | | | |  |  | | | | |
| **Student Name** | Omkareshwar Bhangari | | | | |  |  | | | | |
| **SRN** | 01FE21BEC150 | | | | **Roll Number** | | 340 | | **Division** | C | |
| **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** | | NULL | | | | | (n \* (n + 1)) / 2 | | | |
| **2** | | ARRAY | | | | | O(n) | | | |
| Recursive | **1** | | LINKED LIST | | | | | O(n^2) | | | |
| **2** | | ARRAY | | | | | O(n) | | | |
| Back tracking | **1** | | ARRAY | | | | | O(I^2) | | | |
| **2** | | ARRAY | | | | | O(2^(n^2)) | | | |
| Divide and conquer | **1** | | ARRAY | | | | | O(nlogn) | | | |
| **2** | | ARRAY | | | | | O(nlogn) | | | |
| Dynamic programming | **1** | | ARRAY | | | | | O(n) | | | |
| **2** | | ARRAY | | | | | O(n^3) | | | |
| Greedy | **1** | | ARRAY | | | | | O(n^2) | | | |
| **2** | | ARRAY | | | | | O(n) | | | |
| Branch and bound | **1** | | ARRAY | | | | | O(n^2) | | | |
| **2** | | ARRAY | | | | | O(n) | | | |
| Brute force | **1** | | ARRAY | | | | | O((n-m)+1) | | | |
| **2** | | ARRAY | | | | | O(n!) | | | |
| Randomized | **1** | | ARRAY | | | | | O(n) | | | |
| **2** | | ARRAY | | | | | O(n^2) | | | |

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

*<Write your answer here>*

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

*<Write your answer here>*

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

*<Write your answer here>*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Iterative** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| 1. Start the program.  2. Declare the function count\_digits that takes an integer num as input and returns the count of digits.  3. Declare an integer variable count and set it to 0.  4. Output the message: "Enter a number: ".  5. Read an integer from the user and store it in the variable num.  6. Call the function count\_digits with the argument num and assign the result to the variable count.  7. Output the message: "Total digits in number num is: count".  8. End the program. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *#include <stdio.h>*  *void printpatt(int n) {*  *for (int i = 1; i <= n; i++) { // Loop for rows*  *for (int j = 1; j <= i; j++) { // Loop for columns*  *printf("\* ");*  *}*  *printf("\n");*  *}*  *}*  *int main() {*  *int n ;*  *printf(“enter no of rows”\n);*  *scanf(“%d”,&n)*  *printpatt(n);*  *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| ***5*** | | | | | | | |
| **Sample Output:** | | | | | | | |
| \*  \* \*  \* \* \*  \* \* \* \*  \* \* \* \* \* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| 1 + 2 + 3 + ... + n = (n \* (n + 1)) / 2 | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *# include<stdio.h>*  *#define size 5*  *void read(int a[size], int n)*  *{*  *int i;*  *for(i=0;i<n;i++)*  *scanf("%d",&a[i]);*  *}*  *void display(int a[size],int n)*  *{*  *int i;*  *for (i=0;i<n;i++)*  *{*  *printf("%d\t",a[i]);*  *}*  *main()*  *{*  *int a[size],n,b;*  *printf("enter no of elements");*  *scanf("%d",&n);*  *read(a,n);*  *display(a,n);*  *}*  *}* |
| **Sample Input:** |
| n=5  1  2  3  4  5 |
| **Sample Output:** |
| 1 2 3 4 5 |
| **Time complexity calculation:** |
| read func complexity is O(n)  display O(N)  overall O(n) |

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

*<Write your answer here>*

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

*<Write your answer here>*

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

*<Write your answer here>*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Recursive** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *1. Start the program.*  *2. Declare the function count\_digits that takes an integer num as input and returns the count of digits.*  *3. Declare a static integer variable count and set it to 0. (The static keyword ensures that the variable retains its value between recursive calls).*  *4. If num is greater than 0, execute the following steps: a. Increment count by 1 to count the current digit. b. Call the function count\_digits recursively with the argument num/10 to process the remaining digits.*  *5. If num is not greater than 0 (i.e., num has become 0), return count.*  *6. In the main function: a. Declare an integer variable num. b. Declare an integer variable count and set it to 0. c. Output the message: "Enter a number: ". d. Read an integer from the user and store it in the variable num. e. Call the function count\_digits with the argument num and assign the result to the variable count. f. Output the message: "Total digits in number num is: count".*  *7. End the program.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  typedef struct Node \*node;  void pushin(node\*\* head, int newdata) {  node t;  t = (struct Node\*) malloc(sizeof(struct Node));  t->data = newdata;  t->next = \*head;  \*head = t;  }  void printlistrecursive(node nod) {  if (nod == NULL) {  printf("\n");  return;  }  printf("%d ", nod->data);  printlistrecursive(nod->next);  }  // Function to read integers from user and add them to the linked list using recursion  void readlistrecursive(node\*\* head) {  int num;  scanf("%d", &num);  if (num == -1) {  return;  }  pushin(head, num);  readlistrecursive(head);  }  int main() {  // Create an empty linked list  node head = NULL;  // Ask the user to enter integers to add to the linked list  printf("Enter integers to add to the linked list (enter -1 to stop):\n");  readlistrecursive(&head);  // Print the contents of the linked list  printf("Contents of the linked list: ");  printlistrecursive(head);  return 0;  } | | | | | | | |
| **Sample Output:** | | | | | | | |
| **Input= 9 5 7 2 8**  **Output= 8 2 7 5 9** | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n^2) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  int factorialfind(int n);  int main() {  int num;  printf("Enter integer to find the factorial ");  scanf("%d", &num);  printf("factorial = %d", factorialfind(num));  return 0;  }  int factorialfind(int n) {  if (n != 1)  return n \* factorialfind(n - 1);  else  return n;  } |
| **Sample Input:** |
| 4 |
| **Sample Output:** |
| 24 |
| **Time complexity calculation:** |
| *T(n) = T(n-1) + O(1) = T(n-2) + O(1) + O(1) = T(n-3) + O(1) + O(1) + O(1) = ... = T(1) + (n-1)\*O(1) = O(n)* |

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

*<Write your answer here>*

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

*<Write your answer here>*

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

*<Write your answer here>*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Back tracking** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *1. Start the program.*  *2. Declare an integer array a to represent the positions of queens and initialize a variable count to 0.*  *3. Define the function place that takes a position pos as input and checks if a queen can be placed at that position without conflicting with the existing queens. a. Iterate through each previous row (from 1 to pos-1) and check if there is a queen in the same column or diagonally. b. If a conflict is found, return 0; otherwise, return 1.*  *4. Define the function printsol that takes an integer n as input and prints a solution (valid arrangement) of the queens on the chessboard. a. Iterate through each row from 1 to n. b. Iterate through each column from 1 to n and print "Q" if a queen is placed in that position, or "\*" otherwise. c. Print a newline character after each row.*  *5. Define the function queen that takes an integer n as input and solves the N-queens problem using backtracking. a. Initialize k to 1 (representing the current row) and set a[k] to 0 (representing the column of the queen in the first row). b. While k is not 0 (indicating all rows have been processed):*  *• Increment a[k] by 1 to try the next column in the current row.*  *• While a[k] is less than or equal to n and the queen cannot be placed at position k, increment a[k] by 1 to try the next column.*  *• If a[k] is less than or equal to n, a valid position is found:*  *• If k is equal to n, all queens are placed, so call printsol(n) to print the solution.*  *• Otherwise, increment k and set a[k] to 0 to move to the next row.*  *• If a[k] is greater than n, no valid position is found, so decrement k to backtrack to the previous row.*  *6. In the main function: a. Declare an integer variable n to represent the number of queens. b. Output the message: "Enter the Number of queens". c. Read an integer from the user and store it in the variable n. d. Call the function queen(n) to solve the N-queens problem. e. Output the message: "Total Number of Solutions = count".*  *7. End the program.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  #define I 5  void printsol(int arr[I][I]) {  for (int i = 0; i < I; i++) {  for (int j = 0; j < I; j++) {  printf("%d ", arr[i][j]);  }  printf("\n");  }  }  int safecheck(int arr[I][I], int row, int col) {  // Check the row on the left side  for (int i = 0; i < col; i++) {  if (arr[row][i]==1) {  return 0;  }  }    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {  if (arr[i][j]) {  return 0;  }  }  for (int i = row, j = col; j >= 0 && i < I; i++, j--) {  if (arr[i][j]) {  return 0;  }  }  return 1;  }  int solvequeens(int arr[I][I], int col) {  if (col >= I) {  return 1;  }  for (int i = 0; i < I; i++) {  if (safecheck(arr, i, col)) {  arr[i][col] = 1;  if (solvequeens(arr, col + 1)) {  return 1;  }  arr[i][col] = 0;  }  }  return 0;  }  int main() {  int arr[I][I] = {0};  if (solvequeens(arr, 0)) {  printf("Solution:\n");  printsol(arr);  } else {  printf("No solution found.\n");  }  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| I =5 | | | | | | | |
| **Sample Output:** | | | | | | | |
| 1 0 0 0 0  0 0 0 1 0  0 1 0 0 0  0 0 0 0 1  0 0 1 0 0 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(I^2) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include<stdio.h>  #define N 4  // Function to print the solution matrix  void printsol(int arr[N][N])  {  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++)  printf(" %d ", arr[i][j]);  printf("\n");  }  }  // Function to check if x, y is valid index for N\*N maze  int is\_safe(int maze[N][N], int x, int y)  {  // if (x, y outside maze) return false  if (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1)  return 1;    return 0;  }  // Function to solve Rat in a Maze problem using backtracking  int inmazeuntil(int maze[N][N], int x, int y, int arr[N][N])  {  // if (x, y is goal) return true  if (x == N - 1 && y == N - 1) {  arr[x][y] = 1;  return 1;  }    // Check if maze[x][y] is valid  if (is\_safe(maze, x, y) == 1) {  // mark x, y as part of solution path  arr[x][y] = 1;    // Move forward in x direction  if (inmazeuntil(maze, x + 1, y, arr) == 1)  return 1;    // If moving in x direction doesn't give solution  // then Move down in y direction  if (inmazeuntil(maze, x, y + 1, arr) == 1)  return 1;    // If moving in y direction also doesn't give arrution  // then backtrack and unmark x,y as part of solution path  arr[x][y] = 0;  return 0;  }    return 0;  }  // Function to solve Rat in a Maze problem  void mazesol(int maze[N][N])  {  int arr[N][N] = { { 0, 0, 0, 0 },  { 0, 0, 0, 0 },  { 0, 0, 0, 0 },  { 0, 0, 0, 0 } };    if (inmazeuntil(maze, 0, 0, arr) == 0) {  printf("Solution doesn't exist");  return;  }    printsol(arr);  }  int main()  {  int maze[N][N] = { {0, 1, 1, 0 },  { 1, 1, 0, 0 },  { 0, 1, 0, 1 },  { 1, 1, 1, 1 } };    mazesol(maze);  return 0;  } |
| **Sample Output:** |
| 0 1 1 0  1 1 0 0  0 1 0 1  1 1 1 1 |
| **Time complexity calculation:** |
| O(2^(N^2)) |

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

*<Write your answer here>*

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

*<Write your answer here>*

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

*<Write your answer here>*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Divide and conquer** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *1. Start the program.*  *2. Declare variables: A[1000] (array), n, k, f, i, j, s, e, t (doubles).*  *3. Print "Enter n value" to the console.*  *4. Read the value of n from the user.*  *5. Initialize the array A with values from 0 to n-1.*  *6. Print "Array Elements are ..." to the console.*  *7. Iterate from i = 0 to n-1:*  *• Print the value of A[i] to the console.*  *8. Print "Enter Key element" to the console.*  *9. Read the value of k from the user.*  *10. Record the starting time using the clock() function and assign it to s.*  *11. Perform a delay loop:*  *• Iterate i from 0 to 9999.*  *• Iterate j from 0 to 9999.*  *12. Call the Binary\_Search function with arguments A, k, 0, and n-1 and assign its result to f.*  *13. Record the ending time using the clock() function and assign it to e.*  *14. Calculate the total time taken by subtracting s from e and dividing it by CLK\_TCK. Assign the result to t.*  *15. Print "Time taken = " followed by the value of t to the console.*  *16. Check the value of f:*  *• If f is equal to 1, print "Key element found" to the console.*  *• Otherwise, print "Key element not found" to the console.*  *17. End the program.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| *#include <stdio.h>*  *void mergein(int arr[], int left[], int right[], int leftlenght, int rightlenght) {*  *int i = 0, j = 0, k = 0;*  *while (i < leftlenght && j < rightlenght) {*  *if (left[i] < right[j]) {*  *arr[k++] = left[i++];*  *} else {*  *arr[k++] = right[j++];*  *}*  *}*  *while (i < leftlenght) {*  *arr[k++] = left[i++];*  *}*  *while (j < rightlenght) {*  *arr[k++] = right[j++];*  *}*  *}*  *void mergesort(int arr[], int n) {*  *if (n < 2) {*  *return;*  *}*  *int mid = n / 2;*  *int left[mid], right[n - mid];*  *for (int i = 0; i < mid; i++) {*  *left[i] = arr[i];*  *}*  *for (int i = mid; i < n; i++) {*  *right[i - mid] = arr[i];*  *}*  *mergesort(left, mid);*  *mergesort(right, n - mid);*  *mergein(arr, left, right, mid, n - mid);*  *}*  *int main() {*  *int arr[] = {6, 3, 7, 1, 1, 4, 2};*  *int n = sizeof(arr) / sizeof(arr[0]);*  *mergesort(arr, n);*  *printf("Sorted array is: ");*  *for (int i = 0; i < n; i++) {*  *printf("%d ", arr[i]);*  *}*  *printf("\n");*  *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *6, 3, 7, 1, 1, 4, 2* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Sorted array is: 1 1 2 3 4 6 7* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n\*log(n)) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#include<time.h>*  *#define M 15000*  *int A[M];*  *void quicksorting( int low , int high);*  *int part(int low , int high);*  *void swap(int \*a , int \*b);*  *int main()*  *{*  *int n,i,j;*  *int low , high;*  *clock\_t s , e ;*  *double cpu\_exe\_t;*  *printf("\nPlease enter the size of the array :");*  *scanf("%d",&n);*  */\*printf("\nPlease enter the elements of the array :");\*/*  *for(i=0;i<n;i++)*  *{*  *A[i]=rand()%100; /\*scanf("%d",&A[i]);\*/*  *}*  *printf("\nThe array elements are :\n");*  *for(i=0;i<n;i++)*  *{*  *printf("%d\t",A[i]);*  *}*  *s=clock();*  *for(j=0;j<1000;j++) //Delay loops*  *for(i=0;i<1000;i++)*  *{*  *low=0;*  *high=n-1;*  *quicksorting(low,high);*  *}*  *e=clock();*  *cpu\_exe\_t=(double)(e-s)/CLK\_TCK;*  *printf("\nThe sorted array is :\n");*  *for(i=0;i<n;i++)*  *{*  *printf("%d\t",A[i]);*  *}*  *printf("\nCPU execution time is %lf",cpu\_exe\_t);*  *return 0;*  *}*  *int part( int low , int high)*  *{*  *int i,j;*  *int pivot=A[low];*  *i=low;*  *j=high+1;*  *while(i<j)*  *{*  *do*  *{*  *++i;*  *}while(A[i]<=pivot);*  *do*  *{*  *--j;*  *}while(A[j]>pivot);*  *if(i<j)*  *{*  *swap(&A[i],&A[j]);*  *\*t=A[i];*  *A[i]=A[j];*  *A[j]=t;\*/*  *}*  *}*  *swap(&A[low],&A[j]);*  */\*t=A[low];*  *A[low]=A[j];*  *A[j]=t;\*/*  *return j;*  *}*  *void quicksorting( int lower , int higher)*  *{*  *int j;*  *if(lower<higher)*  *{*  *j=part(lower,higher);*  *quicksorting(lower , j-1);*  *quicksorting(j+1 , higher);*  *}*  *}*  *void swap(int \*a , int \*b)*  *{*  *int t;*  *t= \*a;*  *\*a=\*b;*  *\*b=t;*  *}* |
| **Sample Input:** |
| 8 6 7 1 2 5 |
| **Sample Output:** |
| 1 2 5 6 7 8 |
| **Time complexity calculation:** |
| O(n\*logn) |

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

*<Write your answer here>*

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

*<Write your answer here>*

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

*<Write your answer here>*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Dynamic Programming** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *1. Start the program.*  *2. Declare constants: MAX (maximum size of the matrix).*  *3. Declare function prototypes:*  *• void Floyd(int W[][10], int n)*  *4. Declare variables: i, j, k, n (integers), W[MAX][MAX] (2D array).*  *5. Print "Floyd's Algorithm" to the console.*  *6. Print "Enter the number of nodes: " to the console.*  *7. Read the value of n from the user.*  *8. Print "Enter the edge weights:" to the console.*  *9. Iterate from i = 1 to n:*  *• Iterate from j = 1 to n:*  *• Read the value of W[i][j] from the user.*  *10. Call the Floyd function with arguments W and n.*  *11. End the program.*  *Function Floyd:*  *1. Start the function.*  *2. Declare variables: i, j, k (integers).*  *3. Iterate from k = 1 to n:*  *• Print "matrix D[k]" to the console.*  *• Iterate from i = 1 to n:*  *• Iterate from j = 1 to n:*  *• If (W[i][k] + W[k][j]) is less than W[i][j], then update W[i][j] with (W[i][k] + W[k][j]).*  *• Print W[i][j] to the console.*  *• Print a new line.*  *4. End the function.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  int findfibonnaci(int n)  {  int i, fibno[n];  fibno[0] = 0;  fibno[1] = 1;  for (i = 2; i < n; i++)  {  fibno[i] = fibno[i - 1] + fibno[i - 2];  }  return fibno[n - 1];  }  int main()  {  int n, result;  printf("Enter which of the Fibonacci number to be found: ");  scanf("%d", &n);  result = findfibonnaci(n);  printf("%dth number is %d\n", n, result);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Enter which of the Fibonacci number to be found: 5 | | | | | | | |
| **Sample Output:** | | | | | | | |
| *5th number is 3* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #include <stdlib.h>  #define MAX 10  void Floyd(int W[][10],int n)  {  int i,j,k;  for(k=1;k<=n;k++)  { printf("matrix D[%d]\n", k) ;  for(i=1;i<=n;i++)  { for(j=1;j<=n;j++)  {  if((W[i][k]+W[k][j])<W[i][j])  W[i][j]=W[i][k]+W[k][j];  printf("%d\t", W[i][j]);  }  printf("\n");  }  }  }  int main()  {  int i,j,n;  int W[MAX][MAX];  printf("Floyds Algorithm follows \n");  printf("Enter number of nodes :");  scanf("%d",&n);  printf("\n Enter the edge weights :");  for(i=1;i<=n;i++)  {for(j=1;j<=n;j++)  {scanf("%d",&W[i][j]);  Floyd(W,n);  }Floyd(W,n);}  return 0;  } |
| **Sample Input:** |
| Floyds Algorithm follows  Enter number of nodes :2  Enter the edge weights :3 |
| **Sample Output:** |
| matrix D[1]  3 0  0 0  matrix D[2]  0 0  0 0 |
| **Time complexity calculation:** |
| O(n^3) |

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

*<Write your answer here>*

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

*<Write your answer here>*

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

*<Write your answer here>*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Greedy** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *ALGORITHM Kruskal(G)*  *// Kruskal’s algorithm to construct a minimum spanning tree*  *// Input: A weighted connected graph G(V, E)*  *// Output: ET, the set of edges composing of MST of G*  *sort E in nondecreasing order of the edge weights w(ei1) <= … <= w(ei|E|)*  *ET  Ø*  *ecounter  0*  *k  0*  *while ecounter < |V| - 1 do*  *k  k + 1*  *if ET U {eik} is acyclic*  *ET ET U {eik}*  *ecounter ecounter + 1*  *return ET* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include<stdio.h>  struct Node {  int u;  int v;  int w;  };  struct Node arr[14] = {  {4, 2, 5},  {2, 8, 4},  {6, 6, 2},  {1, 3, 0},  {2, 5, 8},  {6, 8, 12},  {47, 14, 96},  {71, 83, 14},  {17, 21, 16},  {60, 7, 16},  {3, 4, 18},  {4, 5, 20},  {1, 7, 22},  {3, 5, 28}  };  void sort() {  int n = 14;  struct Node t;  int i, j;  // Bubble sort algorithm to sort the array based on weight  for (i = 0; i <= n - 2; i++) {  for (j = 0; j <= n - i - 2; j++) {  if (arr[j].w > arr[j + 1].w) {  t = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = t;  }  }  }  }  void union1(int a[], int u, int v) {  int t = a[u];  int i;  // Union operation in disjoint set  for (i = 0; i < 9; i++) {  if (a[i] == t) {  a[i] = a[v];  }  }  }  int find(int a[], int u, int v) {  if (a[u] == a[v]) {  return 1;  }  return 0;  }  int mst(int a[]) {  int s = 0;  int c = 0;  int k, i;  // Iterate over the sorted array of edges  for (i = 0; i < 14; i++) {  if (c > 8)  break;  k = find(a, arr[i].u, arr[i].v);  // If the vertices belong to different sets, add the edge to MST  if (k == 0) {  printf("%d,%d\n", arr[i].u, arr[i].v);  union1(a, arr[i].u, arr[i].v);  s = s + (arr[i].w);  c++;  }  }  return s;  }  int main() {  int a[9] = {0, 1, 2, 3, 4, 5, 6, 7, 8};  sort();  int ans = mst(a);  printf("\n%d", ans);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| 1,3  2,8  4,2  2,5  6,8  60,7  3,4  1,7  47,14 | | | | | | | |
| **Sample Output:** | | | | | | | |
| *181* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n^2) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *// Prints a maximum set of activities that can be done by a single person, one at a time.*  *void primeactsmax(int start[], int finish[], int n)*  *{*  *int i, j;*  *printf("Following activities are selected:\n");*  *// The first activity always gets selected*  *i = 0;*  *printf("%d ", i);*  *// Consider the rest of the activities*  *for (j = 1; j < n; j++) {*  *// If the start time of the current activity is greater than or equal to the finish time*  *// of the previously selected activity, then select it*  *if (start[j] >= finish[i]) {*  *printf("%d ", j);*  *i = j;*  *}*  *}*  *}*  *// Driver code*  *int main()*  *{*  *int start[] = { 3,7,7,8,5,6,0};*  *int finish[] = { 4,6,0,9,6,4,2};*  *int n = sizeof(start) / sizeof(start[0]);*  *// Function call*  *primeactsmax(start, finish, n);*  *return 0;*  *}* |
| **Sample Input:** |
| Start->{ 3,7,7,8,5,6,0};  Finish->{ 4,6,0,9,6,4,2} |
| **Sample Output:** |
| Following activities are selected:  0 1 2 3 |
| **Time complexity calculation:** |
| O(n) |

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

*<Write your answer here>*

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

*<Write your answer here>*

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

*<Write your answer here>*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Branch and Bound** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *1. Start at an arbitrary city (the starting city).*  *2. Select the nearest unvisited city and add it to the route.*  *3. Mark the selected city as visited.*  *4. Repeat steps 2 and 3 until all cities have been visited.*  *5. Add the starting city to the route, completing the cycle.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  #include <conio.h>  int costMatrix[10][10], visited[10], numberOfCities, totalCost = 0;  void getInputs() {  int i, j;  printf("Enter the number of cities: ");  scanf("%d", &numberOfCities);  printf("\nEnter the cost matrix: \n");  for (i = 0; i < numberOfCities; i++) {  printf("\nEnter elements of row %d: \n", i + 1);  for (j = 0; j < numberOfCities; j++)  scanf("%d", &costMatrix[i][j]);  visited[i] = 0;  }  printf("\n\nThe cost matrix is:\n\n");  for (i = 0; i < numberOfCities; i++) {  printf("\n\n");  for (j = 0; j < numberOfCities; j++)  printf("\t % d", costMatrix[i][j]);  }  }  void findMinimumCostPath(int cityIndex) {  int i, nextCity;  visited[cityIndex] = 1;  printf("%d –>", cityIndex + 1);  nextCity = getMinimumCost(cityIndex);  if (nextCity == 999) {  nextCity = 0;  printf("%d", nextCity + 1);  totalCost += costMatrix[cityIndex][nextCity];  return;  }  findMinimumCostPath(nextCity);  }  int getMinimumCost(int cityIndex) {  int i, nextCityIndex = 999;  int minimumCost = 999, tempMinimumCost;  for (i = 0; i < numberOfCities; i++) {  if ((costMatrix[cityIndex][i] != 0) && (visited[i] == 0))  if (costMatrix[cityIndex][i] < minimumCost) {  minimumCost = costMatrix[i][0] + costMatrix[cityIndex][i];  tempMinimumCost = costMatrix[cityIndex][i];  nextCityIndex = i;  }  }  if (minimumCost != 999)  totalCost += tempMinimumCost;  return nextCityIndex;  }  void displayOutput() {  printf("\n\nMinimum cost:");  printf("%d", totalCost);  }  int main() {  getInputs();  printf("\n\nThe path is:\n\n");  findMinimumCostPath(0);  displayOutput();  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Enter No. of Cities: 6  Enter Cost Matrix:  99 10 15 20 99 8  5 99 9 10 8 99  6 13 99 12 99 5  8 8 9 99 6 99  99 10 99 6 99 99  10 99 5 99 99 99    Enter Elements of Row# : 1  Enter Elements of Row# : 2  Enter Elements of Row# : 3  Enter Elements of Row# : 4  Enter Elements of Row# : 5  Enter Elements of Row# : 6 | | | | | | | |
| **Sample Output:** | | | | | | | |
| The Path is:    1 –>6 –>3 –>4 –>5 –>2 –>1    Minimum cost:46 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n^2) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #define N 4  int cost[N][N] = {{0, 10, 15, 20},  {10, 0, 35, 25},  {15, 35, 0, 30},  {20, 25, 30, 0}};  int visited[N] = {0};  int best\_cost = INT\_MAX;  void tsp(int node, int current\_cost, int level) {  if (level == N) {  current\_cost += cost[node][0]; // Add cost to return to starting node  if (current\_cost < best\_cost) {  best\_cost = current\_cost;  }  return;  }  visited[node] = 1;  for (int i = 0; i < N; i++) {  if (!visited[i]) {  tsp(i, current\_cost + cost[node][i], level + 1);  }  }  visited[node] = 0;  }  int main() {  tsp(0, 0, 1);  printf("The minimum cost for TSP is %d\n", best\_cost);  return 0;  } |
| **Sample Input:** |
| {0, 10, 15, 20},  {10, 0, 35, 25},  {15, 35, 0, 30},  {20, 25, 30, 0} |
| **Sample Output:** |
| The minimum cost for TSP is 80 |
| **Time complexity calculation:** |
| O(N!) |

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

*<Write your answer here>*

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

*<Write your answer here>*

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

*<Write your answer here>*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Brute force | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *1. Start the program.*  *2. Declare function prototypes:*  *• int search(char\* pat, char\* txt)*  *3. Declare variables: txt (string to search within), pat (pattern to search for), c (integer).*  *4. Assign the text "AABAACAADAABAAABAA" to the variable txt.*  *5. Assign the pattern "AACA" to the variable pat.*  *6. Call the search function with arguments pat and txt and assign its result to c.*  *7. Print "Pattern found at index c" to the console, where c is the value returned by the search function.*  *8. End the program.*  *Function search:*  *1. Start the function.*  *2. Declare variables: M (length of the pattern), N (length of the text), j (integer).*  *3. Set M to the length of the pattern.*  *4. Set N to the length of the text.*  *5. Iterate from i = 0 to N - M:*  *• Set j to 0.*  *• While j is less than M and pat[j] is equal to txt[i + j], do the following:*  *• Increment j by 1.*  *• If j is equal to M, return i (starting index of the pattern in the text).*  *6. Return -1 (pattern not found).*  *7. End the function.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  #include <string.h>    int search(char\* pat, char\* txt)  {  int M = strlen(pat);  int N = strlen(txt);  int j;  for (int i = 0; i <= N - M; i++) {  j=0;  while(j<M && pat[j]==txt[i+j]){  j=j+1;;  if(j==M){  return i;  }  }  }  return -1;  }  // Driver's code  int main()  {  char txt[] = "AABAACAADAABA8";  char pat[] = "AACA";  int c=search(pat, txt);  printf("Pattern found at index %d \n", c);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| ="AABAACAADAABA8"  "AACA" | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Pattern found at index 3* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O((N-M+1)M) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *#define MAX 10*  *int n;*  *int dist[MAX][MAX];*  *void input()*  *{*  *printf("Enter the number of cities: ");*  *scanf("%d", &n);*  *printf("Enter the distance matrix:\n");*  *for(int i = 0; i < n; i++) {*  *for(int j = 0; j < n; j++) {*  *scanf("%d", &dist[i][j]);*  *}*  *}*  *}*  *int permutation[MAX];*  *int visited[MAX];*  *int min\_cost = 1e9;*  *void dfs(int pos, int cost)*  *{*  *if (pos == n) {*  *if (cost + dist[permutation[n - 1]][permutation[0]] < min\_cost) {*  *min\_cost = cost + dist[permutation[n - 1]][permutation[0]];*  *}*  *return;*  *}*  *for (int i = 0; i < n; i++) {*  *if (!visited[i]) {*  *visited[i] = 1;*  *permutation[pos] = i;*  *dfs(pos + 1, cost + dist[permutation[pos - 1]][i]);*  *visited[i] = 0;*  *}*  *}*  *}*  *void solve()*  *{*  *dfs(1, 0);*  *}*  *void output()*  *{*  *printf("Minimum cost: %d\n", min\_cost);*  *}*  *int main()*  *{*  *input();*  *solve();*  *output();*  *return 0;*  *}* |
| **Sample Input:** |
| Enter the number of cities: 3  Enter the distance matrix:  1  2  3  4  5  6  7  8  9 |
| **Sample Output:** |
| Minimum cost: 7 |
| **Time complexity calculation:** |
| O(n!) |

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

*<Write your answer here>*

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

*<Write your answer here>*

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

*<Write your answer here>*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Randomized | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *1. Start the program.*  *2. Declare constants: SEED (seed for random number generation).*  *3. Declare variables: i, count, total\_points (integers), x, y, z, pi (doubles).*  *4. Set the seed for random number generation using the current time.*  *5. Print "Enter total number of points: " to the console.*  *6. Read the value of total\_points from the user.*  *7. Initialize the count to 0.*  *8. Iterate i from 0 to total\_points - 1:*  *• Generate a random number between 0 and 1 and assign it to x.*  *• Generate another random number between 0 and 1 and assign it to y.*  *• Calculate z as the square of x plus the square of y.*  *• If z is less than or equal to 1, increment the count by 1.*  *9. Estimate the value of pi by dividing the count by total\_points and multiplying the result by 4. Assign it to pi.*  *10. Print "Approximate value of PI = " followed by the value of pi to the console.*  *11. End the program.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  void random\_permutation(int n) {  int i, j, temp;  int \*perm = malloc(n \* sizeof(int));  srand(time(NULL)); // seed the random number generator  // initialize the permutation  for (i = 0; i < n; i++) {  perm[i] = i + 1;  }  // randomly permute the array  for (i = n - 1; i > 0; i--) {  j = rand() % (i + 1);  temp = perm[i];  perm[i] = perm[j];  perm[j] = temp;  }  // print the permutation  printf("Random permutation:");  for (i = 0; i < n; i++) {  printf(" %d", perm[i]);  }  printf("\n");  free(perm);  }  int main() {  int n;  printf("Enter the number of integers to permute: ");  scanf("%d", &n);  random\_permutation(n);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Enter the number of integers to permute: 6 | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Random permutation: 2 4 5 1 6 3* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include<stdio.h>*  *void swap(int\* a, int\* b) {*  *int temp;*  *temp = \*a;*  *\*a = \*b;*  *\*b = temp;*  *}*  *int partition(int\* arr, int low, int high) {*  *int pivot = arr[high]; // Last element is the pivot*  *int i = low - 1;*  *for (int j = low; j <= high - 1; j++) {*  *// Compares each element from the beginning with the pivot (last element)*  *// If the value is less than the pivot, swaps it with the element at position i+1*  *if (arr[j] < pivot) {*  *i++;*  *swap(&arr[i], &arr[j]);*  *}*  *}*  *// After all the swaps are done, swaps the pivot with the element at position i+1*  *swap(&arr[i + 1], &arr[high]);*  *return i + 1;*  *}*  *void quicksort(int\* arr, int low, int high) {*  *int pi;*  *if (low < high) {*  *// pi is the partition index - arr[pi] is now at the right place*  *pi = partition(arr, low, high);*  *// Recursively sorts the left and right halves of the partitioned array*  *quicksort(arr, low, pi - 1); // for left half*  *quicksort(arr, pi + 1, high); // for right half*  *}*  *}*  *int main() {*  *int arr[] = { 3, 4, 1, 2, 5 };*  *printf("Initial array: ");*  *for (int i = 0; i < 5; i++) {*  *printf("%d\t", arr[i]);*  *}*  *printf("\n");*  *quicksort(arr, 0, 4);*  *printf("Sorted array: ");*  *for (int i = 0; i < 5; i++) {*  *printf("%d\t", arr[i]);*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| Initial array: 3 4 1 2 5 |
| **Sample Output:** |
| Sorted array: 1 2 3 4 5 |
| **Time complexity calculation:** |
| O(n^2) |